Marino Posadas, Tadit Dash

Dependency Injection in .NET Core 2.0

Make use of constructors, parameters, setters, and interface injection to write reusable and loosely-coupled code

Packt

WOW! eBook
www.wowebook.org

Dependency Injection in .NET Core 2.0

Make use of constructors, parameters, setters, and interface injection to write reusable and loosely-coupled code

Marino Posadas
Tadit Dash

WOW! eBook
www.wowebook.org

Packt

BIRMINGHAM - MUMBAI

WOW! eBook
www.wowebook.org

Dependency Injection in .NET Core 2.0

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2017

Production reference: 1071117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street

WOW! eBook
www.wowebook.org

Birmingham
B3 2PB, UK.

ISBN 978-1-78712-130-0

www.packtpub.com

WOW! eBook
www.wowebook.org

Credits

Authors Marino Posadas Tadit Dash	**Copy Editor** Safis Editing
Reviewer Gaurav Aroraa	**Project Coordinator** Prajakta Naik
Commissioning Editor Aaron Lazar	**Proofreader** Safis Editing
Acquisition Editor Denim Pinto	**Indexer** Pratik Shirodkar
Content Development Editor Lawrence Veigas	**Graphics** Jason Monteiro
Technical Editor Tiksha Sarang	**Production Coordinator** Aparna Bhagat

WOW! eBook
www.wowebook.org

About the Authors

Marino Posadas is an independent senior trainer, writer, and consultant in Microsoft Technologies and Web Standards. He is a Microsoft MVP in C#, Visual Studio, and Development Technologies; an MCT, MCPD, MCTS, MCAD, and MCSD; and he was the former Director of Development in Spain and Portugal for Solid Quality Mentors.

Marino has published 15 books and more than 500 articles on development technologies in several magazines and online publications. The topics covered in his books range from Clipper and Visual Basic 5.0/ 6.0 to C # and .NET- safe programming, programming with Silverlight 2.0 and 4.0, and Web Standards.

His latest books are *Mastering C#* and *.NET Framework*, by Packt, and *The Guide to Programming in HTML5, CSS3, and JavaScript with Visual Studio*. He is also a speaker at Microsoft events, having lectured in Spain, Portugal, England, the US, Costa Rica, and Mexico.

His website also contains developer's resources and videos, in English and Spanish, interviewing representatives of the Microsoft and Web Standards development world.

You can follow him on Twitter at `@MarinoPosadas`.

I'd like to thank Dominic Pereira, Denim Pinto, Lawrence Veigas, and Tiksha Sarang from Packt for their continuous support and confidence while writing this book. Special thanks to some professionals and technology evangelists whose work inspired different parts of this book, in particular, Mark Russinowich, Scott Hanselman, Scott Hunter (the "lesser" Scotts), and, of course, Robert Martin and Martin Fowler, pioneers in this work. Also, I would like to remember my MVP lead, Cristina González Herrero, for her continuous encouragement and help, and other people at Microsoft who have always supported my activities. My memories go here to Alfonso Rodríguez, David Carmona, David Salgado, and Leon Welicki. My appreciation also goes to my mates at Netmind for their support in this initiative from the beginning. I dedicate this book to my wife, Milagros, who makes everything possible.

WOW! eBook
www.wowebook.org

Tadit Dash is a software engineer by profession. As a software engineer, he usually works for 8 to 9 hours daily. Besides his daily work, he contributes to both online and offline communities. He co-founded the first technical community in his state, named Microsoft Developers Community Odisha, which is devoted to spreading awareness of the newest trends in technology among developers. This community organizes events and workshops in orphanages, schools, and colleges.

He writes articles and blogs and creates demos and videos for fellow programmers. Answering questions on online forums and social channels are the activities he enjoys the most. Due to his exceptional contribution to the technical community, Microsoft has awarded him with the Microsoft Most Valuable Professional accolade since 2014. CodeProject has awarded him the CodeProject MVP accolade (the first from Odisha and three times in a row for the years 2014, 2015, and 2016). For his constant mentorship, IndiaMentor featured him as a Young Mentor on their site.

He was recognized by DZone and awarded the Most Valuable Blogger accolade. He was awarded the *Star* and *Achiever* accolade from his organization. Motivating students in his sessions is something he is popular for. He is a regular technical and motivational speaker. He has spoken at many local events organized by different communities. He was a featured speaker in DevTechDay Nepal.

I would like to dedicate this book to my grandfather, the late Mr. Ganeswar Tripathy.
My grandmother Santipriya Tripathy, father, Dr. Gobinda Chandra Dash, mother, Mrs. Sasmita Tripathy, and brother, Tworit Dash who made sure that I am high on energy (with healthy food) and confidence all the time during writing. My uncles, Mr. Anil Tripathy, Mr. Aswin Tripathy, and Mr. Amiya Tripathy who always guide me to shape the approach to my new challenges. Mrs. Sujata Tripathy, who takes cares of me like my mother and Mr. Debendra Rath, my uncle, who never misses an opportunity to discuss ideas to strengthen my thoughts so that I deliver the best.

My guide and mentor, Mr. Gaurav Kumar Aroraa (who is a Microsoft MVP and renowned author), encouraged me to take up the project, and constantly motivated me throughout the preparation. I am so grateful to the Packt team, especially Mr. Lawrence Veigas (Content Development Editor), and Miss. Tiksha Sarang who corrected me in every chapter. Thanks to Mr. Nirmal Hota (mobile guru of Odisha), who is like my big brother, for his consistent efforts in throwing me out of my

WOW! eBook
www.wowebook.org

comfort zone while patting my back. Miss. Prakreeti Prasanna, as a well wisher and best friend, always adds an x-factor to my activities. All my friends, family members, colleagues, and Google, of course, are the ingredients for the success of this book. Special thanks to my community members, Microsoft Developers Community, Odisha, for waiting to hear the surprise. Last, but not the least, Miss. Jayashree Satapathy, who, being a part of my life, has taken every care to make my writing schedule smooth.

WOW! eBook
www.wowebook.org

About the Reviewer

Gaurav Aroraa has his M.Phil in computer science. He is a Microsoft MVP, lifetime member of the Computer Society of India (CSI), Advisory member of IndiaMentor, certified scrum trainer/coach, XEN for ITIL-F and APMG for PRINCE-F and PRINCE-P. He is an Open Source developer, contributor to TechNet Wiki, and the founder of Innatus Curo Software LLC. For over 19 years of his career, he has mentored thousands of students and industry professionals. You can tweet Gaurav on his twitter handle at `@g_arora`.

He has authored the following books:

- Building Microservices in ASP.NET Core, Packt Publishing
- Learn C# in 7 Days, Packt Publishing
- SOLID Principles Succinctly, Syncfusion
- ASP.NET WebHooks Succinctly, Syncfusion

To my wife, Shuby Arora, and my angel (daughter), Aarchi Arora, who permitted me to steal some time for this book from the time I was supposed to spend with them. To the entire Packt team, especially Prajakta, whose coordination and communication during the period was tremendous.

WOW! eBook
www.wowebook.org

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Mapt

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

WOW! eBook
www.wowebook.org

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

WOW! eBook
www.wowebook.org

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1787121305.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

WOW! eBook
www.wowebook.org

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
  - Downloading the example code
  - Downloading the color images of this book
  - Errata
  - Piracy
  - Questions

1. The SOLID Principles of Software Design
   - In the beginning
     - .NET and .NET Core
       - About .NET Core
       - ASP.NET Core
       - About the IDE used in this book
       - Other foundational changes in .NET Core
   - The SOLID principles
     - The Single Responsibility Principle (SRP)
       - The background of the Separation of Concerns (SoC)
       - Well-known examples of Separation of Concerns
       - A basic sample of Separation of Concerns
   - Another sample
   - The Open/Closed principle
   - Back to our sample
   - The Liskov Substitution principle
     - Back to the code again
     - Other implementations of LSP in .NET
   - The Interface Segregation principle
   - Another sample
   - The Dependency Inversion principle
   - The (more or less) canonical example
   - Other ways to implement Dependency Inversion
   - Summary
2. Dependency Injection and IoC Containers
   - The Dependency Principle in more detail
     - Let's look at an example
   - Aspects of Dependency Injection

WOW! eBook
www.wowebook.org

Object Composition
Object Lifetime
Interception
Ways to implement DI
Property Injection in action
Method Injection in practice
.NET event architecture as Model Injection
DI containers
Service locators
DI Containers for .NET
Using Unity Container
Using Castle Windsor
Using StructureMap
Using Autofac
Dependency Injection in other frameworks
DI in Angular
Our first demo
Summary
3. Introducing Dependency Injection in .NET Core 2.0
The main characteristics of .NET Core
The main advantages of .NET Core
Installation of .NET Core in the IDE
Installation path for .NET Core in Visual Studio 2015
.NET Core in Visual Studio 2017
Types of deployment
Checking other dependencies in .NET Core
Dependency Injection in .NET Core
DI Architecture and Object's Lifetime
The class ActivatorUtilities and other helpers
The Microsoft.Extensions.DependencyInjection container
Checking the object's lifetime
Mapping interfaces to instance classes
The concept of Scope Applied to services
Other extensions with DI functionality
A reflection on the structure of .NET Core
LogLevels Severity
Summary
4. Dependency Injection in ASP.NET Core
ASP.NET Core from the command-line tools
Creating the minimum possible application
Changing to ASP .NET Core
Middleware
The Startup class and Dependency Injection
Code explanation

WOW! eBook
www.wowebook.org

The New ASP.NET servers
Kestrel
WebListener
Dependency Inversion in the architecture - OWIN
Analyzing the default template
Configuration files
The Entry Point
The default Startup class
Dependency Injection in ASP.NET Core
Services provided by ASP.NET Core
The relation of services available at startup
Identifying Services in the Web Application template
Using Dependency Injection in ASP.NET MVC Views
Garbage Collection and Custom Services
Using Custom Services through Dependency Injection
Services and data management
Using Dependency Injection inside Views
Summary
5. Object Composition
Understanding object relationships
Object Composition
Types of Object Composition
Composition
Consider an example
Contrasting features of Composition
Why are subclasses inside Composition?
Important notes
Aggregation
Consider an example
Composition versus Aggregation
Advantages and disadvantages
Other important relationships
Association
Example
Explanation of the preceding code
Types of Association
Reflexive Association
Indirect Association
Composition over Inheritance
Inheritance
Example on User class
New User Type
Problem which we come across
The solution to the problem
Role of Object Composition in Dependency Injection
Composition Root

WOW! eBook
www.wowebook.org

Composing .NET Core 2.0 Console application
Object Composition in ASP.NET Core MVC 2.0
Summary
6. Object Lifetime
Manage the object life cycle
Stack versus heap
Managed and unmanaged resources
Generations
Object creation
The Copy constructor
Object destruction
Finalize
The IDisposable interface
Consider an example
Implementing the IDisposable Interface
Step1 - Basic structure of class
Step2 - Defining a Dispose Overload method
Step3 -Modifying the Dispose(bool) for Derived classes
Step 4 - Handling duplicate Dispose calls
Object lifetime management in .NET Core
Object creation
Designing the interfaces
The Concrete class
The Service class
The controller
View
Startup ConfigureServices
Object lifetimes
Singleton
Scoped
Transient
Instance
Object disposal
When to choose what?
Relationship and dependencies among lifetimes
Singleton depending on Scoped and Transient
Scoped depending on Singleton and Transient
Transient depending on Singleton and Scoped
Summary
7. Interception
Introducing Interception
The decorator
Patterns and principles for Interception

WOW! eBook
www.wowebook.org

Benefits of the Decorator approach
Problems of the Decorator approach
Aspect-oriented programming
Cross-cutting concerns
Aspect
Aspect characteristics
Advantages
Aspect attached locations
Types of AOP
Investigating Interception
The Interception process
Castle Windsor
Demonstration for using the Castle Windsor
Creating an Interceptor
Attaching an Interceptor
Intermediate Language (IL) Weaving
IL Weaving process
Creating an aspect
Attaching the aspect
Interception in ASP.NET Core
Filters
Global filter
Attributes
Middleware
Registration
Execution
Ordering
Summary
8. Patterns - Dependency Injection
Dependency Inversion Principle
Problems
Solution
Inversion of Control (IoC)
Patterns
Constructor Injection pattern
The problem
The solution
Curveball
Injecting with .NET Core 2.0
Implementation inside ASP.NET Core 2.0
ControllerActivatorProvider.cs
Importance of the Constructor Injection pattern
Advantages of the Constructor Injection pattern
Disadvantages of the Constructor Injection pattern

WOW! eBook
www.wowebook.org

Property Injection pattern
Curveball
Advantages of the Property Injection Pattern
Disadvantages of the Property Injection Pattern
Method Injection Pattern
Curveball
Injecting with .NET Core 2.0
Implementation inside .NET Core 2.0
MvcServiceCollectionExtensions.cs
Ambient context
Curveball
Implementation inside .NET Core 2.0
The advantages of the Ambient Context
The disadvantages of the Ambient Context
Summary
9. Anti-Patterns and Misconceptions on Dependency Injection
When does Dependency Injection become an anti-pattern?
Anti-patterns
Control Freak
Problem
Concrete Factory
Abstract Factory
Static Factory
Solution
Poor Man's DI
The approach
Problem
Solution
Bastard Injection
Problem
Solution
Constrained Construction
Problem
Solution
Service Locator
Design
Advantages
Problem
Code reusability
Solution
Refactoring steps
Summary
10. Dependency Injection in Other JavaScript Frameworks
TypeScript
Architectural changes

WOW! eBook
www.wowebook.org

Modules in TypeScript
External modules
Dependency Injection in TypeScript
Angular
AngularJS
Examples using Visual Studio 2017
Understanding the structure of AngularJS bootstrap
Data access and Dependency Injection
Summing up Dependency Injection features inside AngularJS
Angular 2+
Microsoft's TypeScript is the preferred language in Angular 2+
Angular tools
Working with Angular
Editing the initial project
The structure of the main module
DI in Angular 4
The concept of a provider
Wrapping it up
Summary
11. Best Practices and Other Related Techniques
Tightly coupled systems
The problem
The solution - refactoring with DI
Interface extraction
Layered architecture
The problem - The concrete class reference again
Misuse of business and Data Access Layer assemblies
Best practice for layered architecture
Managed Extensibility Framework (MEF)
Layers and MEF implementation
Introduction of IUser
The IModuleRegistrar interface
The ModuleRegistrar class
The IModule interface
The ModuleInit class
The ModuleLoader class
Executing the ModuleLoader.LoaderContainer() method from Web App
What have we achieved with MEF?
Layers are separated
All classes inside layers are now internal
No concrete class instantiation inside the UI Layer
More layers can be added to the architecture
Conclusion
Summary

WOW! eBook
www.wowebook.org

Preface

This book is an approach to the implementation of Dependency Injection techniques across the new .NET Core 2.0 version. The designers of .NET Core implemented plenty of functionalities related to good practices and have followed the principles stated by Robert C. Martin (the SOLID principles) in distinct areas of this version.

The purpose of this work is to go through those well-stated principles and identify and show through examples how they're implemented, and how they can be used by the programmer.

WOW! eBook
www.wowebook.org

What this book covers

Chapter 1, *The SOLID Principles of Software Design*, introduces you to the five SOLID principles and how they are found or can be easily implemented in .NET Core 2.0.

Chapter 2, *Dependency Injection and IoC Containers*, gives you exposure to how Dependency Injection should be used either by itself or with the help of third-party containers.

Chapter 3, *Introducing Dependency Injection in .NET Core 2.0*, reviews the real implementation of DI inside .NET Core 2.0 from the point of view of Console applications.

Chapter 4, *Dependency Injection in ASP.NET Core*, provides a more detailed study of the implementation of DI techniques inside web applications that use ASP.NET Core 2.0, full of samples.

Chapter 5, *Object Composition,* takes you through all the hidden principles behind the concept of object composition and how it is applied in .NET Core 2.0 and ASP.NET Core MVC 2.0, forming a pillar of DI.

Chapter 6, *Object Lifetime,* the next DI pillar provides a deep dive into lifestyles maintained and typical management strategies, by the objects with DI in place, which enables better decision making with optimized configurations.

Chapter 7, *Interception,* the last pillar of DI ecosystem, provides techniques to intercept calls and insert code dynamically into the pipeline. This chapter also deals with interception's application in .NET Core 2.0 and ASP.NET Core 2.0 with proper illustrations.

Chapter 8, *Patterns – Dependency Injection,* walks you through the D of SOLID and all the important techniques to apply DI in applications with .NET Core 2.0.

Chapter 9, *Anti-Patterns and Misconceptions on Dependency Injection,* deals with the common bad usage of DI patterns and scenarios to avoid while coding, in order to get a good outcome from DI in the application.

WOW! eBook
www.wowebook.org

Chapter 10, *Dependency Injection in Other JavaScript Frameworks,* teaches you about Dependency Injection techniques using other popular frameworks, such as Angular.

Chapter 11, *Best Practices and Other Related Techniques*, covers the well-proven coding, architectural, and refactoring practices that you should adopt while applying DI in your current and legacy applications.

WOW! eBook
www.wowebook.org

What you need for this book

You will need Visual Studio 2017 Community Edition, the Chrome Navigator, and IIS (Internet Information Server Express) to successfully test and execute all code files.

WOW! eBook
www.wowebook.org

Who this book is for

This book is for C# and .NET developers who have no idea what DI (Dependency Injection) is and would like to understand how to implement it in their applications.

WOW! eBook
www.wowebook.org

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "For this new requirement, we can create a new and overloaded the `ReadData()` method that receives an extra argument."

A block of code is set as follows:

```
public Student(int id, string name, DateTime dob)
{
  Id = id;
  Name = name;
  Dob = dob;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
static void Main(string[] args)
{
  Packt packt = new Packt
  {
     Name = "Packt Publications",
     OfficialFacebookLink = "https://www.facebook.com/PacktPub/",
     TotalBooksPublished = 5000
  };

  packt.PrintPacktInfo();
  Console.ReadKey();
}
```

Any command-line input or output is written as follows:

```
npm install -g json
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Double-click on Add ArcGIS Server."

Warnings or important notes appear like this.

WOW! eBook
www.wowebook.org

Tips and tricks appear like this.

WOW! eBook
www.wowebook.org

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

WOW! eBook
www.wowebook.org

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

WOW! eBook
www.wowebook.org

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Dependency-Injection-in-.NET-Core-2.0. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

WOW! eBook
www.wowebook.org

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/DependencyInjectioninNETCore20_ColorImages.pdf.

WOW! eBook
www.wowebook.org

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

WOW! eBook
www.wowebook.org

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

WOW! eBook
www.wowebook.org

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

WOW! eBook
www.wowebook.org

The SOLID Principles of Software Design

This book focuses on techniques related to Dependency Injection and the way those techniques are implemented by default and can be extended by the programmer in .NET Core--the first version of .NET that executes on every platform.

It works on Windows, macOS, and Linux distro on the desktop, and the idea can even be extended to the mobile world covering the Apple, Android, and Tizen (Samsung) operating systems.

This is, with no doubt, the most ambitious project from Microsoft in its search for a universal coverage of programming technologies and tools, and it can be considered a natural step after the initial **UWP** (**Universal Windows Platform**) project that allows building applications for any device supporting Windows, from IoT devices to the desktop, XBOX, or HoloLens.

So, in this chapter we'll start with a quick review of the main architectural components of .NET Core and its derivative frameworks (such as ASP.NET Core), to be followed with the foundations on which Dependency Injection techniques are based, as part of the SOLID principles, stated by Robert C. Martin (*Uncle Bob*) in 2000. (See Wikipedia: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).)

Therefore, we'll revise those five SOLID principles, explaining their purpose and advantages, together with some basic implementations of each one of them in the C# language using Console applications coded in .NET Core. In all we'll see an explanation of each principle and its coverage:

- Separation of concerns (clearly implemented in the core infrastructure of .NET Core and also from the initial configuration of pipelines and middleware in ASP.NET Core)
- Open/Closed (already implemented in classic .NET Framework since version 3.0 and also present here)
- Liskov Substitution Principle (available in two ways--in a classic manner through the support of typecasting, and through the use of generics)
- Interface segregation: Explanation of Interface segregation and its advantages

WOW! eBook
www.wowebook.org

- Dependency Inversion: Explanation of the principle, its derivatives, and the concept of IoC containers

WOW! eBook
www.wowebook.org

In the beginning

The evolution of programming techniques is, somehow, related to language evolution. Once the initial (and, in some ways, chaotic) times had passed, the universality of computing became clear, and the need for good patterns and languages capable of affording large projects turned out to be evident.

The 70s marked the start of the adoption of other paradigms, such as procedural programming, and later on, **object-oriented programming** (**OOP**), proposed by Ole-Johan Dahl and Kristen Nygaard with the Simula language, when they both worked at the Norwegian Computing Center. They were given the Turing Award for these achievements, among other recognitions.

A few years later (around 1979), Bjarne Stroustrup created C with Classes, the prototype of what C++ is today because he found valuable aspects in the Simula language, but he thought that it was too slow for practical purposes, being the first OOP language that was universally adopted.

C++ originally had imperative features and object-oriented and generic ones, while also providing the ability to program for low-level memory manipulation. While it's true that it has become a de facto standard for building critical systems and applications, for many people it was not adequate for **LOB** (**Line of Business**) applications.

Years later, Java and the .NET platforms proposed a much easier and affordable solution for many programmers while still moving within the ordered space that object-oriented programming languages promote.

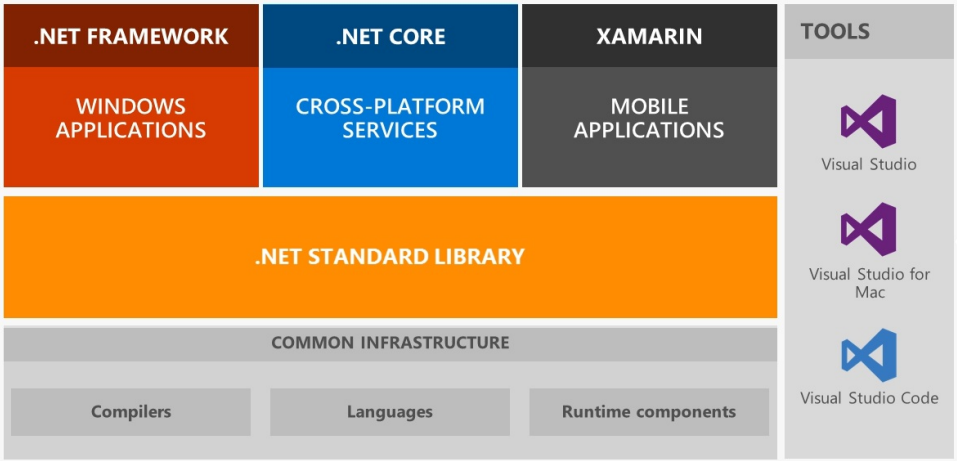
So, OOP was adopted, and so far no other important programming paradigm has replaced these ideas. Certainly, there are other approaches, such as functional programming, but even the most significant representative of this tendency, JavaScript, is becoming more object-oriented in the latest versions (ECMAScript 2015).

WOW! eBook
www.wowebook.org

.NET and .NET Core

.NET has been revamped lately in order to achieve the goal that Microsoft has pursued since Satya Nadella arrived in the company--""*Any Developer, Any App, Any Platforms.*"".

According to Principal Manager Scott Hunter, the company now presents a set of unified application models that can be summarized in the following screenshot:



Source: http://www.hanselman.com/blog/AnUpdateOnASPNETCore10RC2.aspx

As you see, the situation now is quite promising for a .NET Developer. The screenshot shows a **Common Infrastructure** (compilers, languages, and runtime components), powered by Roselyn services and other features. All those integrate with the IDEs that support these projects, now including Visual Studio for Mac.

On top of that lies a **.NET Standard Library**, which has points in common that allow us to share code along the three different frameworks--the classic **.NET Framework** (in version 4.6.2, at the time of writing this), **.NET Core** (now in version 2.0), and **Xamarin**, which allows building applications for any type of mobile target--Android, iOS, Windows Phone, and Tizen (Samsung).

WOW! eBook
www.wowebook.org

About .NET Core

.NET Core is the new version of .NET presented officially in the summer of 2016, and updated to version 1.1 in the November Connect() event the same year. It's defined as a *cross-platform, open source, cloud-ready and modular .NET platform for creating modern web apps, microservices, libraries, and console applications that run everywhere (Windows, Linux, and MacOS).*

It can be deployed along with the application itself, minimizing installation issues.

Prior to its publication, Microsoft decided to restart the numbering, reinforcing the idea that this is a totally new concept with respect to classical versions, as a better way to avoid ambiguities.

MSDN architect Cesar de la Torre defines in his blog very precisely the goals and structure of .NET Core--unlike the traditional .NET Framework, which is a single package installation, system-wide, and Windows-only runtime environment, .NET Core is about decoupling .NET from Windows, allowing it to run in non-Windows environments without having to install a giant 400 Mb set of binaries (versus just the footprint of the components you need from .NET Core) plus the ability to deploy applications accompanying the framework itself, supporting side-by-side execution of different versions of the framework.

A very interesting part of its architecture and deployment infrastructure, as mentioned in the same source, is that instead of being part of the operating system, .NET Core is composed of NuGet packages and is either compiled directly into an application or put into a folder inside the application. This means applications can carry .NET Core within and thus are completely side by side on the machine.

I, personally, think this is absolutely crucial for the project to be successful. No side-effects, no component installation in the target machine, and no dependencies. (As you'll see throughout this book this avoiding of dependencies is totally foundational when building software that follows good practices.)

NET Core 2.0 - Supported OS Versions Proposal:

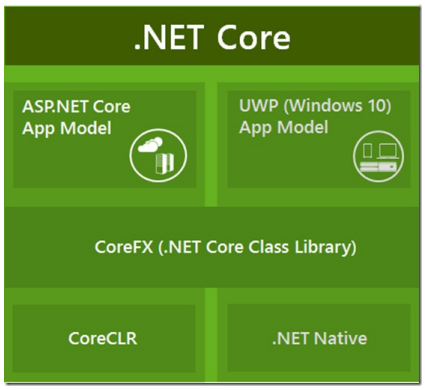
OS	Version	Architectures	Notes

WOW! eBook
www.wowebook.org

Windows Client	7 SP1+	x64, x86	
Windows Server	2008 R2 SP1+	x64, x86	Configurations: Full, Server Core, Nano
Windows IoT	10+	[C] arm32	IoT Core - see Raspberry Pi instructions
Red Hat Enterprise Linux	7.3+	x64	This includes Centos and Oracle Linux
Fedora	25+	x64	
Debian	8.7+	x64	Debian 9 (Stretch) workaround
Ubuntu	14.04+	x64, [C] arm32	This includes Linux Mint 17 for x64 For arm32, see Raspberry Pi instructions
openSUSE	42.2+	x64	
Tizen	4+	[S] arm32	Tizen .NET Developer Preview
Mac OS X	10.12+	x64	
In Progress OS's			
Arch Linux	[C] TBD	TBD	Blocked on missing OpenSSL 1.0 package in distro. Arch Linux community efforts tracked here.
FreeBSD & NetBSD	[C] TBD	TBD	Tracking main issue and label. NetBSD packages for .NET Core 1.0.0

As for the types of programmable project available from any of the above-mentioned IDE's, .NET Core can support its own application model, and also the Universal Windows Platform Model, optionally compiled to .NET Native (see the following screenshot):

WOW! eBook
www.wowebook.org



Source: http://www.hanselman.com/blog/AnUpdateOnASPNETCore10RC2.aspx

We end this introduction to .NET Core with the summary from the same page mentioned previously in relation to this framework:

- **Cross-platform**: .NET Core currently supports three main operating systems--Linux, Windows and OS X. There are other OS ports in progress such as FreeBSD, NetBSD, and Arch Linux. .NET Core libraries can run unmodified across supported OSes. The apps must be re-compiled per environment, given that apps use a native host. Users select the .NET Core supported environment that works best for their situation.
- **Open Source**: .NET Core is available on GitHub at https://github.com/dotnet/core/blob/master/release-notes/2.0/2.0.0-preview1.md, licensed with the MIT and Apache 2 licenses (licensing is per component). It also makes use of a significant set of open source industry dependencies (see release notes). Being OSS is critical for having a thriving community plus a must for many organizations where OSS is part of their development strategy.
- **Natural acquisition**: .NET Core is distributed as a set of NuGet packages that developers can pick and choose from. The runtime and base framework can be acquired from NuGet and OS-specific package managers, such as APT, Homebrew, and Yum. Docker images are available on docker hub. The higher-level framework libraries and the larger .NET library ecosystem are available on NuGet.
- **Modular framework**: .NET Core is built with a modular design, enabling applications to include only the .NET Core libraries and dependencies that are needed. Each application makes its own .NET Core versioning choices, avoiding conflicts with shared components. This approach aligns with the trend of developing software using container technologies such as Docker.
- **Smaller deployment footprint**: Even when in v1.0/1.1 the size of .NET Core is a lot smaller than .NET Framework; note that the overall size of .NET Core doesn't set out to be smaller than the .NET Framework over time, but since it is

WOW! eBook
www.wowebook.org

pay-for-play, most applications that utilize only parts of CoreFX will have a smaller deployment footprint.

- **Fast release cycles of .NET Core**: .NET Core's modular architecture plus its OSS nature provide more modern and much faster release cycles (even per NuGet package) compared to slow release cycles from larger monolithic frameworks. This approach allows a much faster innovation pace from Microsoft and the OSS .NET community than what was traditionally possible with the .NET Framework.

Thus, there are multiple application model stacks built on top of the .NET Core that allow developers to build applications ranging from console applications, across UWP Windows 10 apps (PC, tablet, and phones) to scalable web applications and microservices with ASP.NET Core.

WOW! eBook
www.wowebook.org

ASP.NET Core

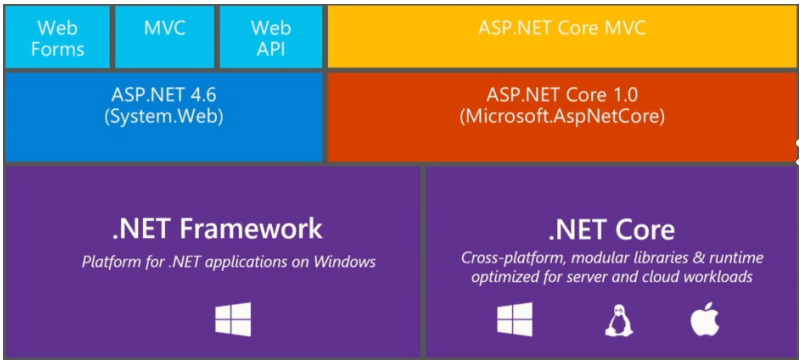
ASP.NET applications that use .NET Core promote a model based on the previous MVC model, although built from scratch, targeted at cross-platform execution, the elimination of some unnecessary features, and the unification of the previous MVC with the web API variant; so, they work with the same controller type.

Besides this, the code doesn't need to be compiled prior to execution while you're developing. The BrowserSync technology allows you change the code on-the-fly and the Roselyn services take care of updating; so, you just have to refresh your page to see the changes.

ASP.NET Core also uses a new hosting model, completely decoupled from the web server environment that hosts the application. It supports IIS versions and also self-hosting contexts via Kestrel (cross-platform, extremely optimized, built on top of LibUv, the same component that Node.js uses) and WebListener HTTP (Windows-only) servers.

As part of its architecture, it proposes a new generation of middleware that is asynchronous, very modular, lightweight, and totally configurable, where we define things such as routing, authentication, static files, diagnostics, error handling, session, CORS, localization; and they can even be user-defined.

Notice also that ASP.NET Core can run as well in the classic .NET Framework with access to the functionality exposed by those libraries. The following screenshot shows the schema:



ASP.NET Core joins many things that were separate in previous versions. Thus,

WOW! eBook
www.wowebook.org

there are no distinctions between MVC and Web API and, if you target .NET Core or if you prefer to target any of the other version of .NET, the architectural model can be MVC using this rebuilt architecture.

In addition, a new built-in IoC container for dependency injection is responsible for bootstrapping the system, together with a new configuration protocol, which we'll see in practice in the following chapters.

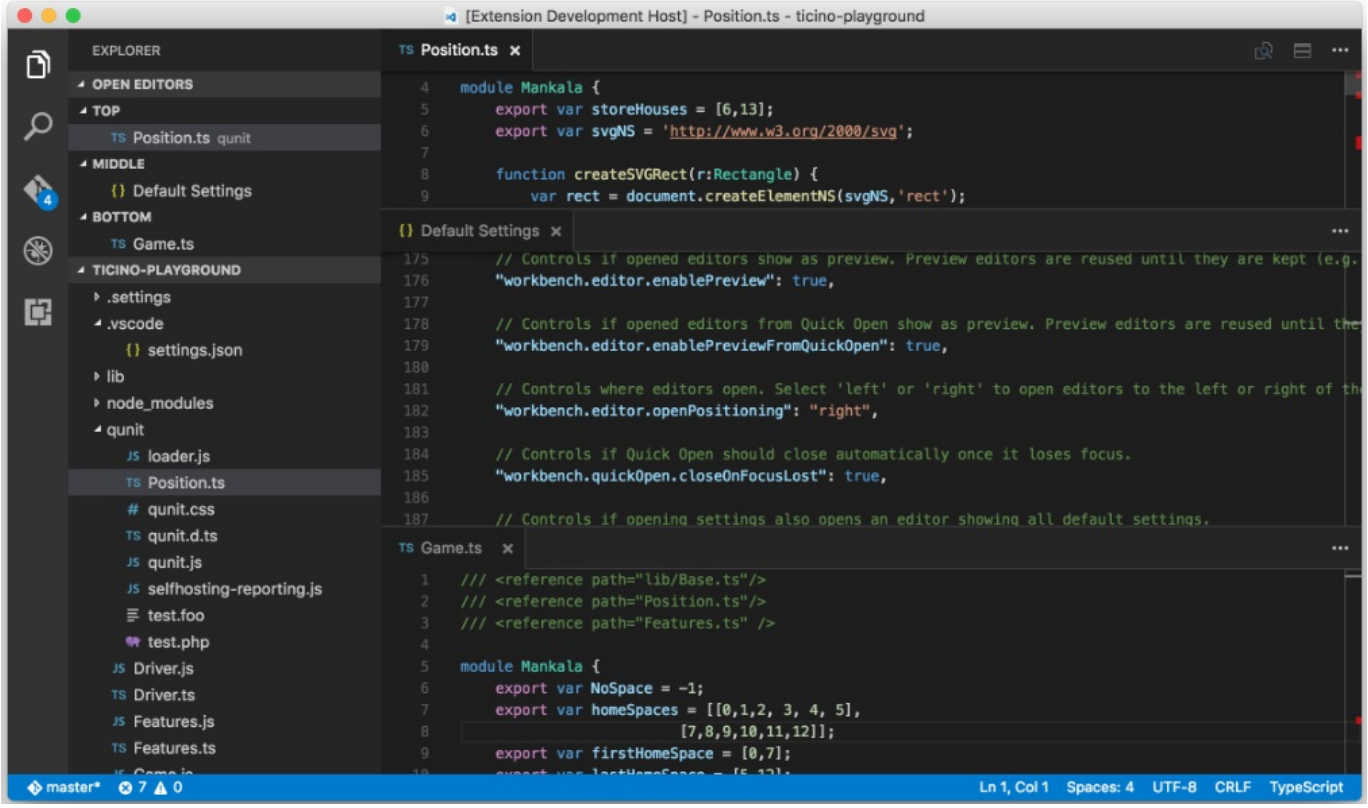
WOW! eBook
www.wowebook.org

About the IDE used in this book

Since this book deals with .NET Core and ASP.NET Core and their built-in capabilities covering SOLID principles in general and DI in particular, we're using the latest available version of Visual Studio (Visual Studio 2017 Enterprise), which includes full support for these platforms, together with a bunch of convenient extensions and templates.

You can also use Visual Studio 2017 Community Edition, which is free, or any higher version with practically no changes, as far as the codes samples are concerned.

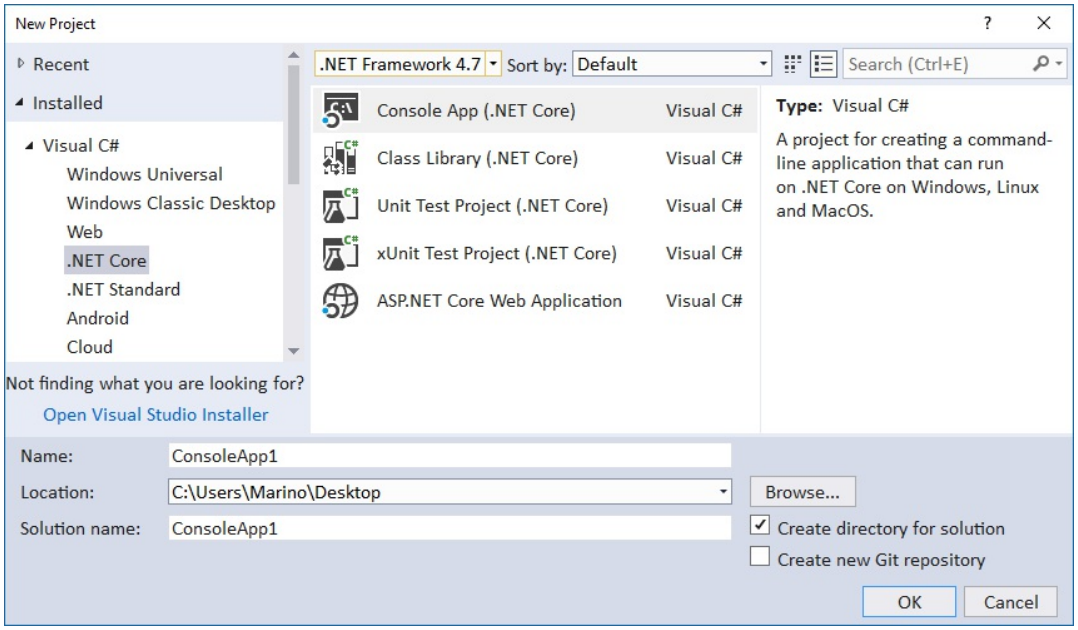
If you're a Mac user, you can also use Visual Studio for Mac (`https://www.visualstudio.com/vs/visual-studio-mac/`), available since November 2016, and, if you prefer a light, full-fledged, and free IDE for any platform (Linux, Mac or Windows), you can opt for Visual Studio Code (`https://code.visualstudio.com/download`), which also has excellent editing and debugging capabilities. All of them have full support for .NET Core/ASP.NET Core as well (see the following screenshot):



WOW! eBook
www.wowebook.org

Throughout this and other chapters, I'll use indiscriminately .NET Core or ASP.NET Core for the demos, depending on whether we need a more complex user interface or not. Notice also that .NET Core (for the time being) does not offer any visual UI beyond Console applications.

Actually, the currently available templates shown by default when we select *New Project* and click on .NET Core are the ones you can see in the following screenshot:



As you see, the choices are basically threefold (besides testing): Console apps, Class libraries, and ASP.NET Core Web apps, based on NET Core. In the three cases, the resulting apps run on any platform.

WOW! eBook
www.wowebook.org

Other foundational changes in .NET Core

It's important to keep in mind that, with NET Core, you no longer depend on .NET Framework libraries (the BCL libraries), either installed by the OS or manually and located in the GAC (Global Assembly Cache).

All libraries are available via NuGet and downloaded accordingly. But, if you have tried .NET Core prior to Visual Studio 2017, you might miss the file `project.json` in which all dependencies were referenced.

The official documentation states that when using Visual Studio 2017:

- MSBuild supports .NET Core projects, using a simplified `csproj` project format that makes it easier to be edited by hand, without the need for unloading the project
- There is support for file wildcards in the project file, enabling folder-based projects that don't require individual files to be included
- NuGet package references are now part of the `csproj` format, consolidating all project references in one file

So, if you try a new .NET Core project with this tool, the project's dependencies are now referenced in the `csproj` file (in XML format), as you can see when opening it in any text editor:

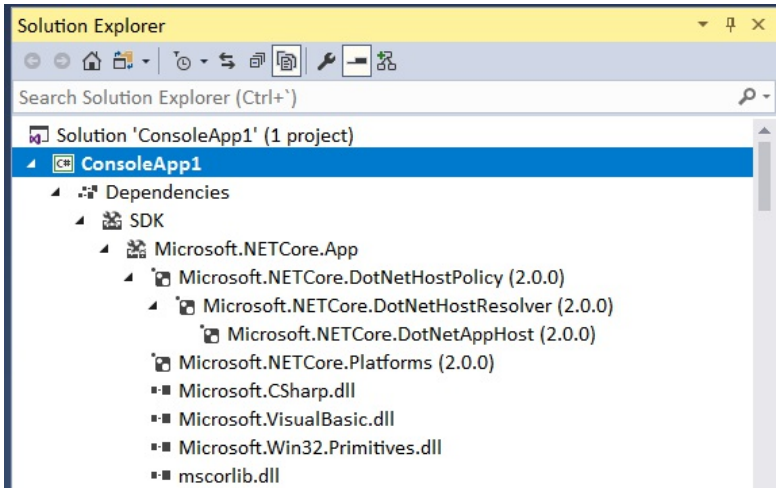
```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

In parallel, Visual Studio reads that file, creates a `Dependencies` entry in the Solution Explorer, and starts looking for that information (either in the PC's cache or in NuGet).

Note also that they're not real, classic DLLs, but fragments of code that are

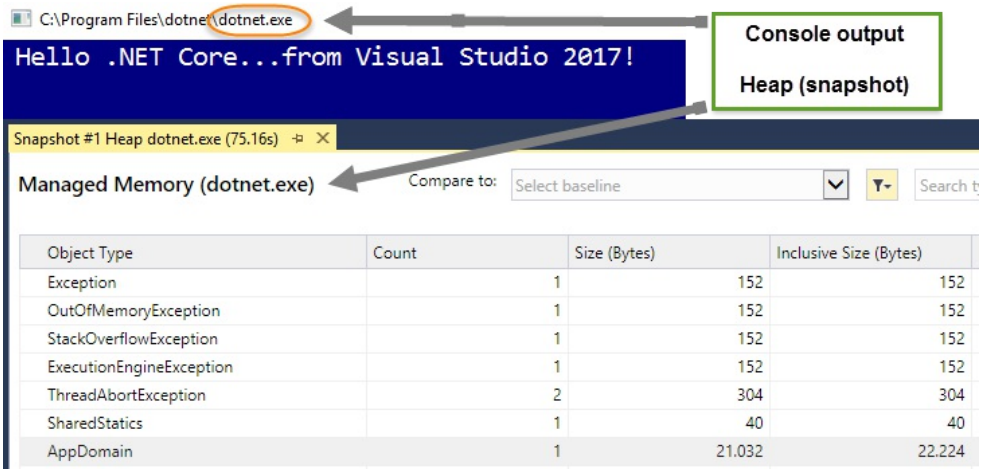
WOW! eBook
www.wowebook.org

assembled all together at compile time to minimize size and launch time. If you take a look at that entry you can see the Dependencies' dependencies, and so on:



Another critical point to highlight relates to the deliverables produced after the compiling process. If you open the demo included as ConsoleApp1 (or create a basic one of your own), and just compile it, you'll see that the bin directory does not contain any executable file. You'll see a DLL with that name instead (ConsoleApp1.dll).

When you launch the application (after adding a `Console.Read()` sentence to stop execution), you'll see that the executable is, indeed, `dotnet.exe`. And the same is true when you open the Diagnostics Tool and take a snapshot of the executable to see what is in place in that moment. The following screenshot shows the situation:



Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)
Exception	1	152	152
OutOfMemoryException	1	152	152
StackOverflowException	1	152	152
ExecutionEngineException	1	152	152
ThreadAbortException	2	304	304
SharedStatics	1	40	40
AppDomain	1	21.032	22.224

The reason for this is directly related to the complexity of this model. The application is thought to execute on distinct platforms. The default option allows the deployment architecture to determine the best way to configure the JIT compilers

WOW! eBook
www.wowebook.org

depending on the target. This is why the execution is undertaken by the dotnet runtime (named dotnet.exe).

From the point of view of deployment, in .NET Core, two types of application are defined: portable and self-contained.

In .NET Core, portable applications are the default. Of course, that means that (as developers) we can be sure about their portability in distinct .NET core installations. However, a standalone app does not depend on any previous installation to run. That is to say, it holds within itself all the necessary components and dependencies, including the runtime packaged with the application. Certainly, that builds a larger app, but it also makes the application capable of executing on any .NET Core platform whether you have .NET Core installed in the target or not.

For the main purposes of this book, it doesn't matter which runtime mode we choose. Anyhow, this brief introduction can give you an idea of how the new framework behaves and is managed inside Visual Studio 2017.

And, remember, anything I do using Visual Studio 2017, you can also do with Visual Studio Code.

WOW! eBook
www.wowebook.org

The SOLID principles

Some programming guidelines have a comprehensive, general-purpose intention, while others are mainly designed to fix certain specific problems. Therefore, before we focus on specific problems, it's important to review those features that can be applied in different scenarios and solutions. I mean those principles that you should consider beyond the type of solution or specific platform to program for.

This is where the SOLID principles (and other related problems) come into play. In 2001, Robert Martin published a foundational article on the subject (`http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`), in which he enumerated a set of principles and guidelines that, in his own words, focus very tightly on dependency management, its potential problems, and how to avoid them.

To explain this further, in his words, *poor dependency management leads to code that is hard to change, fragile, and non-reusable*. So, this principle is directly related with two of the OOP mantras--reusability, and maintainability (the capacity to change as the project grows, one of the main goals of inheritance).

Overall, Martin stated his 11 commandments to consider, but they can be divided into three areas:

- The five SOLID principles, which deal with class design
- The other six principles, mainly focused on packages--three of them are about package cohesion, and the other three explain the dangers of package coupling and how to evaluate a package structure

We're going to start with the SOLID principles, which by extension not only affect the class design, but also other aspects of software architecture.

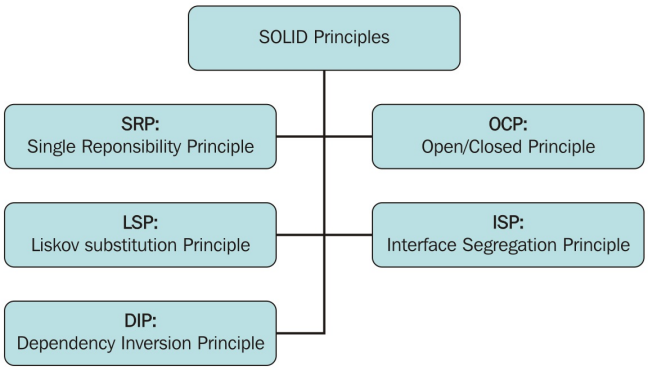
TIP

*The application of these ideas has, for example, been decisive in important modifications made to the HTML5 standard. Concretely, the application of the **SRP (Single Responsibility principle)** only highlighted the need to totally separate presentation (CSS) from content (HTML) and the subsequent deprecation of some tags (`<cite>`, `<small>`, `<font>`).*

WOW! eBook
www.wowebook.org

This applies to other popular frameworks as well, such as AngularJS (and even more in Angular 2), both designed not only with the Single Responsibility principle in mind but also based on the **Dependency Inversion** principle (the **D** in SOLID).

The following diagram schematizes the five principles' initials and correspondences:



The explanation of every letter in the acronym as expressed in Wikipedia is as follows:

- **S - Single Responsibility Principle**: A class should have only a single responsibility (that is, only one potential change in the software's specification should be able to affect the specification of the class). Martin states that this principle is based on the principle of cohesion, previously defined by Tom de Marco in a book named *Structured Analysis and Systems Specification* and by Meilir Page-Jones in his work *The Practical Guide to Structured Systems Design*.
- **O - Open/Closed Principle**: Software entities should be open for extension, but closed for modification. Bertrand Meyer was the first to propose this principle. Martin puts this in another way at `http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`, saying that *You should be able to extend a class's behavior, without modifying it.*
- **L - Liskov Substitution principle**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. Barbara Liskov first stated this, and Martin rephrases the principle in this manner--*Derived classes must be substitutable for their base classes*.
- **I - Interface Segregation principle**: *Many client-specific interfaces are better than one general-purpose interface*. Robert C. Martin was the first to use and formulate this principle, which he rewords in the aforementioned article as--*Make fine grained interfaces that are client specific.*
- **D - Dependency inversion principle**: *We should 'Depend on Abstractions'. Do not depend upon concretions*. This too is an idea developed by Robert C.

WOW! eBook
www.wowebook.org

Martin.

WOW! eBook
www.wowebook.org

The Single Responsibility Principle (SRP)

The **Single Responsibility Principle** (**SRP**), focuses on the fact that there should never be more than one reason for a class to change. In this context, responsibility is defined as a reason for a change. If, under any circumstances, more than one reason comes up to change the class, then the class' responsibilities are multiple and should be redefined.

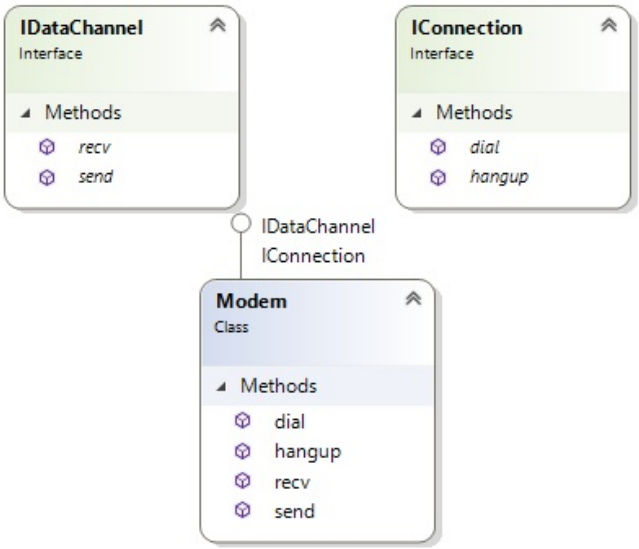
This is, indeed, one of the most difficult principles to apply properly because as Martin says, *conjoining responsibilities is something that we do naturally*. In his book, *Agile Principles, Patterns, and Practices in C#*, Martin proposes a canonical example to show the differences, as follows:

```
interface Modem
{
  public void dial(String phoneNumber);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

Given the previous interface, any class implementing it has two responsibilities: connection management and the communication itself. Such responsibilities can be used from the different parts of an application, which, in turn, might change as well.

We're going to use the Visual Studio 2017 Class Designer to express the way Martin proposes we express this class design instead:

WOW! eBook
www.wowebook.org



As we see, in Martin's solution, the class depends on two interfaces, each one in charge of a responsibility--connection and channel transmission (two abstractions, really: remember that an interface is not compiled and it only serves as a contract for the compiler to check).

However, one wonders, should these two responsibilities be separated? It only depends on application changes. To be precise, the key here is to know whether changes in the application affect the signature of connection functions. If they do, we should separate both; otherwise, there's no need for separation because we would then create needless complexity.

So, overall, a reason to change is the key, but keep in mind that a reason to change is applicable only if changes occur.

In other situations, there might be reasons to keep distinct responsibilities together as long as they are closely related to the business definitions or have to do with the hardware requirements of the operating system.

WOW! eBook
www.wowebook.org

The background of the Separation of Concerns (SoC)

As always happens, there were previous approaches to the problem of software separation. Dijkstra in "*On the role of scientific thought*" (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html) mentioned that *It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.*

Another advance was Information Hiding, defined by Wikipedia (https://en.wikipedia.org/wiki/Information_hiding) as *the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.* This was the seed that later became a basic pillar of OOP--Data Encapsulation.

Even Barbara Liskov, whom we mentioned in connection with the substitution principle, published at the same time *Programming With Abstract Data Types* (http://dl.acm.org/citation.cfm?id=807045), which she describes as an approach to the computer representation of abstraction. The definition of ADTs as a class of objects whose logical behavior is defined by a set of values and a set of operations links data and functionality.

Later approaches have improved these ideas. Proposals for Code Contracts, originally introduced by Bertrand Meyer in his Eiffel language, and implemented in C# via Code Contracts (https://msdn.microsoft.com/es-es/library/dd264808(v=vs.110).aspx) foster the use of pre and post conditions that our software has to accomplish.

Finally, we can think of the separation of what Hayim Makabee (https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/) reports as cross-cutting concerns--aspects that might affect distinct pieces of software in even distinct layers of the application and that should be managed in a similar fashion (authorization or instrumentation issues, and so on.). In .Net, we count on Attributes, applicable equally to classes and class members, to modify and tune such behavior.

A bit later in the same article, Makabee clearly establishes the main purposes for these techniques. If we understand coupling as the degree of dependency between

WOW! eBook
www.wowebook.org

two modules, the goal is to obtain low coupling. Another term is cohesion or the measure of how strongly-related the set of functions performed by a module is. Obviously, high cohesion is better.

He ends by summarizing the benefits obtained with these techniques:

Patterns and methodologies are always intended to reduce coupling and at the same time increase congruity. By hiding information, we reduce coupling since we isolate implementation details. Thus, ADT's reduce coupling by using clear and abstract interfaces. We have an ADT specifying the set of function that can be executed on a type, that's more cohesive than a global data structure modified by external functions. The way that OOP reaches that cohesion is the implementation of two of its basic principles--encapsulation and polymorphism, together with dynamic binding. Furthermore, inheritance reinforces cohesion by means of hierarchies that are based on generalization and specialization, which permits a suitable separation from the functionality belonging to a superclass from its subclasses. AOP, in turn, supplies solutions for cross-cutting concerns in a way that both aspects and functionality may become more cohesive.

Maintainability, reusability, and extensibility are only three of the main advantages gained with its implementation.

WOW! eBook
www.wowebook.org

Well-known examples of Separation of Concerns

All of us have gone through cases and scenarios where the separation of concerns lies at the heart of the system or technology that implements it. One such case is HTML (and, especially HTML5).

Since its inception, the standard HTML5 was thought to clearly separate content from presentation. And the popularity of mobile devices only made that requirement more evident. The huge variety of form factors available today demanded a technology capable of adapting to these sizes, in such a way that content could be held by HTML tags and the final presentation in a given device decided at runtime depending on the device.

Therefore, some tags were declared deprecated, such as `<font>`, `<big>`, `<center>`, and a list of others, and the same happened to some attributes, such as `background`, `align`, `bgcolor`, or `border` since they didn't make sense in this new system. Even some of them that still remain unchanged and that have a visual effect on the output (such as `<b>`, `<i>`, or `<small>`) are kept for their semantic meaning and not for their presentational effects, which is a role totally dependent on CSS3.

So, one of the main goals is to avoid functionality overlapping, although this is not the only benefit. If we understand concerns as the different aspects of software functionality, the business logic of software is a concern, and the interface through which a person uses this logic is another.

In practice, this translates into keeping the code for each of these concerns separate. That means, changing the interface should not require changing the business logic code, and vice versa. The underlying principle of encapsulation reinforces these ideas in the OOP paradigm, and the **Model-view-controller** (**MVC**) design pattern is a great example of separating these concerns for better software maintainability.

WOW! eBook
www.wowebook.org

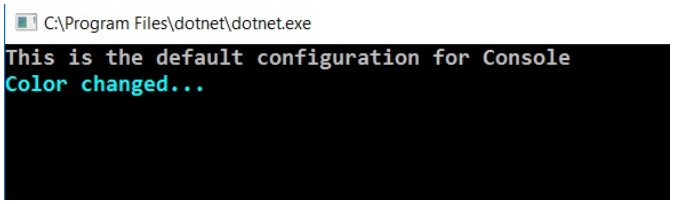
A basic sample of Separation of Concerns

Let's put this into code with a very basic sample and check for the differences between coupled and decoupled implementations. Imagine that a Console application in .NET Core has to show the user the initial configuration of Console colors, change a value, and present those changes.

If you make a basic project `ConsoleApp1`, the following code could be the first approach:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    Console.ResetColor();
    Console.WriteLine("This is the default configuration for Console");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("Color changed...");
    Console.Read();
  }
}
```

This produces the expected results (see the following screenshot showing the output):



Which problems can we find in this code? First, the main point is in charge of everything: resets the previous initial configuration of the console, changes the foreground, and prints the results.

The first attempt at separation would be to realize that further needs might require other fragments of code to use the same functionality. Even more, that functionality would be better located in another difference-- a piece of software such as a library, for example. So, we should enhance our solution with a new project containing a

WOW! eBook
www.wowebook.org

library project that would be referenced by any other projects along with the solution.

Besides, the manual change to Cyan color implicitly reminds us of the need for a function that allows changing to any valid color.

So, we might end up with another piece of code like this:

```
namespace Utilities
{
  public class ConsoleService
  {
    public void ChangeForegroundColor(ConsoleColor newColor)
    {
        Console.ForegroundColor = newColor;
    }
    public void ResetConsoleValues()
    {
        Console.ResetColor();
    }
  }
}
```

Now, in the main entry point, we could write:

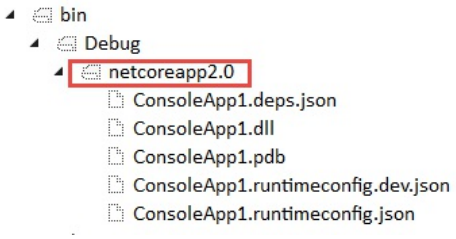
```
/* This is version 2 (with utilities) */
ConsoleService cs = new ConsoleService();
cs.ResetConsoleValues();
Console.WriteLine("This is the default configuration for Console");
cs.ChangeForegroundColor(ConsoleColor.Cyan);
Console.WriteLine("Color changed...");
Console.Read();
```

With exactly the same results (I omit the output since there are no changes). So, we made a physical separation together with a logical one, given that now any change to the `Console`, should be managed by the `Utilities` library, which increases their reusability and therefore maintainability and testing.

Notice also that we could have opted for creating the library as static, to avoid instantiation.

The only change over previous versions of .NET is that, as we showed in a previous screenshot, the reference to the library is now made slightly differently, as it appears in the Dependencies section on the Solution Explorer. Once the project is compiled, we can also see that reference in the `bin` directory resulting from compilation:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

Another sample

Let's take a more everyday approach with another sample: something simple, such as reading from a JSON file on a disk and presenting the results in the output. So, I've created a .NET Core Console app that includes a JSON file with five books from PACKT.

A first approach could be the following code:

```
using System;
using System.IO;
using Newtonsoft.Json;

class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine(" List of Books by PACKT");
    Console.WriteLine(" -----------------------");
    var cadJSON = ReadFile("Data/BookStore.json");
    var bookList = JsonConvert.DeserializeObject<Book[]>(cadJSON);
    foreach (var item in bookList)
    {
        Console.WriteLine($" {item.Title.PadRight(39,' ')} " +
            $"{item.Author.PadRight(15,' ')} {item.Price}");
    }
    Console.Read();
  }

  static string ReadFile(string filename)
  {
    return File.ReadAllText(filename);
  }
}
```

As we can see, the code uses a `Book` class that implements the `IBook` interface, defined in a very simple manner:

```
interface IBook
{
  string Title { get; set; }
  string Author { get; set; }
  double Price { get; set; }
}
class Book : IBook
{
  public string Author { get; set; }
  public double Price { get; set; }
  public string Title { get; set; }
}
```

WOW! eBook
www.wowebook.org

This works fine, and generates the following output:

```
C:\Program Files\dotnet\dotnet.exe
List of Books by PACKT
----------------------
Mastering C# and .NET Programming       Marino Posadas  45,99
F# 4.0 Programming Cookbook             Kevin Ashton    31,71
Mastering .NET Machine Learning         Jamie Dixon     54,99
Learning Node.js for .NET Developers    Harry Cummings  29,99
ASP.NET Core and Angular 2              V. De Sanctis   39,92
```

Notice that we're using the popular Newtonsoft JSON library, to easily convert the string into an array of `Book` objects.

If we analyze the code, we can identify several places where that SoC principle is present:

- First, since the entity to manage is a `Book` (which has three properties), I created a `Model` folder to hold the definition of a `Book` interface (`IBook`), and also a `Book` class that implements that interface
- Secondly, the use of the `Newtonsoft` library is another separation since it's the library that takes care of the conversion of the string into an array of `Books`
- Finally, file reading takes place in the method `ReadFile()`, which receives the name of the file

Is there any other separation required? As we mentioned, the reason to change would be key at the time to decide. For example, does the app read another type of information (apart from `Books`)? Or, does our UI really need to include the `ReadFile()` method? And what about having to reference directly the `Newtonsoft` library directly in the user interface?

If this isn't the case, perhaps a better approach would be to separate that method in a `Utilities` class, just like in the first sample, thus ensuring architecture has three separate folders to hold different aspects of the application: the data model, the utilities area, and the main user interface.

In this manner, we would end up with a `Utilities` class like this:

```
using Newtonsoft.Json;
using System.IO;

internal class Utilities
{
  internal static Book[] ReadData()
```

WOW! eBook
www.wowebook.org

```
    {
      var cadJSON = ReadFile("Data/BookStore.json");
      return JsonConvert.DeserializeObject<Book[]>(cadJSON);
    }

    static string ReadFile(string filename)
    {
      return File.ReadAllText(filename);
    }
  }
```

And the resulting `Program` class gets reduced to the following:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    var bookList = Utilities.ReadData();
    PrintBooks(bookList);
  }

  static void PrintBooks(Book[] books)
  {
    Console.WriteLine(" List of Books by PACKT");
    Console.WriteLine(" ----------------------");
    foreach (var item in books)
    {
      Console.WriteLine($" {item.Title.PadRight(39, ' ')} "  +
         $"{item.Author.PadRight(15, ' ')} {item.Price}");
    }
    Console.Read();
  }
}
```

Of course, we get the same output, but now we have an initial separation of concerns. There's no need to reference external libraries in the UI, which facilitates maintainability and extensibility.

Let's now explore the second principle: Open/Closed.

WOW! eBook
www.wowebook.org

The Open/Closed principle

We can detect the need to use this principle when a change in the module results in a cascade of changes that affect dependent modules. The design is said to be too inflexible.

The **Open/Closed principle** (**OCP**) principle advises us that we should refactor the application in such a manner that future changes don't provoke further modifications.

The form to apply this principle correctly would be by extending the functionality with new code (for instance, using polymorphism) and never changing the old code, which is working already. We can find several strategies to achieve this goal.

Observe that closed for modification is especially meaningful when you have distinct, separate modules (DLLs, EXEs, and so on) that depend on the module that has to be changed.

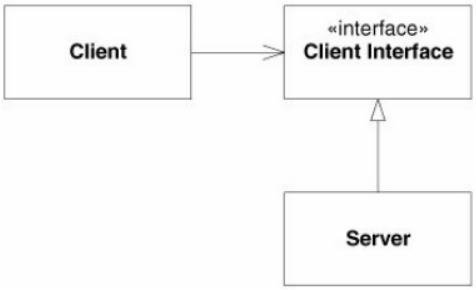
On the other hand, using extension methods or polymorphic techniques allows us to perform changes in code without affecting the rest. Think, for example, about the extension methods available in the C# language since version 3.0.

You can consider extension methods as a special type of static methods with the difference being that they are called as if they were instance methods of the extended type. You find a typical example in the LINQ standard query operators because they add a query functionality to the existing types, such as `System.Collections.IEnumerable` or `System.Collections.Generic.IEnumerable<T>`.

The classical and simplest example of this pattern is the client/server cohesion that has been largely seen in development for many years. It is preferable that clients depend on server abstractions, not on their concretions.

This can be achieved with interfaces. Servers can implement a client interface that clients will use to connect to them. In this manner, servers can change without affecting the way clients use them (refer to the following diagram):

WOW! eBook
www.wowebook.org



Any subtype of the client interface will be free to implement the interface in the way it deems more appropriate, and as long as it doesn't break other clients' access.

WOW! eBook
www.wowebook.org

Back to our sample

Let's imagine a simple case in which the app has to cover a new aspect. For example, the app now has to allow the user to list an extra file of books to be added to the previous list.

For this new requirement, we can create a new and overloaded `ReadData()` method that receives an extra argument. Notice here that the argument doesn't even have to be used. It's enough if it declares another signature to be invoked for this extra situation.

If we have the extra data in another file (`BookStore2.json`, in our demo), we could create this extra version of the method:

```
internal static List<Book> ReadData(string extra)
{
  List<Book> books = ReadData();
  var cadJSON = ReadFile("Data/BookStore2.json");
  books.AddRange(JsonConvert.DeserializeObject<List<Book>>(cadJSON));
  return books;
}
```

Notice that we don't even use the method's argument in this implementation (of course, there are other ways to do this, but let's put it this way for the purpose of the demo).

We have now two versions of `ReadData()` that should be called in the user interface depending on the user's choice (I also changed the `Book[]` definition into a `List<Book>` for simplicity, but you can see the older version as well in the source code):

```
static List<Book> bookList;
static void Main(string[] args)
{
  Console.WriteLine("Please, press 'yes' to read an extra file, ");
  Console.WriteLine("or any other key for a single file");
  var ans = Console.ReadLine();
  bookList = (ans != "yes") ? Utilities.ReadData() : Utilities.ReadData(ans);
  PrintBooks(bookList);
}
```

Now if the user's answer is *yes* you have an extra set of books added to the list, as you can see in the output:

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
Please, press 'yes' to read an extra file,
or any other key for a single file
yes
 List of Books by PACKT
 ----------------------
 Mastering C# and .NET Programming       Marino Posadas       45,99
 F# 4.0 Programming Cookbook             Kevin Ashton         31,71
 Mastering .NET Machine Learning         Jamie Dixon          54,99
 Learning Node.js for .NET Developers    Harry Cummings       29,99
 ASP.NET Core and Angular 2              V. De Sanctis        39,92
 ASP.NET Core 1.0 High Performance       James Singleton      49,99
 ASP.NET Core Essentials                 Shahed Chowdhuri     34,99
 ASP.NET Site Performance Secrets        Matt Perdeck         59,99
 ASP.NET Web API Security Essentials     Rajesh Gunasundaram  34,99
 Mastering ASP.NET Core                  Ricardo Peres        30,99
```

Besides all these reasons, you can think of situations such as having the `Utilities` code separated in a distinct library that could also be used by other parts of the application. The implementation of the Open/Closed principle here allows a more stable and extensible approach.

WOW! eBook
www.wowebook.org

The Liskov Substitution principle

Let's remember this definition--subtypes must be substitutable for their base types. This means that this should happen without breaking the execution or losing any other kind of functionality.

You'll notice that this idea lies behind the basic principles of inheritance in the OOP programming paradigm.

If you have a method that requires an argument of the `Person` type (let's put it that way), you can pass an instance of another class (`Employee`, `Provider`, and so on) as long as these instances inherit from `Person`.

This is one of the main advantages of well-designed OOP languages, and most popular and accepted languages support this characteristic.

WOW! eBook
www.wowebook.org

Back to the code again

Let's take a look at the support inside our sample, where a new requirement arises. Actually, our demo simply calls the `PrintBooks` method and expects to receive a `List<Book>` object as the argument.

However, another reason for change might come up when new lists of books appear, and those lists include some new field, like the topic each book belongs to (.NET, Node, Angular, and so on).

For example, a new list appears containing a fourth field, `Topic`, in this way:

```
{
  "Title": "AngularJS Services",
  "Author": "Jim Lavin",
  "Price": 30.99,
  "Topic": "Angular"
}
```

The class `Book` should not be changed since it's being used already. Thus, we can inherit from `Book` and create a `TopicBook` class just adding the new field (I'm trying to keep things as simple as possible to focus on the architecture we're dealing with):

```
public class TopicBook : Book
{
  public string Topic { get; set; }
}
```

To cover this new aspect, we can change the user interface to allow the user to select a new option (`topic`) that includes the new type of book:

```
static void Main(string[] args)
{
  Console.WriteLine("Please, press 'yes' to read an extra file, ");
  Console.WriteLine("'topic' to include topic books or any
      other key for a single file");
  var ans = Console.ReadLine();
  bookList = ((ans != "yes") && (ans != "topic")) ?
    Utilities.ReadData() : Utilities.ReadData(ans);
  PrintBooks(bookList);
}
```

Notice that we're just including a new condition and calling the overloaded method in case the new condition is selected.

WOW! eBook
www.wowebook.org

As for the `ReadData()` overloaded method, we can make some minimal changes (basically, adding an `if` condition to include the extra data), like you can see in the following code:

```
internal static List<Book> ReadData(string extra)
{
  List<Book> books = ReadData();
  var filename = "Data/BookStore2.json";
  var cadJSON = ReadFile(filename);
     books.AddRange(JsonConvert.DeserializeObject<List<Book>>(cadJSON));
  if (extra == "topic")
  {
     filename = "Data/BookStore3.json";
     cadJSON = ReadFile(filename);
     books.AddRange(JsonConvert.DeserializeObject<List<TopicBook>>(cadJSON));
  }
  return books;
}
```

Observe that the method's changes are minimal, and especially that we're adding to the list of books the result of deserializing a different class (`TopicBook`), without any compilation or execution problems.

Therefore, the implementation of Generics in .NET (and .NET Core, in this case) correctly implements the Liskov Substitution Principle, and we don't have to make modifications in our logic.

We can check the results in the Automatic Window using a breakpoint before the `return` sentence of `ReadData` and seeing how the `List<Book>` now includes five elements of type `TopicBook`, with no complaints:

WOW! eBook
www.wowebook.org

books	Count = 15	System.Collections.Generic.List<Book>
[0]	{Book}	Book
[1]	{Book}	Book
[2]	{Book}	Book
[3]	{Book}	Book
[4]	{Book}	Book
[5]	{Book}	Book
[6]	{Book}	Book
[7]	{Book}	Book
[8]	{Book}	Book
[9]	{Book}	Book
[10]	{TopicBook}	Book {TopicBook}
Author	"Alex Vantson"	string
Price	42.99	double
Title	"AngularJS Directives"	string
Topic	"Angular"	string
[11]	{TopicBook}	Book {TopicBook}
[12]	{TopicBook}	Book {TopicBook}
[13]	{TopicBook}	Book {TopicBook}
[14]	{TopicBook}	Book {TopicBook}

What about the other side (the user interface logic) and, especially, our `PrintBooks` method, which expects a `List<Book>`? Well, there's no difference insofar as we don't try to print out a field that doesn't exist.

You can check the output in the following screenshot:

```
C:\Program Files\dotnet\dotnet.exe
Please, press 'yes' to read an extra file,
'topic' to include topic books or any other key for a single file
topic
 List of Books by PACKT
 ----------------------
 Mastering C# and .NET Programming       Marino Posadas       45,99
 F# 4.0 Programming Cookbook              Kevin Ashton         31,71
 Mastering .NET Machine Learning          Jamie Dixon          54,99
 Learning Node.js for .NET Developers     Harry Cummings       29,99
 ASP.NET Core and Angular 2               V. De Sanctis        39,92
 ASP.NET Core 1.0 High Performance        James Singleton      49,99
 ASP.NET Core Essentials                  Shahed Chowdhuri     34,99
 ASP.NET Site Performance Secrets         Matt Perdeck         59,99
 ASP.NET Web API Security Essentials      Rajesh Gunasundaram  34,99
 Mastering ASP.NET Core                   Ricardo Peres        30,99
 AngularJS Directives                     Alex Vantson         42,99
 Learning AngularJS Animations            Richard Keller       37,99
 AngularJS Directives Cookbook            Fernando Monteiro    39,5
 AngularJS Essentials                     Rodrigo Branas       34,99
 AngularJS Services                       Jim Lavin            30,99
```

Thanks to the Liskov Substitution principle support, we were able to add behavior and information with minimum effort, and consequently, enforce the OOP principle of code reutilization.

WOW! eBook
www.wowebook.org

Other implementations of LSP in .NET

What we've seen up to this point is not the only implementation of the LSP principle that we find inside .NET, since different areas of the framework, have grown using this conception.

Events are flexible enough to be defined in a way that allows us to pass our own information via classic definitions, alternatively with the participation of generics, we can simply define a generic event handler that holds information of any kind. All these techniques foster the implementation of good practices, not just the SOLID principles.

WOW! eBook
www.wowebook.org

The Interface Segregation principle

As Martin states, this principle deals with the inconveniences of *fat* interfaces. And the problem arises when the interfaces of the class can be logically fragmented into distinct groups or methods.

In this case, if there is more than one client of our application, chances are that some clients are connected to a functionality they never use. As Martin states in his *Agile Principles Patterns and Practices in C#* book,

When the clients are separate, the interfaces should remain separate, too. Why? Because clients exert forces on their server interfaces. When we think of forces that cause changes in software, we normally think about how changes to interfaces will affect their users.

And, as a conclusion, he remarks that

When clients are forced to depend on methods they don't use, those clients are subject to changes to those methods. This results in an inadvertent coupling between all the clients. Said another way, when a client depends on a class that contains methods that the client does not use but that other clients do use, that client will be affected by the changes that those other clients force on the class. We would like to avoid such couplings where possible, and so we want to separate the interfaces.

WOW! eBook
www.wowebook.org

Another sample

Let's see this situation with another example that starts from a new scenario. Let's imagine another app in which we have to cover not only the two types of books available at the moment but also a new publication in video format, that holds another field named `Duration`.

A single record of this file would look like this:

```
{
  "Title": "HTML 5 Game Development",
  "Author": "Daniel Albu",
  "Price": 5.68,
  "Topic": "HTML5 Games",
  "Duration": "2h20m"
}
```

But the application maintains the other two previous formats, so we have the possibility to list files with three, four, or five fields, depending on the initial selection the user chooses.

A first approach could lead us to an interface like this:

```
interface IProduct
{
  string Title {get; set;}
  string Author {get; set;}
  double Price {get; set;}
  string Topic { get; set; }
  string Duration { get; set; }
}
```

Based on this interface we could create the `Product` class (the new name is supposed to locate a step above the books or videos since both have four fields in common):

```
public class Product : IProduct
{
  public string Title { get; set; }
  public string Author { get; set; }
  public double Price { get; set; }
  public string Topic { get; set; }
  public string Duration { get; set; }
}
```

Now, the equivalent `Utilities` class could select a file depending on the user's entry, read it, deserialize it, and send the information back to a `PrintProducts` method in

WOW! eBook
www.wowebook.org

charge of the console output.

Our new user interface would look like this:

```
using System;
using System.Collections.Generic;

class Program
{
  static List<Product> productList;
  static void Main(string[] args)
  {
    string id = string.Empty;
    do
    {
        Console.WriteLine("File no. to read: 1/2/3-Enter(exit): ");
        id = Console.ReadLine();
        if ("123".Contains(id) && !String.IsNullOrEmpty(id))
        {
            productList = Utilities.ReadData(id);
            PrintBooks(productList);
        }
    } while (!String.IsNullOrWhiteSpace(id));
  }

  static void PrintBooks(List<Product> products)
  {
    Console.WriteLine(" List of Products by PACKT");
    Console.WriteLine(" ----------------------");
    foreach (var item in products)
    {
        Console.WriteLine($" {item.Title.PadRight(36, ' ')} " +
        $"{item.Author.PadRight(20, ' ')} {item.Price}" + " " +
        $"{item.Topic?.PadRight(12, ' ') } " +]
        $"{item.Duration ?? ""}");
    }
    Console.WriteLine();
  }
}
```

Observe that we had to deal with the two cases in which some field could be null, so we use string interpolation, together with the null coalescence operator (`??`) and the Null-conditional operator (`?`), to prevent failure in these cases.

The `Utilities` class gets reduced to a much simpler code:

```
using System.Collections.Generic;
using System.IO;

internal class Utilities
{
  internal static List<Product> ReadData(string fileId)
  {
    var filename = "Data/ProductStore" + fileId + ".json";
    var cadJSON = ReadFile(filename);
    return JsonConvert.DeserializeObject<List<Product>>(cadJSON);
```

WOW! eBook
www.wowebook.org

```
    }
    static string ReadFile(string filename)
    {
      return File.ReadAllText(filename);
    }
  }
```

The output lets the user select a number and print the file's content in a similar way to what we did in previous demos, only this time selecting each file individually:

```
C:\Program Files\dotnet\dotnet.exe
File no. to read: 1/2/3-Enter(exit):
1
 List of Products by PACKT
 ----------------------
 Mastering C# and .NET Programming    Marino Posadas       45,99
 F# 4.0 Programming Cookbook          Kevin Ashton         31,71
 Mastering .NET Machine Learning      Jamie Dixon          54,99
 Learning Node.js for .NET Developers Harry Cummings       29,99
 ASP.NET Core and Angular 2           V. De Sanctis        39,92

File no. to read: 1/2/3-Enter(exit):
2
 List of Products by PACKT
 ----------------------
 AngularJS Directives                 Alex Vantson         42,99 Angular
 Learning AngularJS Animations        Richard Keller       37,99 Angular
 AngularJS Directives Cookbook        Fernando Monteiro    39,55 Angular
 AngularJS Essentials                 Rodrigo Branas       34,99 Angular
 AngularJS Services                   Jim Lavin            30,99 Angular

File no. to read: 1/2/3-Enter(exit):
3
 List of Products by PACKT
 ----------------------
 HTML 5 Game Development              Daniel Albu          5,68 HTML5 Games  2h20m
 Advanced JavaScript Development      Marco Covarrubias    5,68 JavaScript   3h11m
 Mastering CSS3 Selectors             Ben Fhala            5,68 CSS3         1h44m
 Mastering HTML5 Canvas               Ben Fhala            5,68 HTML5        3h23m
 Mastering TypeScript                 David Sherret        5,68 TypeScript   3h13m
```

If our application now requires more changes, like the addition of statistics, for example, the use of a single class to hold them all (the `Product` class, here) denotes a violation of the Interface Segregation principle.

This is because we should separate the interfaces and use a compound approach to prevent a class from dealing with unwanted or unneeded functionality.

The alternative and proper separation could be to create the following (distinct) interfaces:

```
interface IProduct
{
  string Title { get; set; }
  string Author { get; set; }
  double Price { get; set; }
}
```

WOW! eBook
www.wowebook.org

```
interface ITopic
{
  string Topic { get; set; }
}

interface IDuration
{
  string Duration { get; set; }
}
```

Now we should have three classes, since three entities can be distinguished, but could maintain three fields in common. The definitions of the three classes could be expressed in this way:

```
class Book : IProduct
{
  public string Author { get; set; }
  public double Price { get; set; }
  public string Title { get; set; }
}
class TopicBook: IProduct, ITopic
{
  public string Author { get; set; }
  public double Price { get; set; }
  public string Title { get; set; }
  public string Topic { get; set; }
}
class Video: IProduct, ITopic, IDuration
{
  public string Author { get; set; }
  public double Price { get; set; }
  public string Title { get; set; }
  public string Topic { get; set; }
  public string Duration { get; set; }
}
```

Thanks to this division every entity keeps its own personality, and we can later create methods that use generics, or apply the Liskov Substitution principle to deal with the distinct requirements that might arise during the lifecycle.

WOW! eBook
www.wowebook.org

The Dependency Inversion principle

The last of the SOLID principles is based on two statements, which Wikipedia (https://en.wikipedia.org/wiki/Dependency_inversion_principle) defines in this form:

- *High-level modules should not depend on low-level modules. Both should depend on abstractions*
- *Abstractions should not depend upon details. Details should depend upon abstractions*

As for the first statement, we should clarify what we understand by high-level and low-level modules. The terminology is related to the importance of the actions performed by the module with respect to the application as a whole.

Let's put it simply: if a module holds the business logic of a `Customers` class, and another module `PrinterService` includes the format that a list of the `Customers` class uses in a report, the first one would be high-class and the second would be low-class (the reason for the existence of the second is to provide some functionality to the first).

The second statement speaks for itself. If an abstraction depends on details, the usage as a definition contract is compromised (a change in the details could force a redefinition).

WOW! eBook
www.wowebook.org

The (more or less) canonical example

Dependency Injection techniques are just a way of implementing this principle, and we will see them exemplified in many forms and scenarios along this book.

So, I'll use here the (almost) canonical code that you could find on the internet about this subject. I'm showing you here an adaptation made by Munir Hassan (https://www.codeproject.com/Articles/495019/Dependency-Inversion-Principle-and-the-Dependency) in CodeProject which uses a notification scenario to illustrate this situation, and I think it's particularly interesting. He starts with an initial code such as this:

```
public class Email
{
  public void SendEmail()
  {
    // code
  }
}
public class Notification
{
  private Email _email;
  public Notification()
  {
    _email = new Email();
  }
  public void PromotionalNotification()
  {
    _email.SendEmail();
  }
}
```

Notification depends on `Email`, creating an instance in its constructor. This kind of interaction is said to be tightly coupled. If we want to send other types of notification as well, we have to modify the way the `Notification` class is implemented.

A way to achieve this could be the introduction of an interface (a new level of abstraction) to define the concept of sending messages and force the `Email` class to implement that interface:

```
public interface IMessageService
{
  void SendMessage();
}
public class Email : IMessageService
```

WOW! eBook
www.wowebook.org

```
{
  public void SendMessage()
  {
    // code
  }
}
public class Notification
{
  private IMessageService _iMessageService;
  public Notification()
  {
    _iMessageService = new Email();
  }
  public void PromotionalNotification()
  {
    _iMessageService.SendMessage();
  }
}
```

Now, the class calls something named `_iMessageService`, whose implementation could vary. As Hamir mentions, there are three ways to implement this pattern:

DI is the act of supplying all classes that a service needs rather than leaving the responsibility to the service to obtain dependent classes. DI typically comes in three flavors: Constructor Injection, Property Injection, Method Injection

In the first form, (constructor injection) Hamir proposes the following:

```
public class Notification
{
  private IMessageService _iMessageService;
  public Notification(IMessageService _messageService)
  {
    this._iMessageService = _messageService;
  }
  public void PromotionalNotification()
  {
    _iMessageService.SendMessage();
  }
}
```

This reminds us of what we will see in the implementation of Dependency Injection in ASP.NET Core in the following chapters. No mention of `Emails` here: only an `IMessageService` is implied.

You can visit the aforementioned page for more details about the other ways to implement injection, but, as I mentioned, we'll cover all those in detail in the coming chapters.

WOW! eBook
www.wowebook.org

Other ways to implement Dependency Inversion

Generally speaking, there are many ways in which the DIP principle can lead to a solution. Another way to implement this principle is by using the Dependency Injection techniques, derived from another way to see Dependency Inversion: the so-called **Inversion of Control** (**IoC**).

According to the paper written by Martin Fowler (https://martinfowler.com/articles/injection.html), Inversion of Control is the principle whereby the control flow of a program is inverted; instead of the programmer controlling the flow of a program, the external sources (framework, services, and other components) take control of it.

One of them is a dependency container, which is a component and serves or provides you with some code, injecting it when required.

Some popular Dependency Containers for C# are Unity and Ninject, to name just a couple. In NET Core, there's an embedded container so there's no need to use an external one, except in cases where we might require some special functionality provided by them.

In the code, you instruct this component to register certain classes of your application; so, later on, when you need an instance of one of them, you just have to declare it (typically in the constructor), and it is served to your code automatically.

Other frameworks implement this principle as well, even if they're not purely object-oriented. This is the case with AngularJS or Angular 2, in which, when you create a controller that requires access to a service, you ask for the service in the controller's function declaration, and the internal Angular's DI system serves a singleton instance of the service without the intervention of the client's code.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we've reviewed the five SOLID Principles in the way they were formulated by Robert C. Martin in 2000.

We've explored each of these principles, discussing their advantages and checking their implementation with some simple code using .NET Core Console applications, to see how they can be coded.

In the next chapter, we will talk about Dependency Injection and the most popular IoC containers, reviewing how they can be used and analyzing their pros and cons in everyday applications.

WOW! eBook
www.wowebook.org

Dependency Injection and IoC Containers

This chapter is intended to cover the Dependency Inversion principle in more depth. This means coverage of how it is used in already popular frameworks with different languages, like C#, or JavaScript. We will see what are its main advantages and caveats are, together with a brief analysis of why it is gaining momentum and consensus among developers all over the world.

In this chapter, we'll cover the following topics:

- In all, we'll talk about the concepts and implementation of software artifacts like Factories and Service Locators, and how they relate to Dependency Injection. Factories and Service Locators--concepts, implementations, and how they relate to Dependency Injection
- We'll also cover how IoC containers implement this concept, and what the main points that these IoC containers have in common are, beyond the language they're implemented in
- Then, we'll go through a brief introduction to Object Composition, Object Lifetime, and the different types of injection, ending with some commentaries about those cases in which Dependency Injection is not the best choice
- We will end up with some demos about popular IoC containers in use today for .NET Framework and a brief introduction to how DI is used in JavaScript frameworks, like AngularJS

We'll cover the following topics in this chapter:

- Factories and Service Locators--Concepts, Implementations, and how they relate to Dependency Injection
- IoC containers
- Demos about popular IoC containers in use today for .NET Framework
- Introduction to Object Composition, Object Lifetime, and the different types of injection
- Brief introduction to how DI is used in JavaScript frameworks, like AngularJS

WOW! eBook
www.wowebook.org

The Dependency Principle in more detail

But, before all that, let's remember that we postponed a more detailed explanation of the principle for this chapter. In `Chapter 1`, *The SOLID Principles of Software Design*, we discussed the five SOLID principles in general, so it's time to go deeper into the Dependency Principle. Actually, it's not that difficult. It only requires to properly understand the two basic points, which Robert Martin highlights, and express the idea by means of some source code. Recall:

"High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend upon details. Details should depend upon abstractions."

Remember, also, that we explained the intuitive notion of high-level and low-level classes, depending on their functionality for the application. For this respect, you can think of an app as if it was an organization hierarchy, with its different levels which depend on their role in the company.

WOW! eBook
www.wowebook.org

Let's look at an example

So, let's see an example that violates the dependency principle, propose a solution, and see how that solution might take several flavors depending on the application's needs (or even the programmer's tastes, in many situations).

We're going to use a basic .NET app to settle the initial scenario. It's just a classic console application that reads a file with a movie's information from a specific directory, and presents the content in the console.

We'll start with a file called `MovieDB.xml` (in the XML format and stored in the app's data subdirectory) with a data structure like the following:

```
<Movie>
  <ID>1</ID>
  <Title>Jurassic Park</Title>
  <OscarNominations>3</OscarNominations>
  <OscarWins>3</OscarWins>
</Movie>
```

So, we can use LINQ to XML to easily read data from that file and iterate over the results to present a list of movie titles in the console. As a good practice, we'll define a class (the model) that represents the data to be read.

Consequently, we'll have the following:

```
public class Movie
{
  public string ID { get; set; }
  public string Title { get; set; }
  public string OscarNominations { get; set; }
  public string OscarWins { get; set; }
}
```

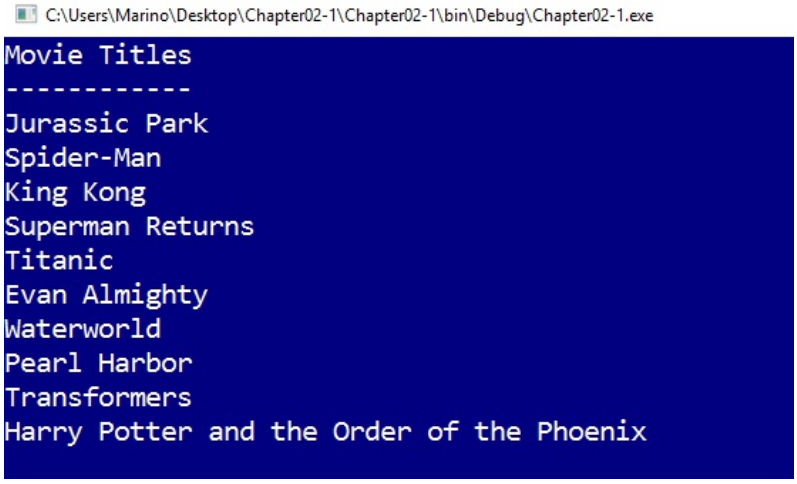
Remember that you can use the Edit menu in Visual Studio and select Paste Special to get the options Paste XML as classes and Paste JSON as classes, which will build a new class in the open editor page and insert a definition according to the data pasted, with the class name `Rootobject`.

A first approach to the problem might end up with the following code (notice I'm using here the initial and simplest approach to have, both, a model to deal with, and the functionality required):

WOW! eBook
www.wowebook.org

```
class Program
{
  static string url = @"Data";
  static XDocument films = XDocument.Load(url + "MoviesDB.xml");
  static List<Movie> movies = new List<Movie>();
  static void Main(string[] args)
  {
    var movieCollection =
        (from f in films.Descendants("Movie")
    select new Movie
    {
      ID = f.Element("Title").Value,
      Title = f.Element("Title").Value,
      OscarNominations = f.Element("OscarNominations").Value,
         OscarWins = f.Element("OscarWins").Value
    }).ToList();
    Console.WriteLine("Movie Titles");
    Console.WriteLine("------------");
    foreach (var movie in movieCollection.Take(10))
    Console.WriteLine(movie.Title);
    Console.ReadLine();
  }
}
```

As you see, we end up with a `List<Movie>` collection, and iterate over it, presenting the first ten results of the `Title` field in the console (see the following screenshot):



Obviously, that's fine for a single use, but not for extended purposes. Since the `Program` class has several responsibilities, it should be decoupled accordingly. So, we can think of a `MovieReader` class which takes care of reading data and apply the first SOLID principle (separation of concerns).

The new `MovieReader` class could look like this:

```
public class XMLMovieReader
{
  static string url = @"Data";
  static XDocument films = XDocument.Load(url + "MoviesDB.xml");
```

WOW! eBook
www.wowebook.org

```
    static List<Movie> movies = new List<Movie>();
    public List<Movie> ReadMovies()
    {
      var movieCollection =
         (from f in films.Descendants("Movie")
      select new Movie
      {
        ID = f.Element("Title").Value,
        Title = f.Element("Title").Value,
        OscarNominations = f.Element("OscarNominations").Value,
        OscarWins = f.Element("OscarWins").Value
      }).ToList();
      return movieCollection;
    }
  }
```

So, we just moved the declarations required to access data to the new class and wrapped the reading functionality around a method `ReadMovies`, which reads and returns the required data.

Our `Main` entry point is now much simpler. Consider the following code snippet:

```
  static void Main(string[] args)
  {
    XMLMovieReader mr = new XMLMovieReader();
    var movieCollection = mr.ReadMovies();
    Console.WriteLine("Movie Titles");
    Console.WriteLine("------------");
    foreach (var movie in movieCollection.Take(10))
    Console.WriteLine(movie.Title);
    Console.ReadLine();
  }
```

That's fine, but still, our `Program` class depends on the `XMLMovieReader`. What happens if we (or somebody else) needs to read data in another format, like JSON, for example?

This is where Dependency Injection comes in. It would be much better than if our `Program` class could depend on an abstraction, not on a concrete class. It could be an abstract class, or it could be an interface.

This implies another class in charge of deciding which concrete implementation has to be served, depending on the file format. In this manner, it could also be possible to add further methods of reading data (like accessing a web service or database) without changing the already working code.

So we could have another specialized reader called `JSONMovieReader` with this implementation:

```
  public class JSONMovieReader
```

WOW! eBook
www.wowebook.org

```
{
  static string file = @"Data\MoviesDB.json";
  static List<Movie> movies = new List<Movie>();
  static string cadMovies;
  public List<Movie> ReadMovies(string file)
  {
    var moviesText = File.ReadAllText(file);
    return JsonConvert.DeserializeObject<List<Movie>>(moviesText);
  }
}
```

So, we just have to implement the appropriate class depending on the format we use. Besides that, given that both files include exactly the same data, we would obtain identical results in both cases (I omitted the output for that reason).

Now, we should create an interface that defines the common operation that all readers will have in common; the `ReadMovies()` method.

```
interface IMovieReader
{
  List<Movie> ReadMovies();
}
```

This interface is the contract that both classes (and other possible candidates) implement, so, we just have to change both declarations to explicitly indicate that they indeed implement the `IMovieReader` interface. In this way, the final definitions will be:

```
public class XMLMovieReader : IMovieReader
...
public class JSONMovieReader : IMovieReader
...
```

And the last step consists of the creation of the new class in charge of deciding which reader has to be used (the `ReaderFactory`, in this demo):

```
public class ReaderFactory
{
  public IMovieReader _IMovieReader { get; set; }
  public ReaderFactory(string fileType)
  {
    switch (fileType)
    {
      case "XML":
      _IMovieReader = new XMLMovieReader();
      break;
      case "JSON":
      _IMovieReader = new JSONMovieReader();
      break;
      default:
      break;
    }
```

WOW! eBook
www.wowebook.org

```
    }
  }
```

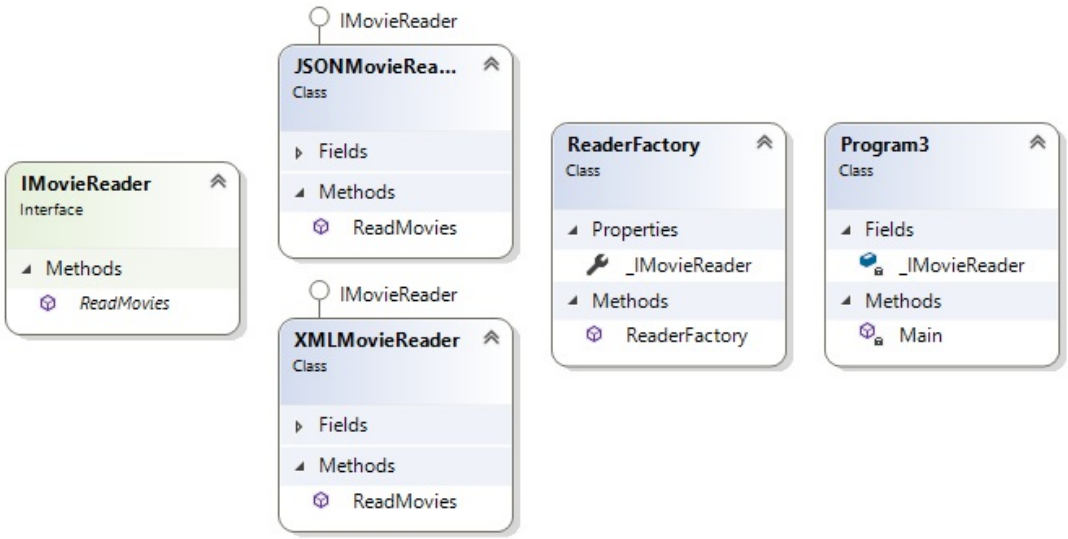
Notice that the constructor of `ReaderFactory` decides which type of reader is assigned to the `_IMovieReader` property. This could easily grow as needed. Our `Program` class has a new definition but it is an extensible one, and we can add as many reading methods as required with few or no changes:

```
class Program3
{
 static IMovieReader _IMovieReader;
 static void Main(string[] args)
  {
   Console.SetWindowSize(60, 15);
   Console.WriteLine("Please, select the file type to read (1)
     XML, (2) JSON: ");
   var ans = Console.ReadLine();
   var fileType = (ans == "1") ? "XML" : "JSON";
   _IMovieReader = new ReaderFactory(fileType)._IMovieReader;
   var typeSelected = _IMovieReader.GetType().Name;
   var movieCollection = _IMovieReader.ReadMovies();
   Console.WriteLine($"Movie Titles ({typeSelected})");
   Console.WriteLine("------------");
   foreach (var movie in movieCollection.Take(10))
   Console.WriteLine(movie.Title);
   Console.ReadLine();
  }
}
```

In this case, we offer an option to choose the file format, and, depending on the user's choice, the `IMovieReader` returned deals with the peculiarities of the format (you can think as well in other formats, such as Excel spreadsheets, pure text formats, comma-delimited files, databases, web services, and so on).

The Class Diagram that Visual Studio generates from this architecture adopts the following aspect (just right-click on the name of the class--`Program3`, in this demo, and select View Class Diagram), to obtain a graphic structure like the following:

WOW! eBook
www.wowebook.org



In summary, the `IMovieReader` interface is the contract that both classes agree upon. As long as any other class implements this interface, we'll be able to extend the potential data access mechanisms with new ways, as mentioned above.

An important part of this implementation is the fact that, from the user interface, we access to a read-only property, `_IMovieReader` inside the `ReaderFactory` class. In that fashion, we avoid further changes in the property, once assigned a value. Finally, we get the `Name` property of the resulting type of `_IMovieReader`, to include it in the final output.

The list is like the previous one, only this time the user can select the type of format (XML or JSON):

WOW! eBook
www.wowebook.org

```
C:\Users\Marino\Desktop\Chapter02-1\Chapter02-1\bin\Debug\Chapter02-1.exe
Please, select the file type to read (1) XML, (2) JSON:
2
Movie Titles (JSONMovieReader)
------------
Jurassic Park
Spider-Man
King Kong
Superman Returns
Titanic
Evan Almighty
Waterworld
Pearl Harbor
Transformers
Harry Potter and the Order of the Phoenix
```

At first sight, you might think that we have to write more code when using the DI approach, but, this only happens when we're dealing with simple demos, like in this case.

In real applications, with thousands or tenths of thousands of lines of code, the amount of required code usually diminishes and it greatly facilitates other aspects of the lifecycle, like maintainability, testability, extensibility, parallel development, and so on.

WOW! eBook
www.wowebook.org

Aspects of Dependency Injection

However, before going on with the distinct aspects of Dependency Injection, it is advisable to recall some of the basic concepts that have a deep influence in the way this principle takes form, and that we should consider before its implementation. Concretely, there are three main points to evaluate--Object Composition, Object Lifetime, and Interception.

Since these three topics are crucial in software development in general (not just when talking about DI), we will go back to them in `Chapter 6`, *Object Lifetime*, `Chapter 7`, *Interception*, and `Chapter 8`, *Patterns - Dependency Injection*, but let's include now this light introduction to serving as a basic reminder of what is to come.

WOW! eBook
www.wowebook.org

Object Composition

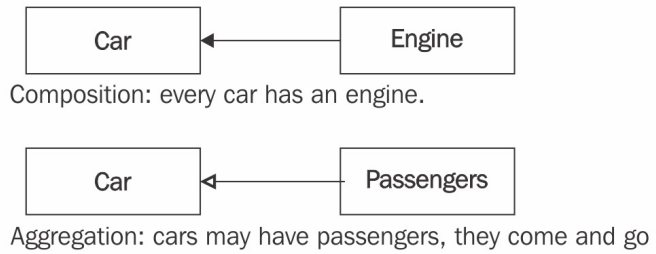
One of the important notions behind Dependency Injection and other SOLID patterns is Object Composition, which, as Wikipedia (https://en.wikipedia.org/wiki/Object_composition) reminds, "*is a way to combine simple objects or data types into more complex ones. Compositions are a critical building block of many basic data structures, including the tagged union, the linked list, and the binary tree, as well as the object used in object-oriented programming.*"

It puts a quite a clear example; types can often be divided into composite and non-composite types, and composition can be regarded as a relationship between types: an object of a composite type (for example, a car) *has an* object of a simpler type (for example, a wheel).

As you know, these relations have been at the heart of OOP, since the very beginning. It also has to do with Aggregation and should not be confused with inheritance.

Actually, there is a well-known principle of the OOP called *Composition over Inheritance*, which states that "*classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class.*"

Thus, the recommendation is to favor object composition over class inheritance:



(Image courtesy: https://atomicobject.com/resources/oo-programming/object-oriented-aggregation)

The previous schema shows the difference between both approaches: composition and aggregation. The individual elements used to compose a car are part of the car.

WOW! eBook
www.wowebook.org

The object could not perform its tasks without them.

In the second case, passengers could come and go (or even, now, since driverless cars don't require any), but they can eventually be managed by the car's instance.

Keep in mind that Wikipedia (https://en.wikipedia.org/wiki/Composition_over_inheritance) remarks that--"*Classes implementing the identified interfaces are built and added to business domain classes as needed. Thus, system behaviors are realized without inheritance. In fact, business domain classes may all be base classes without any inheritance at all. Alternative implementation of system behaviors is accomplished by providing another class that implements the desired behavior interface. Any business domain class that contains a reference to the interface can easily support any implementation of that interface and the choice can even be delayed until run time."*

WOW! eBook
www.wowebook.org

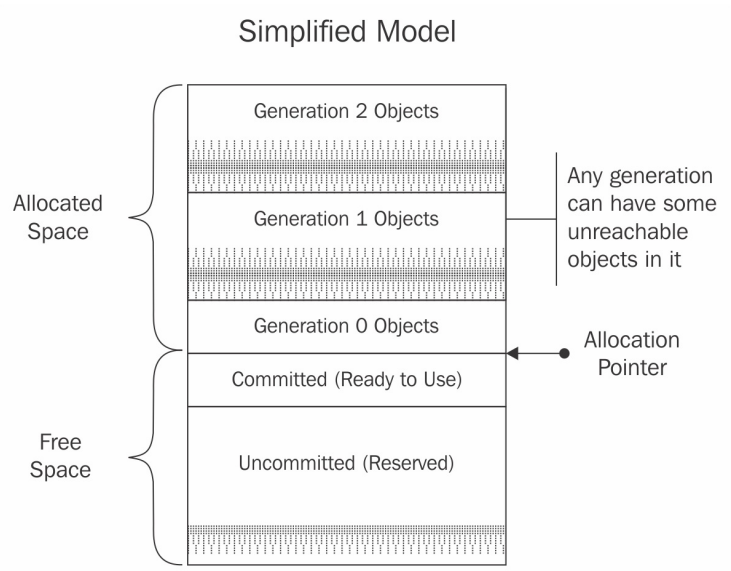
Object Lifetime

In the previous demo, we've seen a way to get rid of a classes' dependencies by means of abstractions, and the possibilities that we have later on to change those abstractions as needed, along with the application's lifecycle.

But, besides this fundamental ability, this practice allows us to determine abstractions' life: when they are born (instantiated) and when they go out of scope (and leave the Garbage Collector in charge of the task of finishing the useful life).

You know, when an object is not referenced by any other object it automatically becomes eligible for destruction, liberating its associated memory.

The way the GC works is not trivial (although transparent to the user) and quite a few things should be considered, especially the way in which object generations are processed and the memory recovered, even in the Simplified Model (see the following image):



(Image courtesy: https://msdn.microsoft.com/en-us/library/ms973837.aspx)

A little about Garbage Collection from Wikipedia (https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))--"In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage or

WOW! eBook
www.wowebook.org

> *memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to simplify manual memory management in Lisp."*

Bear in mind that problems might arise when two objects share an instance of the same interface, or when a new scenario shows up when we inject two distinct instances to different clients.

The way those objects are managed in memory also depends on our code to a large extent. Therefore, we will explain those intricacies in `Chapter 7`, *Interception*, so you can have a detailed picture of this behavior and the implications it might have in your code.

WOW! eBook
www.wowebook.org

Interception

We can consider Interception as an application of the Decorator design pattern. For some authors, Interception is the process of pre-filtering a given call, in a way that we can include (or exclude) certain information from its standard (original) behavior.

In the case of the IMovieReader implementation, the creation of a valid object capable of reading movies is redirected to the `ReaderFactory`, instead of the previous call to a concrete constructor. This is possible because abstractions allow us to defer instantiation and decide what to create depending on arguments, a system's configuration, configuration files, and so on.

Another typical usage of interception relates to Instrumentation: distinct aspects of the application that go beyond application domains, like logging, auditing, validation and many others.

Finally, we can find interception when we capture the default behavior of system components using the so-called sub-classing technique, which allows for interleaving system calls and, effectively, change the system's behavior, substituting it for our own.

WOW! eBook
www.wowebook.org

Ways to implement DI

In this case, Dependency Injection is implemented by means of a constructor, which is the preferred way of today's implementation for many of the popular IoC containers we can find for the .NET Framework (and even, for other frameworks, like Angular).

However, there are other two classical paths for DI implementation: via properties (also called setter injection) or methods.

In the Property Injection flavor, we're dealing with a scenario in which it makes sense to allow the user to change the dependency once the program is running. For instance, imagine that you start with a concrete implementation, and later on, the client or some program's condition, require change.

There are situations in which that change doesn't require -strictly speaking- a new instance of the class, so it would not be coherent to create it just to change a given value. It's much better to keep the level of independence that DI promotes, but allow the client of the dependency to change that value once used.

To achieve this goal we must create a writable property (not a read-only one, like we did before). But, there's a danger. We have to avoid null values. We can do this easily by creating a default value, something very easy and concise to implement using C# latest techniques, as we'll see in a demo. So, a change of a dependency's value is the key to determining if we have to use properties in the injection.

In Method Injection, the code block that requires the dependency is usually a parameter of some method, and the purpose of the dependency's argument is to provide a context, which determines how the method should behave.

So, we can say that the scenario is scope-dependent. When the scope of the dependency is somehow local, as it happens when it is only being used in one concrete method (it does not affect the whole class), it's a good practice to limit the presence of the dependency exclusively to the method that it is going to use it.

Let's see a couple of samples about these two ways of implementing DI.

WOW! eBook
www.wowebook.org

Property Injection in action

Let's imagine something very simple for this demo. The code presents an initial configuration of color values for the current console, together with a message indicating those values. We offer the user the ability to change themes, to avoid unreadable combinations.

We reduce these combinations to light and dark, besides the initial one, which is black/white. We could define a very simple `ConsoleDisplayFactory` class with two default values that are assigned in its instantiation:

```
public class ConsoleDisplayFactory
{
 // Both properties asume a default (initial) configuration
 public ConsoleColor ForeColor { get; set; } = ConsoleColor.White;
 public ConsoleColor BackColor { get; set; } = ConsoleColor.Black;
 public ConsoleDisplayFactory ConfigureConsole (string theme)
 {
   switch (theme)
   {
     case "light":
     BackColor = ConsoleColor.Yellow;
     ForeColor = ConsoleColor.DarkBlue;
     break;
     case "dark":
     BackColor = ConsoleColor.DarkBlue;
     ForeColor = ConsoleColor.Yellow;
     break;
     default:
     break;
   }
   return this;
 }
}
```

With this definition, every time we create an instance, both properties (`ForeColor` and `BackColor`) are assigned the default theme configuration. Our `Program` class will depend on `ConsoleDisplayFactory`, but we make sure that both values hold a consistent color configuration.

Now, our main entry point, located in `Program4`, would look like the following code:

```
class Program4
{
  static ConsoleDisplayFactory cdf = new ConsoleDisplayFactory();
  static void Main(string[] args)
  {
```

WOW! eBook
www.wowebook.org

```
        // Initial config
        cdf.ConfigureConsole("default");
        Console.BackgroundColor = cdf.BackColor;
        Console.ForegroundColor = cdf.ForeColor;
        Console.WriteLine("Console Information");
        Console.WriteLine("-------------------");
        Console.WriteLine("Initial configuration: \n");
        Console.WriteLine($"Back Color: { cdf.BackColor}");
        Console.WriteLine($"Fore Color: { cdf.ForeColor }");
        // User's config
        Console.WriteLine("New theme ('light', 'dark',
            'Enter'=>default):");
        var newTheme = Console.ReadLine();
        cdf.ConfigureConsole(newTheme);
        Console.BackgroundColor = cdf.BackColor;
        Console.ForegroundColor = cdf.ForeColor;
        Console.WriteLine("New configuration: \n");
        Console.WriteLine($"Back Color: { cdf.BackColor}");
        Console.WriteLine($"Fore Color: { cdf.ForeColor }");
        Console.ReadLine();
      }
    }
```

Observe changes to console configuration that are performed via the `ConsoleDisplayFactory` instance, which is a property of the `Program4` class. For this initial version of the demo, we have opted for a method (which serves as a setter) that takes care of the values assigned.

Another approach could be to code the `ForeColor` and `BackColor` properties of the `ConsoleDisplayFactory` class in a way that the setters of each property would take care of assigning the suitable changes for each theme.

Though pretty simple, this code shows the idea behind Property Injection. We don't need the whole class to be re-instantiated, so we allow the client to change the required properties, but taking care that the results are injected according to business rules.

The fact that we use an injector method or program the setters directly will always depend on the architecture of the code and your necessities.

(We omit the output since it is quite trivial in this case).

WOW! eBook
www.wowebook.org

Method Injection in practice

As we previously mentioned, the key here is that the injected resource makes sense inside a method of the client class. There are two main reasons for this, which are the following:

- The injected parameter affects the way the method behaves and it can change along its life-runtime (in distinct method calls)
- The injected parameter's functionality affects the method's block of code and nothing else. Therefore, there's no need to create a class-scope dependency when it is only used inside that block

This scenario shows up when we are coding a method that has some functionality that is dynamic in nature (an HTTP context, access to a directory that contains files that might change during the execution, Web Sockets, and so on.).

It is well-known that the .NET Framework already implements this feature in some classes belonging to a variety of namespaces. For example, in the **Base Class Libraries** (**BCL**), the `System.ComponentModel` namespace allows using the `TypeConverter` class which is especially useful in contexts where WPF is involved, to allow conversions between pure CLR types and XAML types or other business logic types.

Some of this classes' methods use an instance of `ITypeDescriptorContext` that carries information about the context in which the execution takes place.

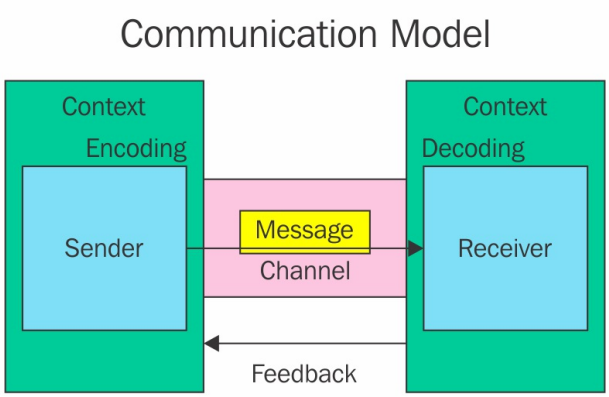
But there's a simpler and much more common scenario in which this is happening all the time: the structure of the .NET's event system, and I think that understanding the way it works is very useful to realize how we can find implementations of this pattern in every-day situations, and how this is being used since the very beginning.

WOW! eBook
www.wowebook.org

.NET event architecture as Model Injection

Let's think of the event model for a second: in practice, method A calls method B and passes some arguments (two, by default). When you program a Click, `SelectedItemChanged`, of `FormClosing` event in classic (and modern) .NET, a communication process takes place.

The process involves a method responsible for calling (the sender) and the callee (the receiver). This can be represented with the classic scheme of any other communication process that we all know from the first books of Information Theory:



Aggregation is a simple collection, like a bag of marbles, while composition implies internal/functional dependencies, like the hinges on a box. Cars aggregate passengers; they get in and out without breaking the car's functionality but the tires are components; remove one and the car no longer functions correctly.

If you don't know these concepts (Composition and Aggregation), PACKT has excellent books to start with that, like Learning Object-Oriented Programming by Gaston C. Hillar (https://www.packtpub.com/application-development/learning-object-oriented-programming).

There is a correspondence between the two schemes in the four elements implied:

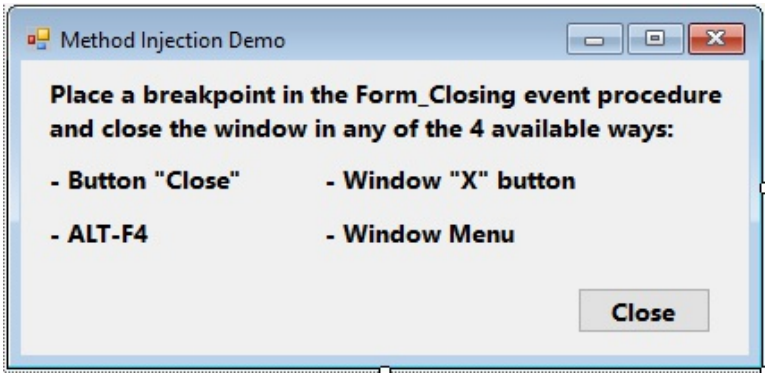
- Issuer (sender): It is the method that makes the call
- Receiver: It is another class (or the same) responding in another method

WOW! eBook
www.wowebook.org

- Channel: It is the environment, replaced by a managed environment in .NET
- Message: The set of values passed to the receiver (the EventArgs in .NET)

Let's think of a Windows app in which the user interface generates events, as, for example, when we use a button to close the window. The piece of code that represents such a scenario is expressed with some code like this:

```
private void btnClose_Click(object sender, EventArgs e)
{
  this.Close();
}
```



Who launches the execution of this code? Well, the Button object, when we program the click event includes the following code in the designer's section of the `Form` class:

```
this.btnClose.Click += new System.EventHandler(this.btnClose_Click);
```

This creates a delegate of type `EventHandler` (the default one) which is going to be in charge of calling the destination method, every time the user clicks on it. To avoid possible problems, that event is just a class with some characteristics:

- Its signature is the same as the method to be called. In this manner, possible casting problems are avoided since there is no need for type conversions.
- The delegate checks for the presence and availability of the `btnClose_Click` method before the call, so it guarantees there are no pointer-to-null problems.

Given that casting and pointers-to-nowhere were the two of the main causes of the infamous BSODs (Blue Screens of Death), the implantation of this architecture was something crucial from the very beginning of .NET.

However, there is something else here that implies Method Injection, if you analyze the code and, even in those cases, like in the next demo, when the delegate is not the default.

WOW! eBook
www.wowebook.org

To really appreciate this point, let's program the `FormClosing` event that will be launched whenever the user clicks the Close button or tries to close the window in any of the other available ways: *Alt* + *F4*, the window's x button, or the window's menu:

```
1 reference
private void frmMethodInjection_FormClosing(object sender, FormClosingEventArgs e)
{
    e.
}
```

Cancel — bool CancelEventArgs.Cancel { get; set; }
CloseReason — Gets or sets a value indicating whether the event should be canceled.

As you can see, this time the second argument of the `FormClosing` event is not the default, but an instance of an object that inherits from `EventArgs`, and encloses extra information providing the context, that we were talking about.

Actually, that object holds two properties: `Cancel` (which is assignable and forces a stop in the exit procedure), and `CloseReason`, a read-only property that indicates which mechanism really launched the closing process.

So, we're not programming or instantiating this parameter: it is given to us via injection, every time we define an event procedure. And that internal injection system is responsible for providing the information relevant to the execution context. This is a clear sample of method injection.

If you peek into the definition of the `FormClosingEventArgs` parameter, you will see that it is, indeed, another class that inherits from `CancelEventArgs` (which, in turn, inherits from EventArgs):

```
public class FormClosingEventArgs : CancelEventArgs
{
 //
 // Summary:
 // Initializes a new instance of the
        System.Windows.Forms.FormClosingEventArgs class.
 //
 // Parameters:
 // closeReason:
 // A System.Windows.Forms.CloseReason value that represents
     the reason why the form
 // is being closed.
 //
 // cancel:
 // true to cancel the event; otherwise, false.
 public FormClosingEventArgs(CloseReason closeReason, bool cancel);

 //
```

WOW! eBook
www.wowebook.org

```
    // Summary:
    // Gets a value that indicates why the form is being closed.
    //
    // Returns:
    // One of the System.Windows.Forms.CloseReason enumerated values.
    public CloseReason CloseReason { get; }
  }
```

It's interesting to note that `ClosingEventArgs` also belongs to the namespace `System.ComponentModel` that we mentioned previously.

So, even when we're doing something as simple as closing a window, we're implicitly using method injection, that is at the heart of the .NET Framework.

This architecture can be extended by the user in many ways, even using techniques like event chaining when we need to concatenate processes that depend on the user's choice and that are usually generated via events, for example.

A typical case for even chaining is when a user's selection from a combo (Like Select a Country), generates code that fills up another combo, let's say Select a City. You first have to select the country if you want the cities combo to be filled with the cities belonging to a given country.

One such case could be when the closing process of a window (something like the preceding code shown) requires an extra user intervention. For instance, imagine that you have to ask the user if he/she wants to save the audits (or perform any other action) but only in the case that a previous question is affirmative, like confirming that the user wants to exit the application (which might, in turn, depend on other conditions, like the `CloseReason` expressed in the previous code of the FormClosing event.

One possible approach would be to create a generic event on our own, that could be launched if the requested conditions apply. Let's say that only if `ClosingReason` is `CloseReason.UserClosing`, we should ask the user to confirm application exit and, in case the answer is positive, ask him again if he/she wants to save the information.

We could write the following:

```
private void frmMethodInjection_FormClosing(object sender,
   FormClosingEventArgs e)
{
  if(e.CloseReason == CloseReason.UserClosing)
  {
```

WOW! eBook
www.wowebook.org

```
        var res = MessageBox.Show("Confirm application exit?", "Notice",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question);
        if (res == DialogResult.Yes)
        FormClosingExtended?.Invoke(this, DateTime.Now.ToLongTimeString());
        else e.Cancel = true;
      }
    }
    public event EventHandler<string> FormClosingExtended;
```

So, we're invoking the event `FormClosingExtended` only when the if block evaluates to true. But, using the new generic event handler we can pass login information to another event that jumps from there to a distinct event procedure:

```
    private void FrmMethodInjection_FormClosingExtended(object
       sender, string e)
    {
      var ans = MessageBox.Show($"Save Closing time: ({e})?","Notice",
      MessageBoxButtons.YesNo,MessageBoxIcon.Question);
      if (ans == DialogResult.Yes)
      {
         File.WriteAllText("ClosingTime.txt", e);
      }
    }
```

This event procedure receives data in the e argument and also asks the user if he/she wants to save it as audit information. In this manner, we don't have to concatenate two `MessageBox` calls and the code is clearly separated.

Besides that, since the e argument could be a generic event handler of any kind, the type of information passed as the event argument, could also be of any kind, of any complexity level. For instance, it could be an object with its own methods suitable for validation purposes, security checking, etc.

The only thing missing in the code is to define the event handlers for each of the events implied in the process, which we can do inside the `MethodInjection` constructor:

```
    public frmMethodInjection()
    {
      InitializeComponent();
      this.FormClosing += frmMethodInjection_FormClosing;
      FormClosingExtended += FrmMethodInjection_FormClosingExtended;
    }
```

So, in terms of DI, we're using the internal injection engine inside .NET Framework in two different ways:

- In the first case, to obtain information passed to us by the framework in relation to the cause that provoked that event to happen (`ClosingReason`).

WOW! eBook
www.wowebook.org

- In the second, because we have implemented our own event, that is executed in concatenation to the previous one and receives the required information expecting the user's approval.

Note that, in case that no business logic argument is required for the second event handler, we could have registered it also for the FormClosing event, since events are multicast in .NET

WOW! eBook
www.wowebook.org

DI containers

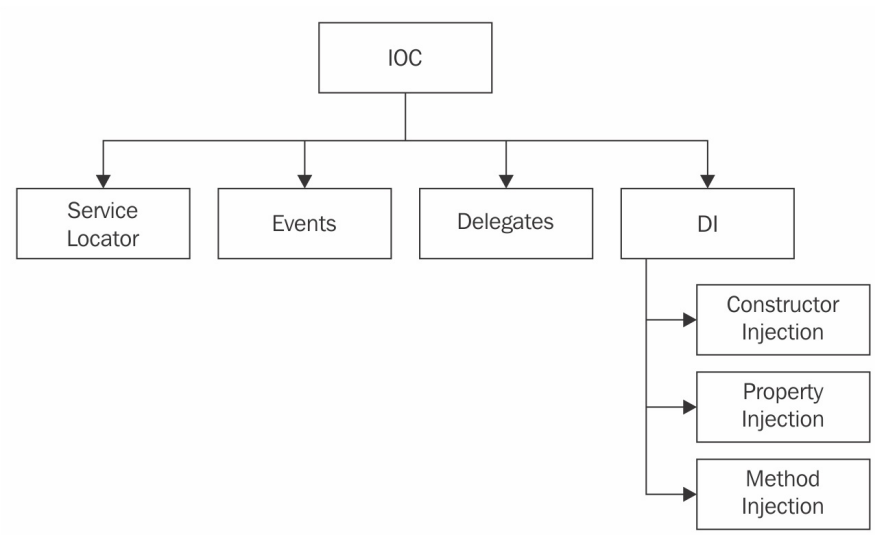
In everyday practice, a good part of the plumbing to have DI working is not done via manual coding, but using something called DI Containers (also called IoC Containers). The previous demo using the .NET Framework event's system implies the existence of some DI Container.

Basically, a DI Container is a software library that facilitates the use of Dependency Injection and its based methodologies, like object composition, lifetime management, and so on. This is quite convenient for the programmer, who doesn't have to worry about the details of creation, destruction, etc.

Many people have some confusion with DI and IoC. You can refer to this website to clear out any confusions: `http://www.itgo.me/a/8238364259424879474/inversion-of-control-vs-dependency-injection.`

In many cases, DI Containers enforce several good practices, like avoiding duplication of instances, by supplying singleton instances of the required component, like it happens in AngularJS since the initial versions.

In .NET, as we have mentioned, IoC is present in several scenarios, and the following image shows some of the main components that use this technique:



(Image courtesy: `https://hotproton.com/category/dependency-injection/`)

WOW! eBook
www.wowebook.org

As you can see in the graphic, .NET (even in the previous versions), holds several components that provide Inversion of Control (IoC), either linked to Events and Delegates, as we've seen in previous demos, or as Service Locators, or DI Injection mechanisms.

WOW! eBook
www.wowebook.org

Service locators

Basically, a service locator is just a design pattern that instructs the implementer on how to construct DI containers capable of obtaining a service with a strong abstraction layer.

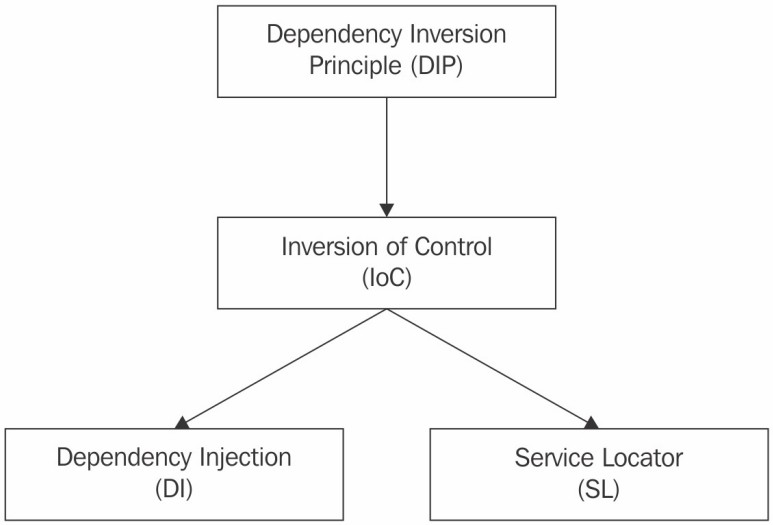
There's a condition that all service locators require: services have to be registered, so they can be later found and served at the code's request.

Wikipedia (http://www.itgo.me/a/8238364259424879474/inversion-of-control-vs-dependency-injection) summarizes the advantages of Service Locators in three main points:

- The service locator can act as a simple run-time linker. This allows code to be added at run-time without re-compiling the application, and in some cases without having to even restart it.
- Applications can optimize themselves at run-time by selectively adding and removing items from the service locator. For example, an application can detect that it has a better library for reading JPG images available than the default one, and alter the registry accordingly.
- Large sections of a library or application can be completely separated. The only link between them becomes the registry.

In general, we can say the application of the Dependency Inversion principle leads to the construction of IoC containers and they reflect in concrete DI techniques and Service Locators. (See the following diagram):

WOW! eBook
www.wowebook.org



(Image courtesy: http://www.dotnettricks.com/learn/dependencyinjection/understanding-inversion-of-control-dependency-injection-and-service-locator/)

Of course, these advantages don't go without their counterparts. Possible problems include the registry acting like a black box for the rest of the application, uniqueness, security vulnerabilities, hiding class dependencies, adding some testing difficulties, and so on.

WOW! eBook
www.wowebook.org

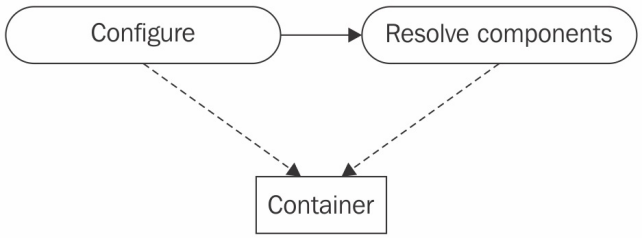
DI Containers for .NET

Besides the internal functionality related to DI inside .NET, it is very common to use external containers that provide extra or extended functionality to programmers, and, being .NET--a very popular programming framework, many have proliferated in recent years.

So, our criteria here will be showing the basic implementation of those that seem to have a greater acceptance in the community, like Unity, Castle Windsor, StructureMap, and Autofac.

Actually, the number of choices available has kept growing in the last few years, and some of them are pretty popular as well, like Ninject, Simple Injector, Dynamo, LinFu, Spring.NET, Hiro, and so on, so our choice, in this case, was mainly driven by the level of implementation in the community, their ease of use, their API's consistency, and the performance tests in the latest versions. Since I don't want to pronounce myself about any of those tests, you can take a look at different benchmarks available on the web, and you'll probably come to the same or similar conclusion as I did.

These DI Containers (and others) share a few things in common: they all require a previous configuration, and they have the ability to resolve the required dependencies at runtime. The following schema shows an approach to this idea:



In practice, this means we will instantiate and configure a container object, and later on, we'll ask the container to resolve the required dependencies in one or more locations within our code.

Besides that, in most of the cases, components are resolved from the same instance that we initially instantiate.

WOW! eBook
www.wowebook.org

A common (and very simple) context for our demos:

Since complications will show up themselves when coding real-life applications, I've chosen a very, very, simple starting point, that can serve us as a common scenario to solve the DI problems it presents. It's based on our previous idea of the `MovieReader` but let's say in this case that we don't even read anything from disk (only display a message in the console), to focus on the code's architecture and not on the details of its implementation.

The demo proposes the existence of two classes in charge of reading some books' information from the disk, and they both share a common interface `IBookReader`, which implements a unique method `ReadBooks()`. These three elements comprise the Data Access Layer:

```
// Data Access Layer
public interface IBookReader
{
  void ReadBooks();
}
public class XMLBookReader : IBookReader
{
  public void ReadBooks()
  {
    Console.WriteLine("Books read in XML Format");
  }
}
public class JSONBookReader : IBookReader
{
  public void ReadBooks()
  {
    Console.WriteLine("Books read in JSON Format");
  }
}
```

Pretty simple, right? Now, we build another succinct Business Layer, made up of a class called `BookManager`, whose only purpose is to perform business logic, so it exposes a public constructor that receives an instance of one of the two possible readers, and implements a call to the `ReadBooks` method, that in turn, will refer to the reading method that corresponds to each case:

```
public class BookManager
{
  public IBookReader bookReader;
  public BookManager(IBookReader reader)
  {
    bookReader = reader;
  }
  public void ReadBooks()
  {
    bookReader.ReadBooks();
```

WOW! eBook
www.wowebook.org

```
    }
  }
```

Finally, in the user interface, that in this case is the `Console`, we ask the user to decide the reading mechanism and call the corresponding version of the `BookManager`, so we can call the `ReadBooks` method:

```
  static void Main(string[] args)
  {
    UnityContainer uc = new UnityContainer();
    BookManager bm;
    Console.WriteLine("Please, select reading type (XML, JSON)");
    var ans = Console.ReadLine();
    if (ans.ToLower() == "xml")
    {
       bm = new BookManager(new XMLBookReader());
    }
    else { bm = new BookManager(new JSONBookReader()); }
      bm.ReadBooks();
    Console.ReadLine();
  }
```

Up to this point, it's a very simple code, already using some dependency inversion, but still stuck to the creation of the type of reader in the user interface. That's exactly what we obtain when we use an external creation mechanism that takes care of this work for us.

Let's see now how we can change this using the distinct DI Containers mentioned previously.

WOW! eBook
www.wowebook.org

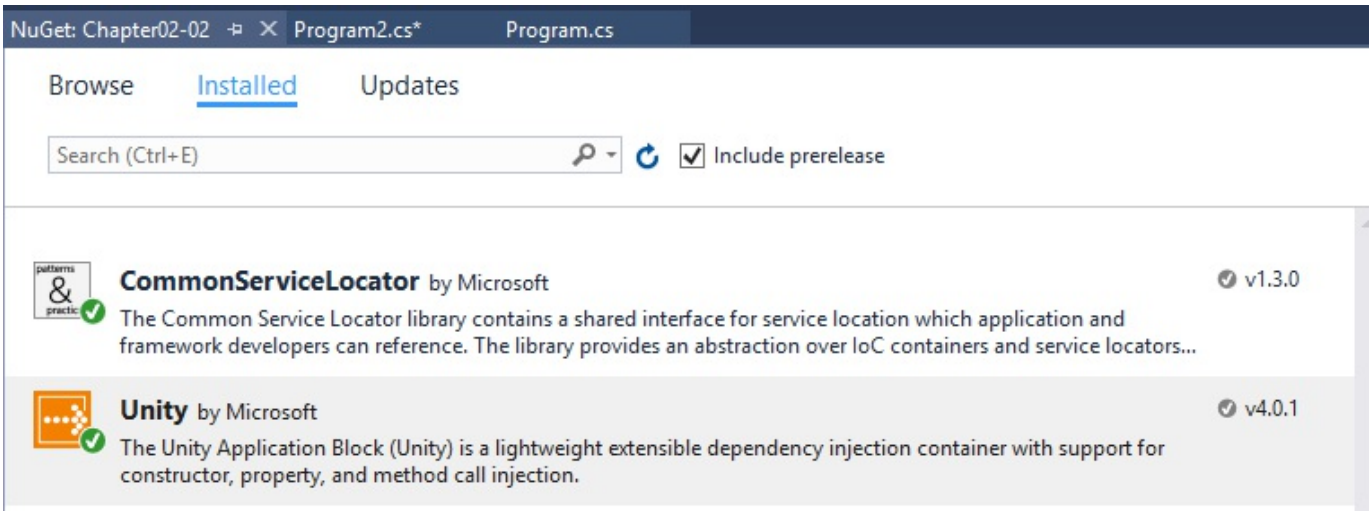
Using Unity Container

The Unity container has been in place for quite a few years now. Somehow, it was the official Microsoft external container and has been linked to the *Patterns & Practices* initiative for many years.

Please note that Unity is not an official Microsoft product or does not belong to the Patterns & Practices team anymore. The project has been redirected to other people (Pablo Cibraro and Pedro Wood), as Immo Landwerth published in .NET Blog in 2015 (`https://blogs.msdn.microsoft.com/dotnet/2015/08/21/the-future-of-unity/`), commenting: "*Dependency injection containers for .NET have continued to mature and evolve significantly. In addition, open source components are now more accepted. The need for having an "official" container from Microsoft is no longer as widespread as it once was.*"

That said, Unity is still a very common feature in thousands of projects, and it has reached version 4.01, which you can install inside any project using the `NuGet Package Manager` available from the contextual menu of the `Solution Explorer`, or the `Project` menu as well, either in V. Studio 2017 or other versions.

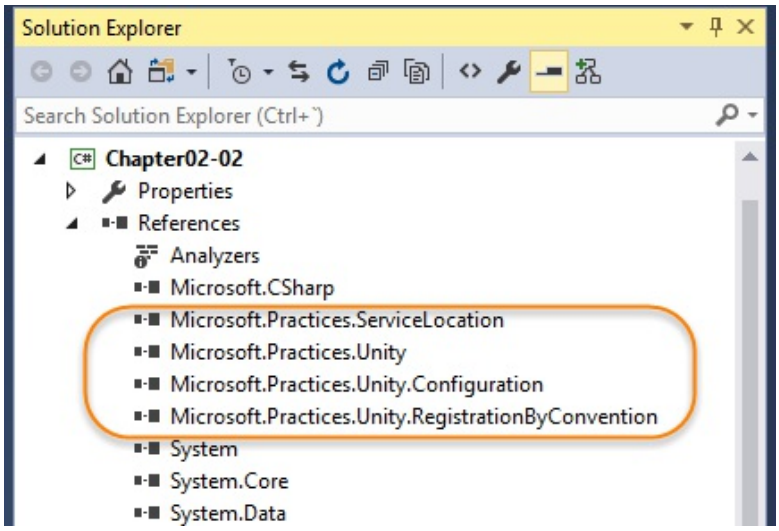
Once installed, you'll see that it really references two different libraries: Unity 4.01 and the CommonServiceLocator 1.30 library, as you can see in the following screenshot:



Once installed, you'll see four new references to DLL's in the Solution's Explorer:

WOW! eBook
www.wowebook.org

three of them belong to Unity and the fourth one to `CommonServiceLocator`:



Inside the `Microsoft.Practices.Unity.Configuration` library, you have utilities that allow writing the configuration required in an XML file, so that it serves as the initial settlement for the DI Container. Classes in that namespace will allow you to read and configure a given execution context based on that XML.

On the other side, the `Microsoft.Practices.Unity.RegistrationByConvention` library is intended to provide a programmable way of configuration to register multiple types with the container automatically by using a set of rules and conventions, as the official documentation defines (https://msdn.microsoft.com/en-us/library/dn507479(v=pandp.30).aspx).

Now, if we just want to register those classes belonging to our Business Layer and Data Access Layer, the most evident way to include all elements in our business model and have our data ready, would be something like this:

```
    static void Main(string[] args)
    {
      UnityContainer uc = new UnityContainer();
      uc.RegisterType<BookManager>();
      uc.RegisterType<IBookReader, XMLBookReader>();
      uc.RegisterType<IBookReader, JSONBookReader>();
      BookManager bm = uc.Resolve<BookManager>();
      bm.ReadBooks();
      Console.ReadLine();
  }
```

Notice, however, that we're defining `BookManager` together with `XMLBookReader` and `JSONBookReader`. This means that if we run the code, we get an instance of the last registered class (`JSONBookReader`), which becomes the default option. The reason is that

WOW! eBook
www.wowebook.org

we're not naming these registrations, so they're assigned unnamed identifiers.

You can test the demo inside `Chapter02_02.Unity` namespace and insert breakpoints to prove it.

To reproduce the initial situation in which the user selects the format, we need to register distinct aliases for registered types, so they can be resolved at runtime, passing the concrete version that we need.

Besides that, notice that Unity plays the role that was previously playing the `BookManager` class. So in this case, we don't need the `BookManager` class anymore:

```
static void Main(string[] args)
{
  Console.WriteLine("Please, select reading type (XML, JSON)");
  // we asume a predefault value
  var format = (Console.ReadLine() != "xml") ? "json" : "xml";
  UnityContainer uc = new UnityContainer();
  uc.RegisterType<IBookReader, XMLBookReader>("xml");
  uc.RegisterType<IBookReader, JSONBookReader>("json");
  IBookReader ibr = uc.Resolve<IBookReader>(format);
  ibr.ReadBooks();
  Console.ReadLine();
}
```

Now, Unity resolves the dependency with the parameter we pass to the `Resolve()` method, as we can see it by setting a breakpoint in this line, or simply, watching the output.

The `UnityContainer` class admits alternative registration mechanisms. For example, we could use a new business layer class entirely dedicated to Unity registration, with the following code (notice that we should reference `Microsoft.Practices.Unity` in the `using` section of our code):

```
public class UnityRegistration
{
  public void Register()
  {
    using (var container = new UnityContainer())
    {
      container.RegisterTypes(
      AllClasses.FromLoadedAssemblies(),
      WithMappings.FromAllInterfaces,
      WithName.Default,
      WithLifetime.ContainerControlled);
    }
  }
}
```

WOW! eBook
www.wowebook.org

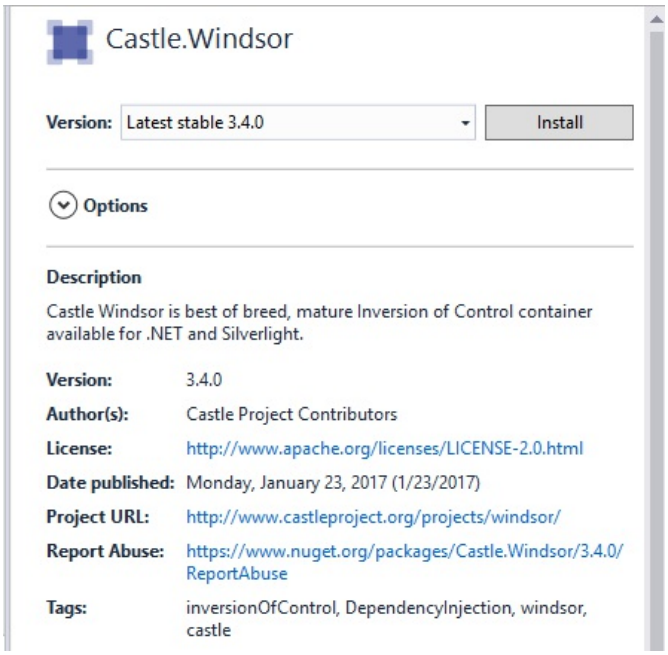
In this manner, all classes from the loaded assemblies are registered for Unity, all mappings (or correspondences) are defined between the existing interfaces and their implementing classes, their default names are used and their lifetime is assigned to be managed by the container, so it's the container itself which decides at runtime when to leave object's instances to the Garbage Collector.

WOW! eBook
www.wowebook.org

Using Castle Windsor

One of the winners in benchmarks and usability tests, Castle Windsor has been around for quite some time, and now rallies all activities around its dedicated GitHub project site at `https://github.com/castleproject/Windsor.`

The community around this project is very active and, at the time of writing these lines, has more than 500 stars and 265 forks, and they were preparing to release version 3.4. Of course, you can download it and install it separately, or use `NuGet` and have it installed for your project in the usual manner:



The process of installation really installs two components: Castle.Core 3.3 and Castle.Windsor 3.4. Both work together, although they include several namespaces to cover many of the possible programming situations in which we might need dependency injection (and other features).

Castle Windsor's API set is very rich in possibilities and the documentation in the official site lets you start quickly with some samples (see `https://github.com/castleproject/Windsor/blob/master/docs/basic-tutorial.md`).

For our demo, we only have to reference those namespaces required to initialize the `WindsorContainer` class and proceed with the registration:

WOW! eBook
www.wowebook.org

```
using Castle.Windsor;
using Castle.MicroKernel.Registration;
```

The first one allows the creation of a new `WindsorContainer` class, while the other defines the required classes for registration. The whole process is similar to the one we saw previously with Unity:

```
static void Main(string[] args)
{
  Console.WriteLine("Please, select reading type (XML, JSON)");
  // we asume a predefault value
  var format = (Console.ReadLine() != "xml") ? "json" : "xml";
  var container = new WindsorContainer();
  container.Register(Component.For<IBookReader>().
     ImplementedBy<XMLBookReader>().Named("xml"));
  container.Register(Component.For<IBookReader>().
       ImplementedBy<JSONBookReader>().Named("json"));
  IBookReader ibr = container.Resolve<IBookReader>(format);
  ibr.ReadBooks();
  Console.ReadLine();
  // clean up, application exits
  container.Dispose();
}
```

Notice how the Component class includes static, generic methods to allow definitions of any interface (`IBookReader`, here), and you can concatenate successive calls to indicate which class implements what interface and what name that we want to assign to each registration, so it can be solved at runtime later on.

Once the registration is completed, the way to resolve it in a concrete implementation accepts a format that is identical to the one we used with Unity.

The code just runs exactly the same as in the previous demo.

WOW! eBook
www.wowebook.org

Using StructureMap

The official site for this DI container defines precisely the differences and spirit behind this implementation:

`StructureMap` is the oldest, continuously used IoC/DI container for .NET dating back to its first public release and production usage all the way back in June 2004 on .NET 1.1. The current 4. release represents 12+ years of lessons learned in the StructureMap and greater .NET community--while also wiping away a lot of legacy design decisions that no longer make sense today.

So, we're dealing with a veteran here, and that means stability, together with a wide presence in internet forums and coder's sites, like `StackOverflow`.

The philosophy that lies behind usage and configuration is quite similar to the other two we've already seen, but it offers several ways to configure the application. As the official documentation states to this respect: "*from release 3.0 onwards, StructureMap provides a streamlined fluent interface called the Registry DSL to configure a StructureMap Container with both explicit registrations and conventional auto-registrations.StructureMap no longer supports XML configuration or MEF-style attribute configuration -- but there is some facility for rolling your own attribute-based configuration support.*"

The main difference is that it recommends a configuration approach via lambda expressions, but still with similar mechanisms, as you can see in the following code used to create and configure the Container object in the same operation:

```
var container1 = new Container(c =>
{
  c.For<IFoo>().Use<Foo>();
  c.For<IBar>().Use<Bar>();
});
```

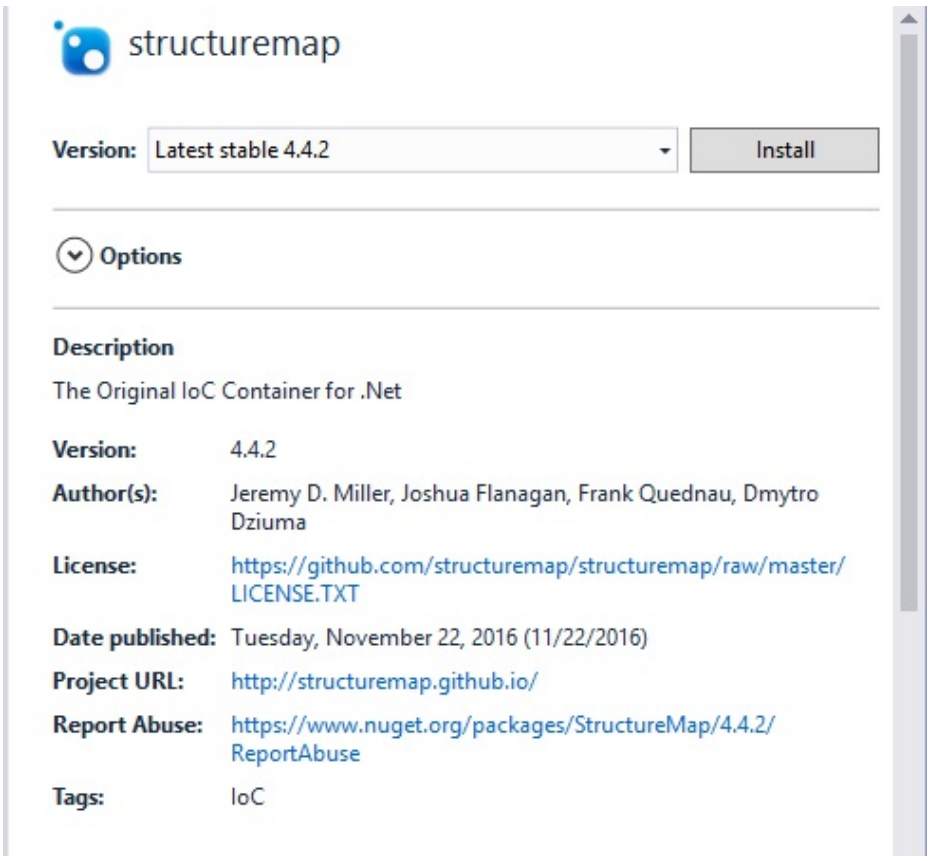
The other main choice relies on creating a `Registry` object and then configuring the container based on it. Something like the following:

```
public class FooBarRegistry : Registry
{
  public FooBarRegistry()
  {
    For<IFoo>().Use<Foo>();
```

WOW! eBook
www.wowebook.org

```
        For<IBar>().Use<Bar>();
      }
    }
    var container1 = new Container(new FooBarRegistry());
```

All this depends on the architecture and complexity of the application to build. For our demo purposes, we will first reference the library (this time there is only one namespace), via NuGet, that presents this installation option:



And, for a basic configuration and usage, we only need to reference the basic `StructureMap` namespace:

```
    using StructureMap;
```

The source code equivalent to the previous demos would be (with identical results in execution):

```
    static void Main(string[] args)
    {
      Console.WriteLine("Please, select reading type (XML, JSON)");
      // we asume a predefault value
      var format = (Console.ReadLine() != "xml") ? "json" : "xml";
      var container = new Container();

      // container configuration
      container = new Container(x => {
```

WOW! eBook
www.wowebook.org

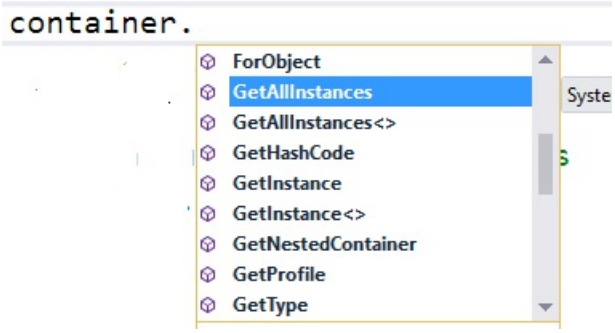
```
        x.For<IBookReader>().Add<XMLBookReader>().Named("xml");
        x.For<IBookReader>().Add<JSONBookReader>().Named("json");
      });
        var ibr = container.GetInstance<IBookReader>(format);
        ibr.ReadBooks();
        Console.ReadLine();
        // clean up, application exits
        container.Dispose();
      }
```

Notice how the container is configured by passing a lambda expression to the newly created instance of Container and, within the body expression, we use the following pattern:

```
container à For <Interface> à Add(Class) à Named("alias")
```

The preceding pattern allows us to express as many registrations as we want in a single operation.

There's only a slight variation in the way the `IBookReader` instance is obtained because it doesn't use the resolve paradigm. In turn, we can find several methods of resolving instances, as shown in the following screenshot:



Of course, execution is the same as in the other cases and there is nothing really relevant in the output, which you can find in the source code for `Chapter02_02.StructureMap` demo.

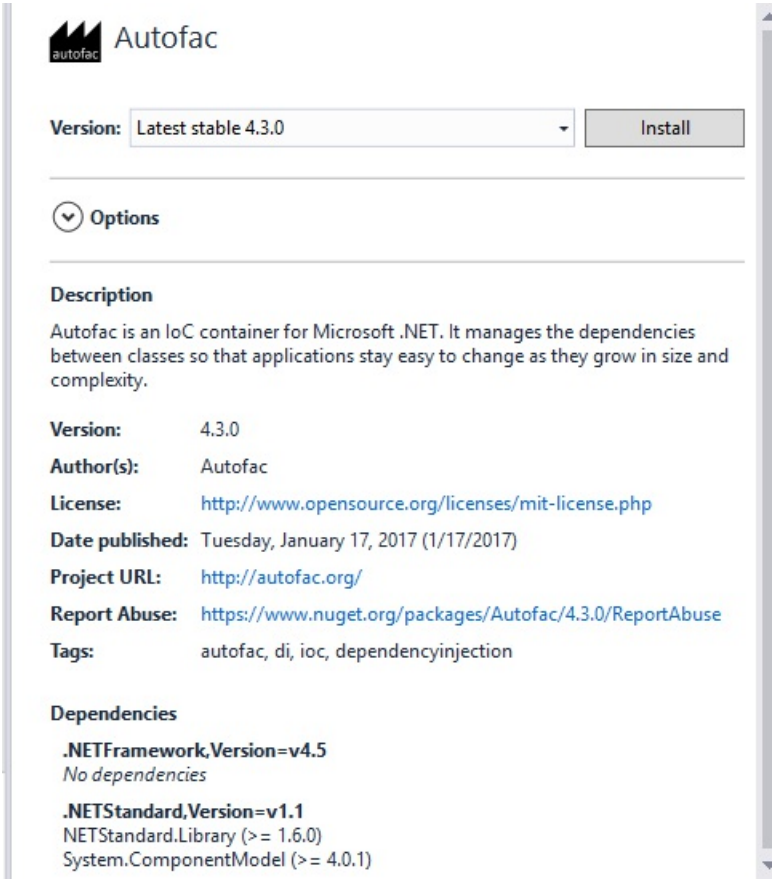
WOW! eBook
www.wowebook.org

Using Autofac

We'll finish this short visit over .NET containers examining the basics of AutoFac, another DI Container well known in the community, and that claims to be up-to-date with versions that cover .NET Core, ASP.NET Core, Universal Windows Apps, and .NET Framework 4.5.1 and later versions. It also allows support for WCF-based applications.

It has its own dedicated site (`https://autofac.org/`) which serves as the starting point, although it is also referenced via `NuGet` packages. Besides this, you will find references to several libraries, some specialized, in this page or NuGet.org at `https://www.nuget.org/packages/Autofac/`.

If you decide to keep on using `NuGet` from Visual Studio, you should find the following reference, once you search for Autofac in the NuGet package editor:



The standard architecture is similar to the other three we've already seen, with minor differences. For example, the container here is named `ContainerBuilder`.

WOW! eBook
www.wowebook.org

After its instantiation, we have to configure the required types and interfaces and, finally, we should call to the `Build()` method of the `ContainerBuilder` to have everything ready.

Although we might use an approach similar to the other demos, in this case, we decided to inject only the version chosen by the user. This can be easily accomplished with the following code:

```
static void Main(string[] args)
{
  Console.WriteLine("Please, select reading type
     (XML, JSON)");
  // we asume a predefault value
  var builder = new ContainerBuilder();
  if (Console.ReadLine() != "json")
  {
    builder.RegisterType<XMLBookReader>().As<IBookReader>();
  }
  else
  {
    builder.RegisterType<JSONBookReader>().As<IBookReader>();
  }
  var container = builder.Build();
  var ibr = container.Resolve<IBookReader>();
  ibr.ReadBooks();
  Console.ReadLine();
}
```

In short, we have many options regarding IoC containers, and many choices about the way to configure them, but all of them provide us with similar functionality: we can abstract dependencies that are resolved at a later time.

Of course, in this and other cases we could opt for other forms of configuration, like using XML or JSON files and also use more complex configuration classes to support all possible situations that our application requires.

Although this is only an introduction to IoC containers, if you are interested in these APIs, you will see that it is possible to handle the three aspects discussed at the beginning: Object Composition, Object Lifetime, and Interception.

WOW! eBook
www.wowebook.org

Dependency Injection in other frameworks

.NET or Java are not the only programming environments in which we can find Dependency Injection containers. Many of the most popular JavaScript frameworks also support DI from its inception. This is the case for AngularJS.

WOW! eBook
www.wowebook.org

DI in Angular

AngularJS (1.x) and Angular (2.x, 4.x, etc) are perhaps the most popular frameworks for JavaScript applications in use today. They are pretty different in their programming model and general purpose, so I will distinguish them both:

Angular frameworks are the result of a Google teamwork lead by Misko Hevery, Igor Minar, and Brad Green, which initially appeared in 2010. The project has bifurcated into two distinct branches: AngularJS 1.x for small/medium projects and Angular 2 (or just Angular) which is intended to cover the needs of big/complex projects and uses TypeScript as the programming language due to its strongly typed nature.

In December 2016, they announced the adoption of a semantic versioning approach, together with a continuous delivery roadmap that will publish a new version every six months, very careful of breaking changes. The latest version is Angular 4, which appeared in March 2017, and is totally backward compatible with Angular 2.

Both projects are also maintained by Google and, despite their different programming models and language, they share a few things in common: They promote the **Single Page Application** (**SPA**) model, and they use a **Model-view-controller** (**MVC**) architecture to provide a suitable separation of responsibilities from the first moment.

AngularJS presents itself as a bunch of libraries so the user can choose only the ones needed for a given purpose, achieving that way a better granularity. All libraries are available via GitHub, NuGet, NPM, Bower, and so on.

WOW! eBook
www.wowebook.org

Our first demo

Let's get down to work with our first demo, and, from this initial approach, we will test how AngularJS provides an excellent, integrated dependency injection system, that facilitates programmers' work to a large extent.

By the way, we can use any IDE to work with Angular, since the three languages implied (two in case we don't care about the visual aspects provided by CSS) are just HTML and JavaScript.

However, I'll keep on using Visual Studio, which has a very good support for Angular programming and provides native Intellisense on Angular directives, and a pretty good debugging experience.

So, let's start a new project or a new website (we don't need any compiled libraries). We can choose an Empty Project in the ASP.NET section. This creates a project with only a `web.config` file in case we need to configure the **Internet Information Server** (**IIS**).

Inside the project, we'll create a new HTML file, and once the project is saved we can add the Angular libraries in the usual manner, via the `NuGet Package Manager`. We should see that AngularJS is now in versions 1.6.x or higher (we should select Angular.Core for this demo, which is the basic module).

Once we accept the installation, we'll see a new `Scripts` folder containing the development (`angular.js`) and deployment (`angular.min.js`) versions of Angular 1.6, plus a Mock library for testing purposes.

We just have to include the development library and create the minimum required plumbing to see Angular in action--just by including the library, some objects and services are loaded into memory and ready to go.

Especially, there is a `$injector` object that will take care of retrieving object instances, instantiate types, load modules and invoke methods.

Also, Angular creates an initial, basic model which serves as the root model for the application ($rootScope), and expects that the user marks an HTML element as the

WOW! eBook
www.wowebook.org

scope of the app. We'll do that in the `<body> tag`, name it `app`, and define a module with the same name, with the following syntax:

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8" />
 <title>Dependency Injection Demo</title>
 <script src="Scripts/angular.js"></script>
</head>
<body ng-app="app">
 <h1>Dependency Injection Demo</h1>
 <h3>Current time: {{ time }}</h3>
 <script>
   var app = angular.module("app", []);
 </script>
</body>
</html>
```

Notice we use an Angular-specific attribute (`ng-app`) to mark the scope of our application. These attributes are called directives in Angular and they allow us to modify the DOM to suit our needs (All predefined directives start with `ng-`).

Now, we want our page to show the local time when it loads in the browser. The angular mode to do this is to select a target area of the DOM and assign a controller to that area. We can do that including the `ng-controller` directive inside the corresponding tag.

As you can see in the code, next to the page's title we've included a `<h3>` tag to hold that information, and inside that tag's text, a message, followed by `{{ time }}`.

The double brackets syntax is one of the possible binding mechanisms that Angular uses to link data inside the model with the active view, and it's called **moustache syntax**. Thus, we modify the tag in this way:

```
<h3 ng-controller="TimeController">Current time:
   {{ time }}</h3>
```

Now we have a `ng-controller` directive named `TimeController`, that we need to define inside the script. Controllers are defined by calling the app's main module, that we previously created, and passing a function as the second argument that will be in charge of the logic linked to that controller.

And, here, we finally get to the Dependency Injection mechanism of AngularJS. This function should have access to the specific model created by Angular to hold the information managed by the controller. We do that via the `$scope` object.

WOW! eBook
www.wowebook.org

Every controller has its own $scope (it's like a sub-model), which allows read/write operations in memory and permits binding between the moustache expressions in the DOM and data stored in that model.

So, how does the user code access to that (or any other) service? Via DI, of course. The final code is surprisingly simple:

```
<body ng-app="app">
<h1>Demo Dependency Injection</h1>
<h3 ng-controller="TimeController">Current time:
   {{ time }}</h3>
<script>
  var app = angular.module("app", []);
  app.controller("TimeController", function ($scope) {
  $scope.time = new Date().toLocaleTimeString();
 });
</script>
</body>
```

As you see, the controller's definition receives a string to identify the controller's name, and an anonymous function, to hold the functionality linked to that controller.

And, in that function, we just declared the $scope object: the Angular's injection system takes care of the rest. In the next line of code, we can use that object, with a total guarantee of its existence. Even further, Angular provides services and objects of this type using a Singleton approach, so there's no other $scope at the same time.

Of course, the output is as expected, and every time we reload the page, the current time is updated:

Demo Dependency Injection

Current time: 6:38:52 PM

This philosophy is extended to any AngularJS object or service and even to those services that the user might want to create.

In Angular (modern versions), the architecture is similar, only that it uses TypeScript and the preferred way for DI is the constructor-injection model we saw in other demos, given that TypeScript is a totally object-oriented language.

In `Chapter 10`, *Dependency Injection in Other JavaScipt Frameworks*, we'll cover Dependency Injection in JavaScript (ES6, or ES 2015, to be precise), TypeScript and

WOW! eBook
www.wowebook.org

Angular 2, so you can have a wider sight of today's implementations of DI in modern web frameworks.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we have focused on the principles and advantages of Dependency Injection, and how to implement it in C# starting from a very simple application and modifying it to achieve decoupling of classes.

Once we saw the foundations of DI, we've covered briefly three programming aspects that influence the way these techniques are applied: Object Composition, Object Lifetime, and Interception, as a first approach that we'll cover in more detail in further chapters.

And, finally, we've seen how some popular IoC containers implement these concepts, together with a brief introduction to its use in other frameworks, like AngularJS.

In `Chapter 3`, *Introducing Dependency Injection in .NET Core 2.0*, we will begin the study of the most interesting features about Dependency Injection included in .NET Core.

WOW! eBook
www.wowebook.org

Introducing Dependency Injection in .NET Core 2.0

This chapter is a first approach in the implementation of .NET Core Dependency Injection technologies in the most recent version of .NET Core (2.0). Its main features, functionalities, and the namespaces holding the set of classes support these features.

We will talk about the following in this chapter:

- In all, we'll talk about how .NET Core includes support for several of the SOLID principles in general and those related to Dependency Injection in particular
- We'll start with the main characteristics of .NET Core and its installation and usage from Visual Studio, especially focusing on the latest version, Visual Studio 2017, together with the distinct types of deployment that this version allows
- Then, we'll get into the Dependency Injection in .NET Core, the `ActivatorUtilities` class, and the `Microsoft.Extensions.DependencyInjection` container
- Later on, we'll see some demos related to the implementation of an object's lifetime and how to map interfaces to instance classes, as well as a brief reminder of some more aspects of the Scope when applied to services
- Finally, we'll see how this DI functionality is also implemented inside some .NET Core services, especially those related to logging with plenty of demos showing how to use it in pure .NET Core

WOW! eBook
www.wowebook.org

The main characteristics of .NET Core

We've explained the basics of the architecture proposal of .NET Core in `Chapter 1`, *The SOLID Principles of Software Design* and how it pretends to be a game-changer, since it offers the possibility of creating code in the same language (C# or VB.NET), capable of executing on any device or platform.

Please note that VB.NET support of new features always goes behind the advances you'll see in the C# language, so, in case you try some new feature using this language, you should make sure that it has been implemented for the version we're using in this book.

This capability also extends to mobile applications, thanks to the incorporation of the Xamarin environment (and IDE) to the set of tools related to .NET Core development.

WOW! eBook
www.wowebook.org

The main advantages of .NET Core

If we look at this framework from a more developer-like point of view, we could say that what makes .NET Core different from other choices can be summarized in the following points:

- **Cross-platform:** This means execution on Windows, macOS and Linux, and also its portability to other operating systems. You can check the list of supported operating systems on various sites, such as https://github.com/dotnet/core/blob/master/roadmap.md, and you should bear in mind that CPUs and application scenarios will keep growing, whether they are provided by Microsoft or other companies.
- **Compatibility:** .NET Core is not only compatible with the .NET Framework, but also with Xamarin and Mono, thanks to the .NET Standard Library. As the official documentation states, the .NET Standard Library is,

"a formal specification of .NET APIs that are intended to be available on all .NET runtimes. The motivation behind the Standard Library is establishing greater uniformity in the .NET ecosystem. ECMA 335 continues to establish uniformity for .NET runtime behavior, but there is no similar spec for the .NET Base Class Libraries (BCL) for .NET library implementations."

- **Deployment:** Perhaps the most interesting feature about runtime is that it can be deployed inside an application or installed in a side-by-side user-or machine-wide basis
- **Distinct command-line options:** All distinct scenarios can be used at the command-line tools (and that's extensible to the rest of the platforms)
- **Open source:** The .NET Core platform is open source, since its inception. It uses MIT and Apache 2 licenses, and the documentation is licensed under Creative Commons 4.0 (CC-BY, see https://creativecommons.org/licenses/by/4.0/). Besides that, .NET Core is a .NET Foundation (http://www.dotnetfoundation.org/) project
- **Microsoft's support:** .NET Core is completely supported by Microsoft, and you'll find plenty of documentation, videos, forums, and so on in promotion sites that the company usually encourages as it is indicated in .NET Core Support (https://www.microsoft.com/net/core/support/)

WOW! eBook
www.wowebook.org

Installation of .NET Core in the IDE

In Chapter 1, *The SOLID Principles of Software Design*, we mentioned that you could use .NET Core (and ASP.NET Core) with any IDE of your choice. However, in this book, I'm using Visual Studio 2017 due to its integration tools and facilities, and its degree of optimization for .NET Core projects.

Nevertheless, and due to its youth, there's no common installation path for all versions of Visual Studio and, depending on the one you use, you'll find two different approaches.

WOW! eBook
www.wowebook.org

Installation path for .NET Core in Visual Studio 2015

If you want to use Visual Studio 2015, you should have Update 3.3 installed. It's available from this link: https://www.visualstudio.com/en-us/news/releasenotes/vs2015-update3-vs. In this site, you'll see how the update is related to NET Core 1.0.0 and .NET Core 1.0.0 SDK Preview 2.

If you're not sure about this version, just check the About Microsoft Visual Studio in the Help menu and make sure the version number is 14.0.25424.00 or higher, and includes Update 3.

You'll also need:

- The **NuGet Manager** extension for Visual Studio (you know, NuGet is the official package manager for Microsoft development, and we can be sure that it includes .NET Core in all its versions). You need NuGet 3.5.0-beta or higher to build .NET Core apps.
- **.NET Core Tooling Preview 2+**, which takes care of project templates and other tools for Visual Studio 2015, and you can find it at https://go.microsoft.com/fwlink/?LinkID=827546.

WOW! eBook
www.wowebook.org

.NET Core in Visual Studio 2017

The case for Visual Studio 2017 is quite different, since it takes care of the installation of prerequisites, as long as you have selected the .NET Core and Docker workload during the initial installation process (remember that, in V. Studio 2017, installation is made in a modular fashion, so, by default, it only installs a minimum set of features).

As it happens in the previous case, if you want to make sure about the installed version, just check in the Help/About Visual Studio menu, and see if you have version 15.0.26020.0 or higher.

WOW! eBook
www.wowebook.org

Types of deployment

We've mentioned how .NET Core allows using two distinct kinds of deployment for your apps--**Framework Dependent Deployment** (**FDD**), and **Self-Contained Deployment (SCD).** Which one to choose? That will depend on the target system and the amount of knowledge and management control that we might have about it.

Let's briefly remind ourselves about the main differences between these two types of deployment and its implications facing the management and structure of the assemblies resulted from any .NET Core compilation process:

- Framework Dependent relies on a shared version of .NET Core that should be installed on the target system. If this is the case, the application is portable between installations of .NET Core.
- Notice that, in this case, the app contains only its own code and any third-party dependencies that are outside of the .NET Core libraries. This is because FDDs contain `.dll` files that can be launched by means of the `dotnet` utility from the command line. Remember that if you launch, for example, the command `dotnetapplication1.dll`, that's enough to run an application named `application1.dll`.
- On the other side, SCD apps don't rely on any external code installed on the destination system. That is, all components, (and that includes both .NET Core libraries and the .NET Core runtime), are included in the final, installable package, and are isolated from other .NET Core applications.
- Consequently, SCDs include an executable (something like `app1.exe` on Windows platforms for an app of that name). This is a renamed version of the platform-specific .NET Core host, and a DLL file (such as `app.dll`), which is the actual application.
- So, you're deploying a concrete version of .NET Core as the executable together with your application, that will always adopt the form of a DLL, and it will run inside the context created by the executable.

As you can see, it is a very different approach, and perhaps, the first time that we can generate totally independent executable applications with .NET.

WOW! eBook
www.wowebook.org

Checking other dependencies in .NET Core

Before we keep on walking through the .NET Core features, it is wise to keep in mind that dependency is not only a matter of classes' relations but also about the components used to build applications, and the IDE might help us when defining and visually analyzing those dependencies even when the application is already compiled.

This is especially important in a framework where these components, are always downloaded from NuGet (or any other valid repository) and updated in our projects within Visual Studio 2017 in a dynamic manner.

Not taking care of these aspects, might lead to several problems. Among them, I'd like to highlight the following:

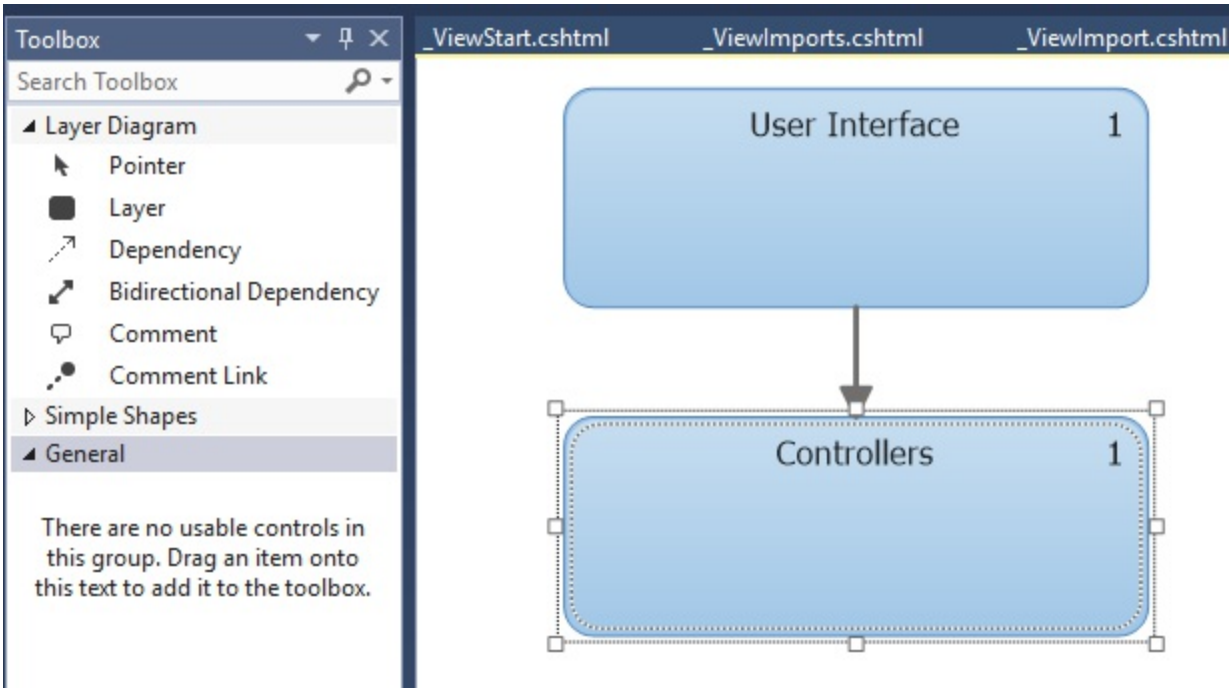
- Maintainability issues
- Security flaws
- License misuse
- Unwanted dependencies

In order to help developers to prevent dependency's issues, from version 2010 of Visual Studio, the IDE offers the ability to create layer diagrams, which have evolved continuously since that version.

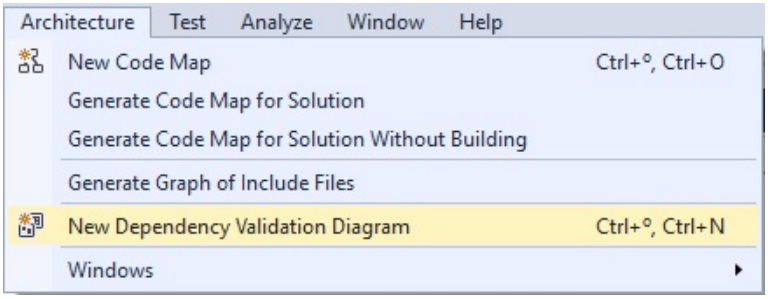
With these diagrams, you can express dependencies between layers, and these dependencies are not just shown through the diagrams hints but, starting in the latest version (2017), in the code itself.

When you design one of these diagrams, you can express dependencies using symbols provided by the designer that include single and bi-directional dependency, layers, and simple shapes, as you can see in the following screenshot:

WOW! eBook
www.wowebook.org



This diagram is created using the new menu option related to the Architecture menu in the IDE, which also presents some features related to code analysis, such as Code Map generation, the possibility to create graphs of those external files (graph of `Include Files`), and other features. In all, that option presents the following options:



Remember that this Architecture menu is only available in V.Studio 2017 Enterprise.

One of this menu's options is the new Dependency Validation diagram, which launches a new editing window, where we can drag and drop elements from the solution, including folders, files (C# and VB.NET) and even assemblies. We could consider these features as other implementations that the IDE offers to study any applications' dependencies.

This technique, has been rebuilt from scratch using Roslyn (refer my book *Mastering C# and .NET Framework* also by Pack, for more details and demos about this

WOW! eBook
www.wowebook.org

feature), and it allows the coder to configure the editor's behavior in a totally customized manner, programming how the IDE should respond when encountering any of these features in code (or even in other assemblies, since the tool accepts drag and drop already-compiled pieces too).

Once you establish relations and dependencies in the diagram, this new Intellisense will be able to validate the existing code, advising the programmer about the conflicting areas inside different application domains.

Actually, you can activate these features in the configuration in a such a way, that, as soon as a problem is detected, you will see a squiggle highlighting your offending code, together with an indication of the root of the problem.

For instance, we can establish which classes are allowed to live in every application's namespace. If a single class is moved to a namespace forbidden by the dependency diagram, a hint will be presented in the code editor itself and, also, in the Error List window (see the following screenshot):

WOW! eBook
www.wowebook.org

LegacyExample2015 - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Architecture Test Load Test Analyze Sign in
Window Help

Debug | Any CPU | Start

CustomerManager.cs | DataUtilities.cs

DataLogic | DataLogic.DataUtilities | GetCustomerDisplayDetails(Customer

```
using MyCorp.Data;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DataLogic
{
    public
    {
        public static string GetCustomerDisplayDetails(Customer customer)
        {
            return $"Id: {customer.Id}, Name: {customer.Name}";
        }
    }
}
```

namespace DataLogic

Layer 'Data' requires that its associated artifacts must belong to one of the allowed namespaces: MyCorp.D

Error List

Entire Solution | 2 Errors | 0 of 2 Warnings | 0 Messages | Build + IntelliSense

Search Error List

Code	Description	Project	File	Li...	Suppression State
DV1003	Layer 'Business' requires that associated artifacts cannot belong to namespace 'BusinessForbidden1'	BusinessLogic	CustomerManager.cs	22	Active
DV1001	Layer 'Data' requires that its associated artifacts must belong to one of the allowed namespaces: MyCorp.Data	DataLogic	DataUtilities.cs	8	Active

Ln 8 Col 11 Ch 11 INS Publish

Of course, this behavior is also configurable, and these features are independent from other options we find in the Analyze menu in relation with Code Metrics, Performance Profilers, and so on.

WOW! eBook
www.wowebook.org

Dependency Injection in .NET Core

When dealing with the proper features related to Dependency Injection in .NET Core, there are a bunch of previous approaches to consider. One of those is the mantra `new` is glue, which we often hear as an advice.

This means that whenever you create a new instance of a class (you use the `new` keyword), there's some code cohesion behind the scenes. You're establishing that the class that defines that instance is going to have a dependency on the instantiated class.

We've seen how to get rid of that problem via factories, or using third-party libraries focused on the DI problem and how they can be used, but, in this chapter, we're going to rely on those features provided by the framework itself to achieve the same results.

Ideally, that's something that we could define and, once the definition is registered, every time we need an instance of any of the predefined classes, something else should take care of providing that instance.

In other words, the behavior we've seen related to other DI containers should (more or less) be present here as well, and cover the most important features that those veteran containers offer, including an object's lifetime, registration, and definition of generic classes and interfaces, and so on.

WOW! eBook
www.wowebook.org

DI Architecture and Object's Lifetime

Considering the preceding points, and even knowing that we're dealing with an initial version of the frameworks, the .NET Core team designed Dependency Injection features based on two ideas:

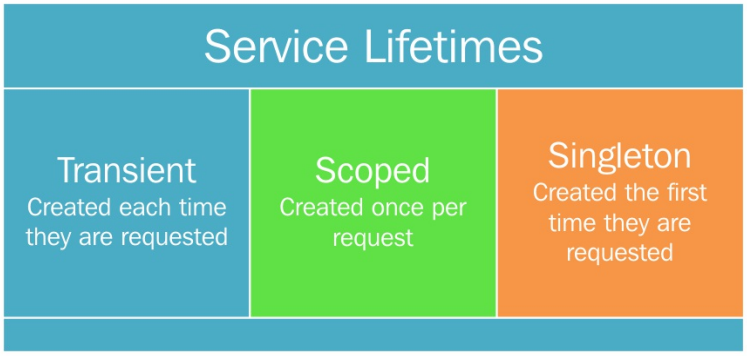
- On one side, there was already existing functionality, that could be perfectly extended to work inside .NET Core
- On the other side, they thought that it would be wise to include the most used features present in other initiatives, (either Microsoft's or external) like those present in Prism Patterns & Practices or third party DI Containers (remember the four containers we saw in previous chapters)

Consequently (following these ideas), when dealing with an object's lifetime, .NET Core presents three types, depending on the way instances will be configured and used--Singleton, Scoped, and Transient.

- These options do not only have influence in the way we define them, but also in the way we can use them, and in some cases, thread safe precautions, and other aspects should be considered.
- In the Singleton option, a unique instance of the object is used (meaning that we only manage a reference to the object). Any further request uses the same instance, according to the singleton pattern.
- The Scoped version is local to the context in which it is used and doesn't extend beyond that context.
- Finally, the Transient option creates a new instance of the class every time is requested.

Although included in the ASP.NET Core documentation, Microsoft provides the following diagram in relation to these options:

WOW! eBook
www.wowebook.org



Since, in the real world, this functionality is mainly linked to ASP.NET Core apps, services and middleware show up with functionality that helps in the task (that's why those references are to the *request*). But it actually works equally well in pure .NET Core applications.

As we mentioned at the beginning, a big part of this functionality is related to the `Microsoft.Extensions.DependencyInjection` namespace, included in a DLL of the same name, which can work in conjunction with another helper DLLs such as `Microsoft.Extensions.DependencyInjection.Abstractions` and others.

Concretely, the `IServiceCollection` interface is the base of a set of classes that provide methods to instantiate objects (usually called services in this context) via DI using these three options. We'll see this in action in the next sections, but also in `Chapter 4`, *Dependency Injection in ASP.NET Core,* dedicated to ASP.NET Core.

WOW! eBook
www.wowebook.org

The class ActivatorUtilities and other helpers

Before we get into the demos, keep in mind that the number of classes included in this namespace is pretty large, since it pretends to offer a wide coverage while still being multi-platform.

One of these cases is the class `ActivatorUtilities` class, also included in the `Microsoft.Extensions.DependencyInjection` library, which contains static methods to help in the configuration and implementation of services, instances, and factories that ease DI management and control.

Thus, in case you find yourself missing some feature or functionality, take a look at the documentation at `http://docs.microsoft.com`, but, keep in mind that you'll find that it is related to ASP.NET Core.

This means that most of the implementations that you will see (on the internet and other sources) are not going to be linked to .NET Core apps, but to ASP.NET Core apps, in which, much of this functionality is implemented by default for different scenarios.

Obviously, there's much more inside those classes that we cannot cover in a single chapter.

To give you an idea of the possibilities linked to this class, I'm including a summary of its functionality presented in the official documentation, reminding you of each method and its main purpose:

Methods summary	**Description**
`CreateFactory(Type, Type[])`	Create a delegate that will instantiate a type with constructor arguments provided directly and/or from a `System.IServiceProvider`.

WOW! eBook
www.wowebook.org

`CreateInstance(IServiceProvider, Type, Object[])`	Instantiate a type with constructor arguments provided directly and/or from a `System.IServiceProvider`.
<p>`CreateInstance<T>(IServiceProvider, Object[])`	Instantiate a type with constructor arguments provided directly and/or from a `System.IServiceProvider`.
`GetServiceOrCreateInstance(IServiceProvider, Type)`	Retrieve an instance of the given type from the service provider. If one is not found, then instantiate it directly.
`GetServiceOrCreateInstance<T>(IServiceProvider)`	Retrieve an instance of the given type from the service provider. If one is not found, then instantiate it directly.

It's now time to start working with the main classes related to DI inside the **`Microsoft.Extensions.DependencyInjection`** and **`Microsoft.Extensions.DependencyInjection.Abstractions`** libraries.

WOW! eBook
www.wowebook.org

The Microsoft.Extensions.DependencyIn container

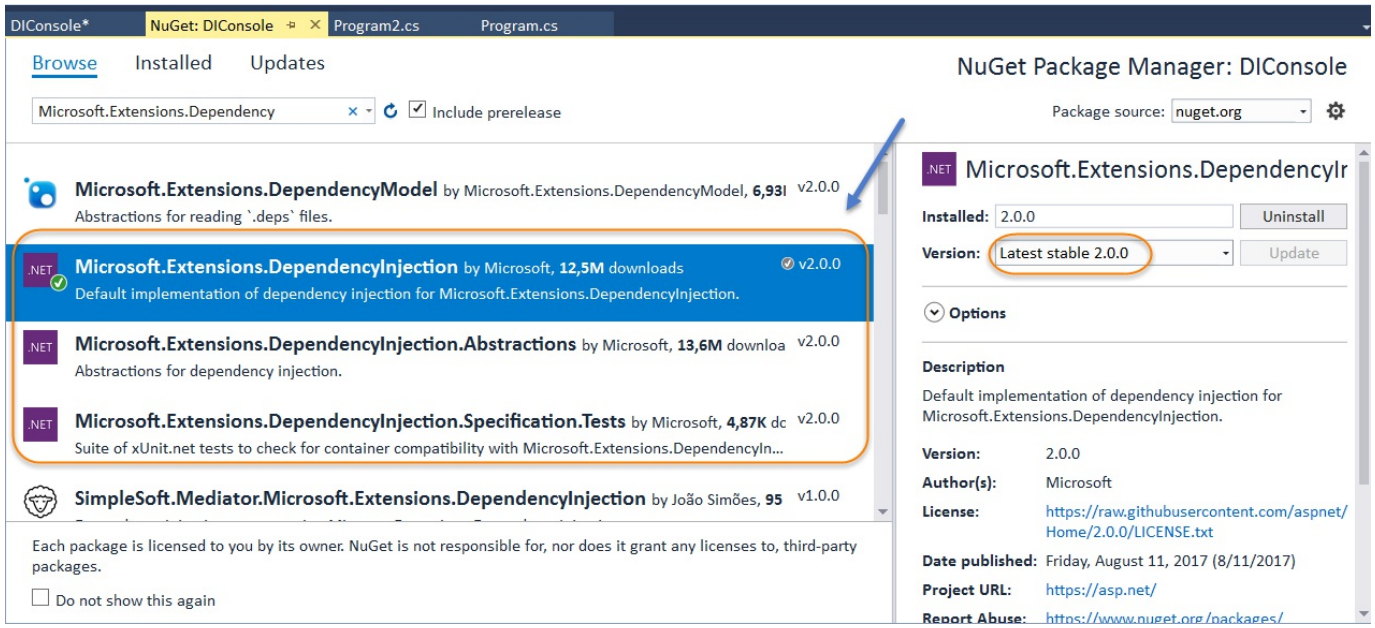
Most of the samples that you'll see on the internet about Dependency Injection and .NET Core will reference to ASP.NET Core applications since it comes by default when you use the predefined template inside Visual Studio.

However, the use of ASP.NET Core applications is not mandatory if you want to see how DI works from scratch. That being said, we will have to configure the registering of our services manually. This implies certain extra aspects such as severity levels, an object's lifetime, and disposable features if we want this architecture to work seamlessly.

Since .NET Core offers its own container linked to the external library `Microsoft.Extensions.DependencyInjection` (now, in version 2.0), we'll have to reference it via NuGet packages.

For a first demo, we can create a simple .NET Core Console application, reference the library using the NuGet package plug-in, and look for its name online. You'll observe that when you filter for that name, the NuGet Package Manager, presents several versions of the library (see the following screenshot):

WOW! eBook
www.wowebook.org

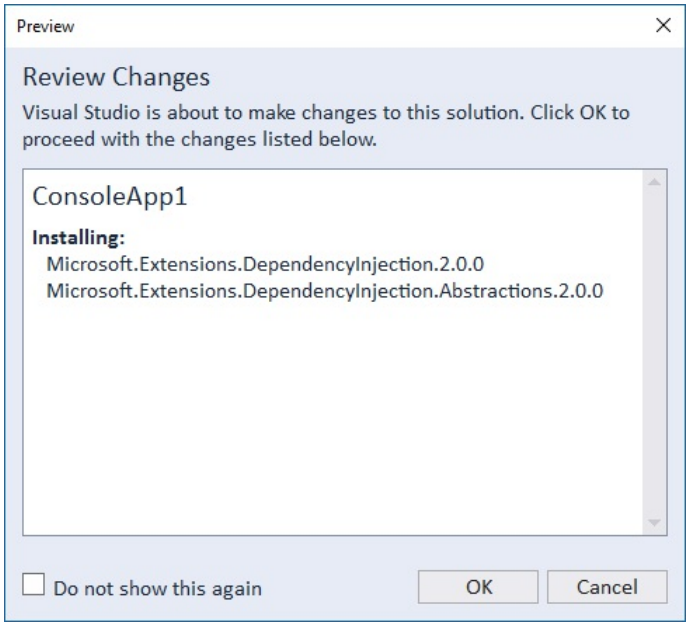


As you see, other libraries are also declared as complementary (`Microsoft.Extensions.DependencyInjection.Abstractions`, and `Microsoft.Extensions.DependencyInjection.Specification.Tests`). The first one (the `Abstractions`) should also appear as referenced inside our project once the installation is completed. But for now, let's focus on the main library.

At this moment, don't be scared of the following dialog box. It will tell you that a whole bunch of libraries is going to be updated, and another set of libraries will be installed anew. The reason for this might be double--on one side, the use of the latest versions of this library implies updating other previously referenced libraries according to an internal library dependency manager that Visual Studio 2017 uses.
On the other side, installation of some new libraries, like this one, might imply other dependencies, so we end up with two distinct areas of updates (go to the Dependencies section in the Solution Explorer after installation).

Visual Studio 2017 will offer a list of every library being updated and if you scroll down a bit, also the list of all the dependent libraries:

WOW! eBook
www.wowebook.org



After this dialog box, you'll be presented another one with the License Acceptance of each one of the libraries. Once completed, another entry in your Dependencies section of the Solution Explorer will show up, pointing to NuGet. That's where you'll find all that new stuff.

A further look at those new entries will reveal all the main and dependent libraries linked to that DependencyInjection reference, which might be a few.

Since, initially, we just want to make a test of how this namespace works, we'll start testing it in a very simple manner (Legit is to say that I was inspired by Jurgen Gustch's blog's entry Using Dependency Injection in .NET Core Console Apps, because, for this initial approach, I found it especially suitable and explanatory).

So, I've modified those ideas to create a couple of classes, in which the first one happens to have dependencies on the other (that is, the first class references the second).

Just for the matter of completeness, I included an extra method to check the right moment in which it was invoked, and also implemented the `IDisposable` interface to be able to convey information about disposal by the Garbage Collector (more about that later).

So, I ended with this code (outside of the class `Program` class that the template creates by default):

```
public class DependencyClass1 : IDisposable
```

WOW! eBook
www.wowebook.org

```
{
  private readonly DependencyClass2 _DC2;
  public DependencyClass1(DependencyClass2 DC2instance)
  {
    _DC2 = DC2instance;
    Console.WriteLine("Constructor of DependencyClass1 finished");
  }
  public void CurrentTime()
  {
    string time = DateTime.Now.Hour.ToString() + ":" +
       DateTime.Now.Minute.ToString() + ":" +
       DateTime.Now.Second.ToString();
    Console.WriteLine($"Current time: {time}");
  }
  public void Dispose()
  {
    _DC2.Dispose();
    Console.WriteLine("DependencyClass1 disposed");
  }
}
public class DependencyClass2 : IDisposable
{
  public DependencyClass2()
  {
    Console.WriteLine("Constructor of DependencyClass2 finished");
  }

  public void Dispose()
  {
    Console.WriteLine("DependencyClass2 Disposed");
  }
}
```

Notice also that `DependencyClass1` takes care of disposing `DependencyClass2`, once it's done with it.

Now it comes when the `DependencyInjection` classes are needed. First, we make a reference to the corresponding namespace on top of our code (only two namespaces are required for this simple demo):

```
using Microsoft.Extensions.DependencyInjection;
using System;
```

Then, inside the `Program` class, we need to register and use these classes. The first step is performed using a new instance of the `ServiceCollection` class.

Inside that class, we find methods to register all required services in a way that at the same time, configures the object's lifetime mentioned previously, as you can see in the following screenshot:

WOW! eBook
www.wowebook.org

```
var services = new ServiceCollection();
services.AddTransient<DependencyClass2>();
services.
var provi
using (va
{
    // Me
```

AddScoped
AddScoped<>
AddSingleton
AddSingleton<>
AddTransient
AddTransient<>
BuildServiceProvider

(extension) IServiceCollection IServiceCollection.AddScoped(Type serviceType, Type implementatio
Adds a scoped service of the type specified in serviceType with an implementation of the type speci

```
uildServiceProvider();
s1 = provider.GetService<DependencyClass1>())

g DependencyClass1
```

As we will see further on, we're provided with two versions of every method--generic and a non-generic. This gives us more flexibility to integrate different classes and services into our DI architecture.

For this initial version, we're using the `AddTransient` generic version of the methods, which will return new instances of every class, every time we make a reference in the code.

Once our classes are registered, a provider has to be built with that definition. That's something we perform with a call to `BuildServiceProvider()` on the `ServicesCollection` class just created (we'll go back to service providers later on). But, for now, suffice to say that an instance of the `IServiceProvider` interface will be created and configured to manage any request for one of the classes previously registered using the `Add*` method.

Another thing to point out is that the way objects are instantiated follows the lazy creation pattern that you, reader, which you might already know from other contexts, such as LINQ. That means until the first instance of a DI Object is requested, nothing gets created.

So, even if we don't do anything (useful) with these classes, once the `DependencyClass1` is requested, the entire dependency injection mechanism is started.

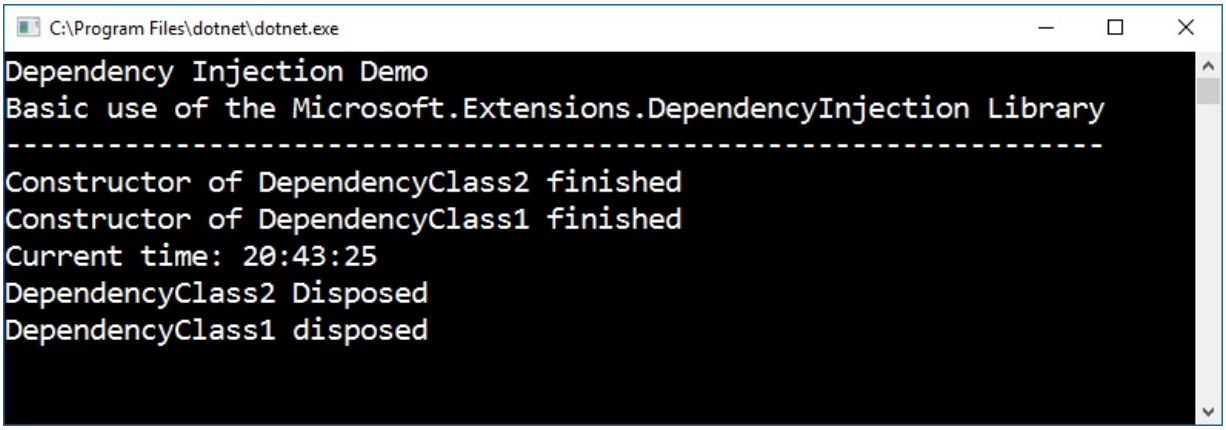
The following code is modified in the class `Program` class to configure this architecture:

```
static void Main(string[] args)
{
  Console.WriteLine("Dependency Injection Demo");
  Console.WriteLine("Basic use of the Microsoft.Extensions.
      DependencyInjection Library");
  Console.WriteLine("-------------------------------------
      ---------------------------");
  var services = new ServiceCollection();
  services.AddTransient<DependencyClass2>();
```

WOW! eBook
www.wowebook.org

```
      services.AddTransient<DependencyClass1>();
      var provider = services.BuildServiceProvider();
      using (var DC1Instance = provider.GetService<DependencyClass1>())
      {
        // Merely by declaring DC1Instance
        // everything gets launched, but we also call
        // CurrentTime() just to check functionality
        DC1Instance.CurrentTime();
        // Notice also how classes are properly disposed
        // after used.
      }
      Console.ReadLine();
    }
```

As you can see in the following output, everything works as expected:



The code shows how whenever we need an instance of a class, we issue a call to the generic `GetService()` method and we can start using it in the next sentence.

Another aspect to notice is that the reference to `DependencyClass2` goes first, and so it does its disposal. Remember that the constructor of `DependencyClass1` receives an instance of `DependencyClass2`, so, it doesn't finish until the latter is totally created.

Also, before disposing of the first class, we call on `Dispose` of the second, that's why the order is inverted.

WOW! eBook
www.wowebook.org

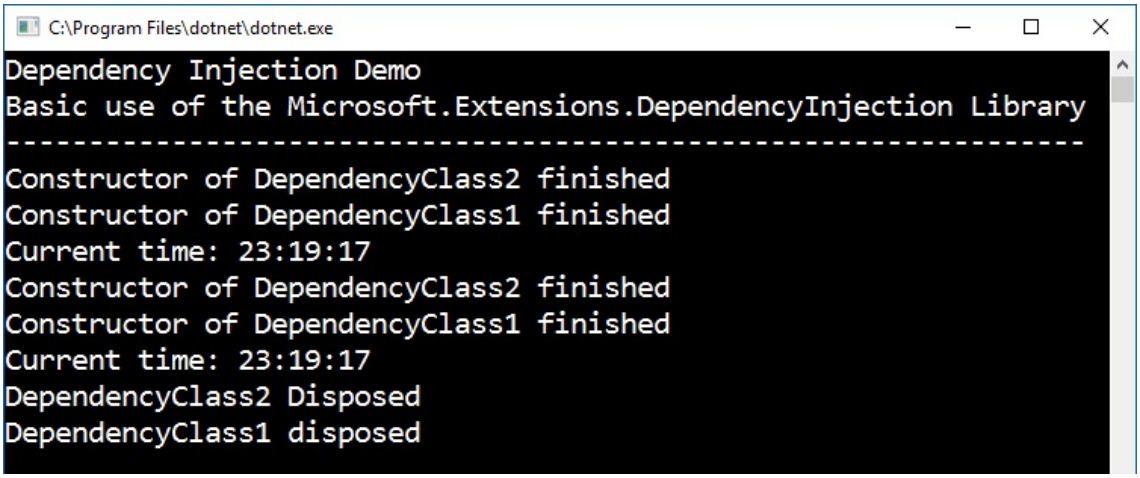
Checking the object's lifetime

Another important aspect of this architecture that we should consider carefully, is the way our instances are obtained from the point of view of its lifetime. Let's see the differences in this very demo, adding references and changing the way they are registered into the `IServiceProvider` class.

What happens if we create another instance of the first class? As expected, when we change the `using` block of code to include another instance of `DependencyClass1`, such as in the following code:

```
using (var DC1Instance = provider.
     GetService<DependencyClass1>())
{
  // Merely by declaring DependencyClass1
  // everything gets launched, but we also call
  // CurrentTime() just to check functionality
  DC1Instance.CurrentTime();
  // Notice also how classes are properly disposed
  // after used.
  var DC1Instance2 = provider.GetService<DependencyClass1>();
  DC1Instance2.CurrentTime();
}
```

The output changes noticeably since we're forcing the engine to create a new instance, instead of reusing the previous one:



As seen in the preceding screenshot, we're making the DI engine call the constructor twice, because we're using the `AddTransient()` version of the `services` configuration object.

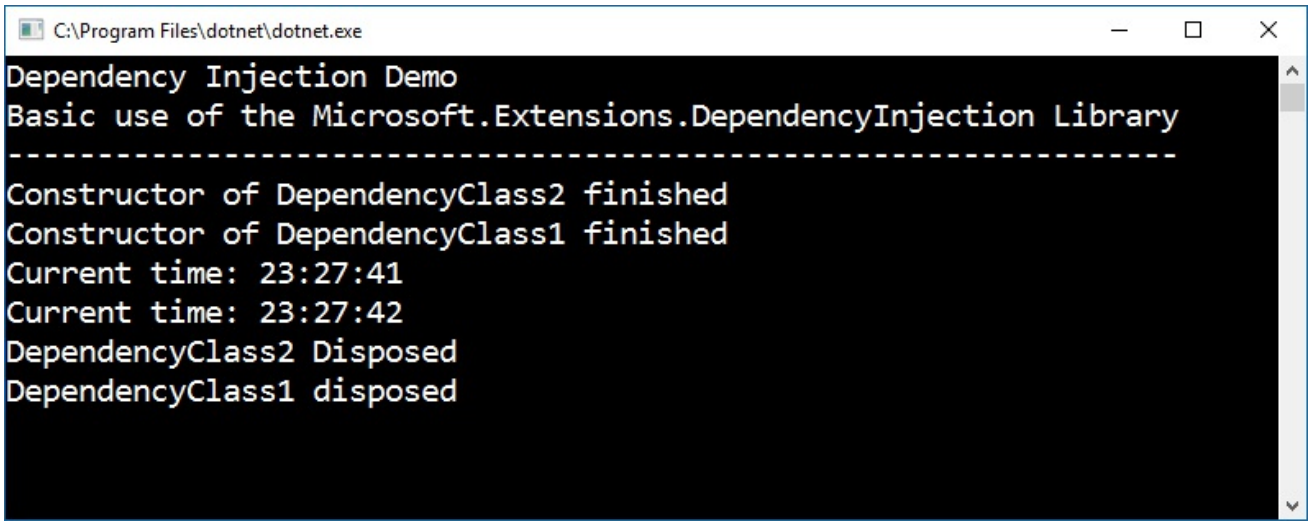
WOW! eBook
www.wowebook.org

However, in this scenario, if we change the registration method for `AddScoped` of `AddSingleton`, we would be reusing the same instance of the objects, and therefore, saving memory and resources.

For instance, by just changing those two lines of code in this manner:

```
services.AddScoped<DependencyClass2>();
services.AddScoped<DependencyClass1>();
```

We can check the different creation behavior with a simple look at the corresponding output:



As you can see, the `CurrentTime` function is still called twice, but the number of instances in use is only one.

And, in this case, we would get exactly the same output using the `AddSingleton<>()` methods, since, in this scenario, there is a coincidence and no more than one instance is going to be used.

Another interesting aspect of the functionality provided by this service is that being a generic collection itself, it is possible to add/remove/clear instances of services at runtime, so we can always have total control of what it is inside the collection and the order in which is defined and instantiated.

To that purpose, we find methods such as `Clear()`, `Contains()`, `IndexOf()`, `Insert()`, `InsertAt()`, `Remove` and `RemoveAt()`, just like we would find in any other generic collection.

WOW! eBook
www.wowebook.org

Mapping interfaces to instance classes

The previous demo is simple enough to understand the basics of DI inside the `Microsoft.Extensions.DependencyInjection` library, but, most likely, in a real application, you will have some interfaces defined and a series of classes that implement those interfaces.

In such a case, it is more convenient to be able to map interfaces to classes that implement them, so you just have to ask for an implementation of the corresponding interface, thinking on the functionality the interface provides (the business problem to be solved), rather than in the concrete class that implements it.

Another advantage is that we're able to change the definitions at runtime (remember, it's a collection where we can add/remove items), so it is perfectly possible to redefine any previous mapping to a new one, according to our applications' needs.

As in the preceding demo, we're using a very simple approach to see this in action. I've created two interfaces, and two classes that implement them, with a single method that writes a basic message to the `Console`. This is the initial code:

```
public interface IXMLWriter
{
  void WriteXML();
}
public interface IJSONWriter
{
  void WriteJSON();
}
public class XMLWriter : IXMLWriter
{
  public void WriteXML()
  {
    Console.WriteLine("<message>Writing in XML Format</message>");
  }
}
public class JSONWriter : IJSONWriter
{
  public void WriteJSON()
  {
    Console.WriteLine("{'message': 'Writing in JSON Format'}");
  }
}
```

WOW! eBook
www.wowebook.org

There is a correspondence between classes and interfaces, so we can now refer to the interface and let the DI engine decide which class instance is returned to us. This is quite similar to the demos we saw in the previous chapter when using third-party DI containers.

With that purpose, the `ServiceCollection` class supports an alternative way of defining references in which you can pass (generic signature) an interface name and the class that maps such interface when a `GetService<Interface>()` is called.

At the time of requesting one of these implementations, we will ask for an instance of the predefined interface, rather than a concrete class.

Note that we also have an alternative way to ask for a service by means of the `GetRequiredService<Interface>()` method, that throws an exception if the service type isn't registered.

Assuming this change, the implementation of the demo is quite simple:

```
static void Main(string[] args)
{
  var services = new ServiceCollection();
  services.AddTransient<IXMLWriter, XMLWriter>();
  services.AddTransient<IJSONWriter, JSONWriter>();
  var provider = services.BuildServiceProvider();
  Console.WriteLine("Dependency Injection Demo (2)");
  Console.WriteLine("Mapping Interfaces to instance classes");
  Console.WriteLine("-------------------------------------");
  Console.WriteLine("Please, select message format
      (1):XML // (2):JSON");
   var res = Console.ReadLine();
  if (res == "1")
  {
    var XMLInstance = provider.GetService<IXMLWriter>();
    XMLInstance.WriteXML();
  }
  else
  {
     var JSONInstance = provider.GetService<IJSONWriter>();
     JSONInstance.WriteJSON();
  }
  Console.ReadLine();
}
```

Unlike the previous case, we don't ask for a given class, but the class implements the required interface.

The output is, again, as expected (see the following screenshot):

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
Dependency Injection Demo (2)
Mapping Interfaces to instance classes
-------------------------------------
Please, select message format (1):XML // (2):JSON
1
<message>Writing in XML Format</message>
```

Notice that the registration works in a similar way to what we already saw in other DI containers. That is, if we use this syntax, that latest registered mapping is the one returned, although that can be changed dynamically.

Another very useful feature is the `GetServices<Interface>` method, since it allows us to recover all registered services and invoke them at will.

We can prove this by adding a couple of new classes that implement the same interfaces and register them together with the previous ones:

```
public class XMLWriter2 : IXMLWriter
{
  public void WriteXML()
  {
    Console.WriteLine("<message>Writing in XML Format (2)</message>");
  }
}
public class JSONWriter2 : IJSONWriter
{
  public void WriteJSON()
  {
    Console.WriteLine("{'message': 'Writing in JSON Format (2)'}");
  }
}
```

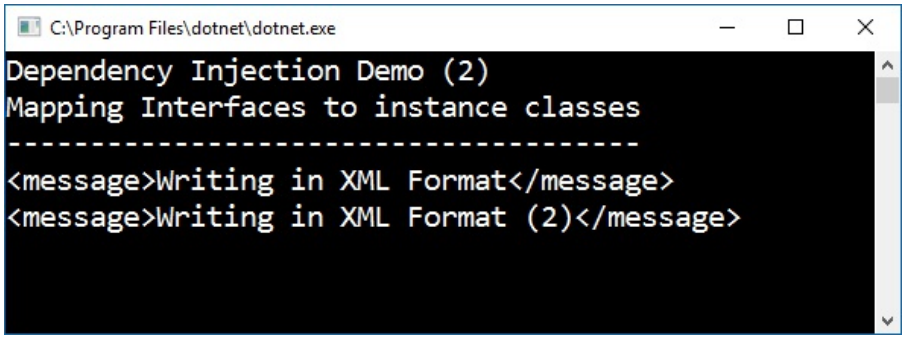
After these definitions, we register both classes under the same Interface contracts, so they can be accessed altogether:

```
services.AddTransient<IXMLWriter, XMLWriter>();
services.AddTransient<IXMLWriter, XMLWriter2>();
services.AddTransient<IJSONWriter, JSONWriter>();
services.AddTransient<IJSONWriter, JSONWriter2>();
Now we can use a whole collection by asking for it by means of the
   GetServices<Interface>() method that I mentioned above:
var registeredXMLServices = provider.GetServices<IXMLWriter>();
foreach (var svc in registeredXMLServices)
{
  svc.WriteXML();
}
```

WOW! eBook
www.wowebook.org

Since we're using the defined interface functionality, we know all of them will implement the `WriteXML()` function, even if they do it differently.

You can appreciate the different calls in the corresponding output:



Another alternative way to get the whole list of services is, of course, from the `services` collection itself. To perform this, we need another helper class also provided by the `Microsoft.Extensions.DependencyInjection` library.

In this case, the procedure is to create a `ServiceDescriptor` collection that holds the information of all services in our `ServiceCollection` class. We use an enumerator and the `CopyTo()` method (which expects a `ServiceCollection` as the first argument) to create such a collection:

```
var myServiceArray = new ServiceDescriptor[services.Count];
// Copy the services into an array.
services.CopyTo(myServiceArray, 0);
IEnumerator myEnumerator = myServiceArray.GetEnumerator();
Console.WriteLine("The Implementation Types in the array are");
while (myEnumerator.MoveNext())
{
  var myService1 = (ServiceDescriptor)myEnumerator.Current;
  Console.WriteLine(myService1.ImplementationType);
}
```

When the collection is copied to the `ServiceDescriptor` collection, we can see at least five interesting properties that might, later on, be used to determine the service we need in a given scenario:

WOW! eBook
www.wowebook.org

```
Console.WriteLine(myService1.);
sole.ReadLine();
rface IXMLWriter
s
```

Equals
GetHashCode
GetType
ImplementationFactory
ImplementationInstance
ImplementationType
Lifetime
ServiceType
ToString

Type ServiceDescriptor.ImplementationType { get; }

Notice that here we're asking for the `ImplementationType` property to obtain all defined types:

```
C:\Program Files\dotnet\dotnet.exe
Dependency Injection Demo (2)
Mapping Interfaces to instance classes
-------------------------------------
The Implementation Types in the collection are:
DIConsole2.XMLWriter
DIConsole2.XMLWriter2
DIConsole2.JSONWriter
DIConsole2.JSONWriter2
```

This gives us a clue on how to select a single service independently of its position in the collection. Another helper method linked to this functionality is the simple `Contains()` that asks for a `ServiceDescriptor` object as its unique argument.

Another easy way to get information about the currently registered components in our services container is by iterating through it directly, using a simple `foreach` loop:

```
//Description of properties in the service collection
foreach (var svc in services)
{
  Console.WriteLine($"Type: {svc.ImplementationType} \n" +
     $"Lifetime: {svc.Lifetime} \n" +
     $"Service Type: {svc.ServiceType}");
}
```

Notice that depending on the way the services are registered and other programming features, not all the properties will have a value (in this case, it only makes sense asking for the interfaces (`ServiceType`), the implementations (`Types`) and their lifetimes).

This is also useful when we have more than a class that implements the same interface, because we can decide which one we need based on these values:

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
Dependency Injection Demo (2)
Mapping Interfaces to instance classes
-------------------------------------
Type: DIConsole2.XMLWriter
Lifetime: Transient
Service Type: DIConsole2.IXMLWriter
Type: DIConsole2.XMLWriter2
Lifetime: Transient
Service Type: DIConsole2.IXMLWriter
Type: DIConsole2.JSONWriter
Lifetime: Transient
Service Type: DIConsole2.IJSONWriter
Type: DIConsole2.JSONWriter2
Lifetime: Transient
Service Type: DIConsole2.IJSONWriter
```

Additionally, it's possible to use some helper classes linked to the namespaces here implied, to perform registrations `a posteriori`. For instance, the `ServiceProviderServiceExtensions` class contains a static method capable of obtaining a given `ServiceType` linked to a certain provider.

In other words, you can get an instance of a registered service without using the `ServiceCollection` that registered it, as long as you can pass the corresponding provider as an argument.

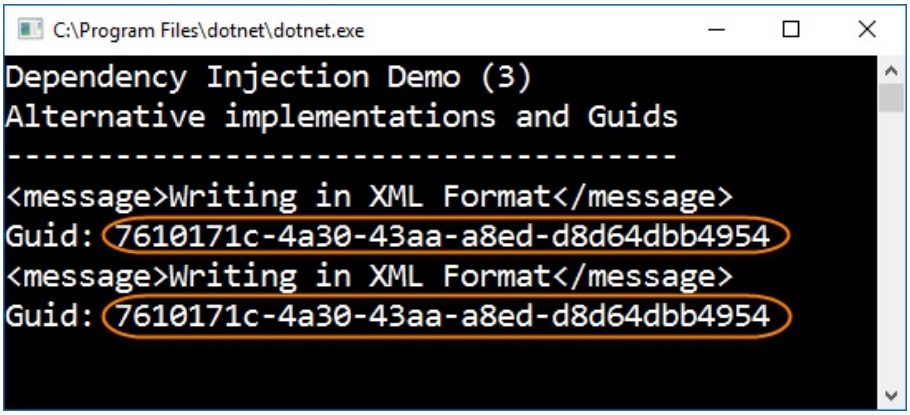
I've created a variant of the previous demo, this time including a read-only property in each of the `*writer` classes to hold a unique identifier (a `GUID`), so it is easy to determine if we're using the same or another instance of the service.

Consider the following code (a variant of the previous demo):

```
static void Main(string[] args)
{
  var services = new ServiceCollection();
  services.AddSingleton<IXMLWriter, XMLWriter>();
  var provider = services.BuildServiceProvider();
  Console.WriteLine("Dependency Injection Demo (3)");
  Console.WriteLine("Choice between implementations");
  Console.WriteLine("------------------------------");
  // Instance via services class
  var XMLInstance = provider.GetService<IXMLWriter>();
  XMLInstance.WriteXML();
  // Instance via ServiceProviderServiceExtensions
  var XMLInstance2 = ServiceProviderServiceExtensions.
                     GetService<IXMLWriter>(provider);
  XMLInstance2.WriteXML();
  Console.ReadLine();
}
```

WOW! eBook
www.wowebook.org

As you can see, we're using two distinct approaches to get the same instance (identified by its `GUID`). We can test it by comparing both outputs (see the following screenshot):



This could be especially useful in situations in which, for whatever reason, a call to services is not suitable or convenient.

Providers can also be obtained via helper classes, besides the `BuildServiceProvider` method of `ServiceCollection`. To do so, we can use the `CreateDefaultServiceProvider` class, which also has instance methods to create, either a provider or a `Builder`.

For the time being, those two are the only possibilities available to that class, but it can also be quite useful in certain scenarios where we prefer the service collection not to be used:

The following code has the provider created via the `DefaultServiceProvider` class:

```
var services = new ServiceCollection();
services.AddSingleton<IXMLWriter, XMLWriter>();
// Provider via DefaultServiceProviderFactory
var factory = new DefaultServiceProviderFactory();
IServiceProvider prov = factory.CreateServiceProvider(services);
var XMLInstance = prov.GetService<IXMLWriter>();
XMLInstance.WriteXML();
```

I omit the output here, since it's exactly the same as in the previous demos, and you

WOW! eBook
www.wowebook.org

can check it yourself within the code accompanying this chapter.

This is not the only way in which we can obtain a service provider. Yes, there's another one, linked to the static method `BuildServiceProvider` of the `ServiceCollectionContainerBuilderExtensions` class.

In this case, programming it is even easier, since we don't need any instance of the class, and the code gets reduced to the following code:

```
var services = new ServiceCollection();
services.AddSingleton<IXMLWriter, XMLWriter>();
// Provider via ServiceCollectionContainerBuilderExtensions
IServiceProvider prov = ServiceCollectionContainerBuilderExtensions.
    BuildServiceProvider(services);
var XMLInstance = prov.GetService<IXMLWriter>();
XMLInstance.WriteXML();
```

Just to get exactly the same results as we did before (again, I omit the output).

WOW! eBook
www.wowebook.org

The concept of Scope Applied to services

When dealing with services and other DI-related functionality, an important point to define is its scope. DI documentation defines the scope of a service closely related to its lifetime, and therefore, with the moment in which that service should be disposed of by the Garbage Collector.

We have talked about the Transient and Singleton lifetimes previously, but the Scope lifetime is, indeed, a bit more confusing.

Concretely, the `IServiceDispose` interface, which implements the `IDispose` interface, holds the method `Disposed of()`, method which, when called, ends the scope lifetime. It is included inside the `Microsoft.Extensions.DependencyInjection.Abstractions.dll`.

In more detail, the documentation states: "*Once this is disposed of, any scoped services that have been resolved from the Microsoft.Extensions.DependencyInjection.IServiceScope.ServiceProvider will also be disposed of*".

Its declaration is as follows:

```
public interface IServiceScope : IDisposable
```

If you remember in the first demos of this chapter, our `DependencyClass1` and `DependencyClass2` classes implemented the `IDisposable` interface, so we could call those methods after the classes' main operations were finished.

As we'll see in `Chapter 4`, *Dependency Injection in ASP.NET Core,* this concept adapts especially well to some internet applications' scenarios in which it makes a lot of sense to have a specific control for the lifetime of some services, and always have access to the execution context in the form of an instance class.

In `Chapter 4`, *Dependency Injection in ASP.NET Core*, you'll see that this feature can be important to face performance issues, resources of the server, scalability issues, and so on.

WOW! eBook
www.wowebook.org

Other extensions with DI functionality

Linked to the `Microsoft.Extensions` global namespace, we find related namespaces that are becoming popular among developers, because they help in different areas of the application's lifecycle.

Two of the most used are `Microsoft.Extensions.Logging` and `Microsoft.Extensions.Logging.Console`, which you can use to configure and code logging services using the techniques of Dependency Injection that we've seen in this chapter. They offer functionality similar to other popular frameworks, such as Serilog, Log4Net or NLog.

I'm talking about `ILoggerFactory` and `ILogger<T>`, mainly used (especially in ASP.NET Core apps) to emit information at runtime, with the ability to redirect that information to different targets (console, output window, and so on).

But they can also be used to monitor and debug applications in .NET Core, and, even not offering the whole set of resources we find in those, more specialized, frameworks, they can be, many times, enough to cover our needs.

WOW! eBook
www.wowebook.org

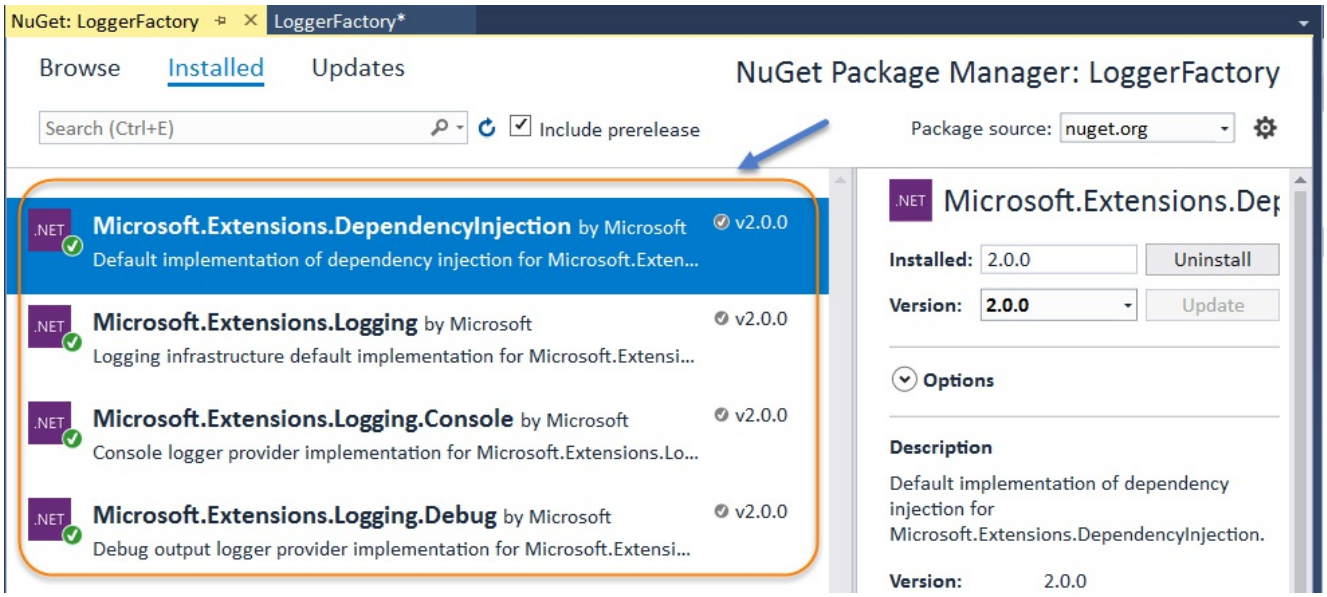
A reflection on the structure of .NET Core

It's quite interesting to see how .NET Core creators have designed this technology based on the Dependency Injection principles. That's why, besides its value as a tool for the developer, we can see the extra value of seeing DI really implemented by default in .NET Core libraries.

Both classes are available inside the `Microsoft.Extensions.Logging` namespace, but other complementary namespaces, such as `Microsoft.Extensions.Logging.Console` and `Microsoft.Extensions.Logging.Debug`, also permit extended logging functionality that we'll use in the next demo. They can be easily referenced via NuGet packages, as we did before.

To keep on demoing things with the simplest possible code, I'll use a simplified version of the previous demo, only this time using a single external class `XMLWriter` and its corresponding interface, with slight changes.

Just like with the `DependencyInjection` namespace, `Microsoft.Extensions.Logging` appears next to `Configuration` and `Dependency` libraries when filtering for this criterion in the Search box (notice that we'll need a total of four extra libraries, including `Logging.Console` and `Logging.Debug`):



WOW! eBook
www.wowebook.org

After installing those libraries, if you're curious about these API's possibilities, you can also inspect the references that will show up in the `Solution Explorer`, next to the NuGet references' entry.

The functionality provided by these logging services include the ability to write to the `Console`, using distinct types of messages (depending on the nature of the output: debug, information, warning, or error) and using several colors and formats to express these categories.

Mark Michaelis clearly explains in his MSDN article *Essential .NET - Dependency Injection with .NET Core* some of the advantages of these DI implementations that we find linked to .NET Core.

He points out that, when you want to switch between different implementations of some service and avoid hard-coding a reference to any service implementation, it is much more extensible and maintainable to ask for a factory of such instance, just like we did in other demos. The `ILoggerFactory` implements that functionality.

He goes even further, highlighting that: "*you ask for an interface (such as ILoggerFactory) with the expectation that the service provider (in this case, NLog, Log4Net or Serilog) will implement the interface*".

Thus, `ILoggerFactory` is extensible even with third-party libraries! He also remarks that: "*the result is that while the client will directly reference the abstract assembly (Logging.Abstractions), defining the service interface, no references to the direct implementation will be needed*".

Actually, you may have noticed that the `ServiceCollection` itself owns a method called `AddLogging()`, which is a convenient method to activate logging functionality for the collection. This call is really translated to an internal registry of the `ILoggerFactory` service as part of the services collection we are configuring.

Considering that we can also concatenate calls to configure our service collections, let's see the first part of our new `Main()` method, including that call:

```
    // Enabling logging with the ServiceCollection
    var services = new ServiceCollection()
      .AddSingleton<IXMLWriter, XMLWriter>()
      .AddLogging();
  var serviceProvider = services.BuildServiceProvider();
```

WOW! eBook
www.wowebook.org

Consequently, we enable logging inside the `ServiceCollection` via `AddLogging()`. So, what happened? Let's check the new members that our collection now has, by iterating through the resulting services object, like in a previous demo:

```
 // Test the register of AddLoggin()
 foreach (var svc in services)
{
   Console.WriteLine($"Type: {svc.ImplementationType} \n" +
     $"Lifetime: {svc.Lifetime} \n" +
     $"Service Type: {svc.ServiceType}");
}
```

We'll obtain a collection containing three services (see the following output), because the `AddLogging()` method has, indeed, registered a `LoggingFactory` class with the `ILoggingFactory` interface, and another generic one, `ILogger<>`. That `ILogger<>` class will later be configured to provide any other class with the ability to send logging messages to the console:

```
C:\Program Files\dotnet\dotnet.exe
Type: LoggerFactory.XMLWriter
Lifetime: Singleton
Service Type: LoggerFactory.IXMLWriter
Type: Microsoft.Extensions.Logging.LoggerFactory
Lifetime: Singleton
Service Type: Microsoft.Extensions.Logging.ILoggerFactory
Type: Microsoft.Extensions.Logging.Logger`1[T]
Lifetime: Singleton
Service Type: Microsoft.Extensions.Logging.ILogger`1[TCategoryName]
```

Therefore, the next step is to obtain an `ILoggerFactory` object and link that object to the `Console`, which we perform with a call to `AddConsole()` in this fashion:

```
//Obtain service and configure logging
serviceProvider.GetService<ILoggerFactory>()
  .AddConsole(LogLevel.Debug);
```

As you see, `AddConsole` expects some additional configuration in the form of an `enum` value of type `LogLevel`, which determines the minimum level of severity the runtime will filter when sending messages to the console--whenever the logging system receives an entry, it will ignore it if it is below that level.

WOW! eBook
www.wowebook.org

LogLevels Severity

The `LogLevel` enumeration also establishes the priority of messages (pretty useful in case we want to distinguish low-level from high-level messages, and even redirect them to different output windows).

According to the official documentation these levels are organized as follows (it goes from least to highest severity):

- **Trace = 0**: For information that is valuable only to a developer debugging an issue. These messages may contain sensitive application data and so should not be enabled in a production environment. Disabled by default. For example, credentials: `{"User":"someuser", "Password":"P@ssword"}`
- **Debug = 1**: For information that has short-term usefulness during development and debugging. For example, entering the `Configure` method with fl the g set to true.
- **Information = 2**: For tracking the general flow of the application. These logs typically have some long-term value. For example, the request received for path `/api/todo`.
- **Warning = 3**: For abnormal or unexpected events in the application flow. These may include errors or other conditions that do not cause the application to stop, but which may need to be investigated. Handled exceptions are a common place to use the warning log level. For example, `FileNotFoundException` for file `quotes.txt`.
- **Error = 4**: For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an application-wide failure. For example, log message: `Cannot insert record due to duplicate key violation`.
- **Critical = 5**: For failures that require immediate attention. For example, data loss scenarios, out of disk space.

The IDE also shows those levels through the IntelliSense services, together with an explanation of each purpose and functionality:

WOW! eBook
www.wowebook.org

```
//Obtain service and configure logging
serviceProvider.GetService<ILoggerFactory>()
    .AddConsole(LogLevel.);
```

Critical
Debug
Error
Information
None
Trace
Warning

LogLevel.Debug = 1
Logs that are used for interactive investigation during development.

These logs should primarily contain information useful for debuggin

In this way, the returned `ILoggerFactory` service we get when asking for a reference will redirect any output to the console using the debug level configuration, if not instructed otherwise.

The `ILogger` and `ILoggerFactory` interfaces are located in `Microsoft.Extensions.Logging.Abstractions`, and default implementations of them are in `Microsoft.Extensions.Logging`.

Now, if we want both classes (`XMLWriter` and `Program`) to use these logging services, we need an `ILogger` instance for each one. We'll start creating one for `Program` and presenting a first set of messages in the console:

```
// Create a logger class from ILoggerFactory
// and print an initial set of messages.
var ILoggerService = serviceProvider.GetService<ILoggerFactory>();
var logger = ILoggerService.CreateLogger<Program>();
```

Notice that the creation of the logger class implies calling the `CreateLogger<Program>()` generic method. Once instantiated, the logger has methods to declare different scopes, (marking the beginning and the end of each scope) and to send six different types of messages to the console, each one representing a different severity level:

```
logger.LogCritical("Critical format message from Program");
logger.LogDebug("Debug format message from Program");
logger.LogError("Error format message from Program");
logger.LogInformation("Information format message from Program");
logger.LogTrace("Trace format message from Program");
logger.LogWarning("Warning format message from Program");
```

If we take a look at the output, we can appreciate the differences between those messages' formats:

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
crit: LoggerFactory.Program[0]
      Critical format message from Program
dbug: LoggerFactory.Program[0]
      Debug format message from Program
fail: LoggerFactory.Program[0]
      Error format message from Program
info: LoggerFactory.Program[0]
      Information format message from Program
warn: LoggerFactory.Program[0]
      Warning format message from Program
```

As you can see, the different messages' severity levels generate different formats in the output using distinct colors and prefixes to denote its `LogLevel` category. But, wait a minute! There's one missing (the `Trace` message).

Well, not quite. What happens is that the `Trace LogLevel` doesn't output to the console, and it is prepared to be used mainly in web applications in which the trace switch is enabled (we will see more on that in `Chapter 4`, *Dependency Injection in ASP.NET Core*).

So, all that said, how can we use this architecture and logging services from our `XMLWriter` class? Let's change the implementation, so we use one of the DI patterns that we saw in the previous chapter--the constructor dependency model.

To use that model, we have to change our `XMLWriter` class a little to include a read-only property that holds the `ILogger<>` instance and assign its value in the class' constructor. So, the final format of our, this time, unique `XMLWriter` classes will be (the interface definition is not affected yet, so it's just the same as in previous demos):

```
public class XMLWriter : IXMLWriter
{
  private readonly ILogger<XMLWriter> logger;
  public XMLWriter(ILoggerFactory loggerFactory)
  {
    logger = loggerFactory.CreateLogger<XMLWriter>();
  }
  public void WriteXML()
  {
    logger.LogInformation("<message>Writing in XML Format
      (via Logger)</message>");
  }
}
```

The only thing remaining is to use the `logger` instead of the previous calls to `Console` and call one of the `Log*` methods to generate the intended output. And that's it.

WOW! eBook
www.wowebook.org

We have totally replaced the functionality provided by the `Console` class with the logging services delivered by the `ILogginFactory` and `ILogger` objects, and we can also configure the severity level to generate distinct output formats.

So, with all that in mind, we end up with a final version of our `Main` method with this implementation:

```
static void Main(string[] args)
{
  // Enabling logging in the ServiceCollection
  // via AddLogging()
  var services = new ServiceCollection()
    .AddSingleton<IXMLWriter, XMLWriter>()
    .AddLogging();
  var serviceProvider = services.BuildServiceProvider();
  //Obtain service and configure logging
  serviceProvider.GetService<ILoggerFactory>()
    .AddConsole(LogLevel.Debug);

  // Create a logger class from ILoggerFactory
  // and print all types of messages.
  var ILoggerService = serviceProvider.GetService<ILoggerFactory>();
  var logger = ILoggerService.CreateLogger<Program>();
  logger.LogCritical("Critical format message from Program");
  logger.LogDebug("Debug format message from Program");
  logger.LogError("Error format message from Program");
  logger.LogInformation("Information format message from Program");
  logger.LogTrace("Trace format message from Program");
  logger.LogWarning("Warning format message from Program");

  //Instantiation of XMLInstance
  var XMLInstance = serviceProvider.GetService<IXMLWriter>();
  XMLInstance.WriteXML();

  logger.LogDebug("Process finished!");
  Console.Read();
}
```

As we can see in the final output (see the following screenshot), all messages are presented in the console in the same sequence they're invoked, using the predefined formats that .NET Core configures, and including our `XMLWriter` message:

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
crit: LoggerFactory.Program[0]
      Critical format message from Program
dbug: LoggerFactory.Program[0]
      Debug format message from Program
fail: LoggerFactory.Program[0]
      Error format message from Program
info: LoggerFactory.Program[0]
      Information format message from Program
warn: LoggerFactory.Program[0]
      Warning format message from Program
info: LoggerFactory.XMLWriter[0]
      <message>Writing in XML Format (via Logger)</message>
dbug: LoggerFactory.Program[0]
      Process finished!
```

This is not all. We still have other options available that allow us to separate and filter the output message destination. This possibility is correlated to the `Microsoft.Extensions.Logging.Debug` library, which should also be referenced just like the others we're using here.

One of the features included in this library is the `AddDebug()` method of `ILoggerFactory`. Once activated, it permits us to send messages to the Debug window, enabling a separation of messages conditional to its severity level, for instance.

To test this functionality, we'll make a few changes in our definitions of the `IXMLWriter` interface and update the implementation accordingly. Our new interface will have another method that also sends messages to the predefined output (which, in this case, will show up in several places at runtime):

```
public interface IXMLWriter
{
    void WriteXML();
    void WriteXMLWithSeverityLevel();
}
```

So, the updated code of XMLWriter will be:

```
public class XMLWriter : IXMLWriter
{
  private readonly ILogger<XMLWriter> logger;
  public XMLWriter(ILoggerFactory loggerFactory)
  {
    loggerFactory.AddDebug().AddConsole(LogLevel.Information);
    logger = loggerFactory.CreateLogger<XMLWriter>();
  }
  public void WriteXML()
  {
    logger.LogDebug(
```

WOW! eBook
www.wowebook.org

```
            "<msg>First Message (LogDebug/SeverityLevel:
                 Information)</msg>");
      }
      public void WriteXMLWithSeverityLevel()
      {
        logger.LogDebug(
            "<msg>Second Message (LogDebug/SeverityLevel:
                 Information</msg>");
      }
    }
```

So, now we have two distinct methods to write messages. To test this functionality, we can configure the `ILoggerService` object in the `Main()` method (which, remember, is of type `ILoggerFactory`). Once the new namespace is loaded and available, we can write:

```
var ILoggerService = serviceProvider.GetService<
      ILoggerFactory>();
ILoggerService.AddDebug();
```

In this manner, we allow sending messages to the `Debug` or `Output` windows, either for console or web applications.

It's easy to test different options, changing the types of severity levels used in the calls to this method, and the already existing ones. For example, we can make a call to `WriteXMLWithSeverityLevel()` and observe the two outputs (now we have two), that are generated during execution:

```
//Instantiation of XMLInstance
var XMLInstance = serviceProvider.GetService<IXMLWriter>();
XMLInstance.WriteXML();
XMLInstance.WriteXMLWithSeverityLevel();
```

On one side, the output now presents the new message (nothing unexpected):

WOW! eBook
www.wowebook.org

```
C:\Program Files\dotnet\dotnet.exe
crit: LoggerFactory.Program[0]
      Critical format message from Program
dbug: LoggerFactory.Program[0]
      Debug format message from Program
fail: LoggerFactory.Program[0]
      Error format message from Program
info: LoggerFactory.Program[0]
      Information format message from Program
warn: LoggerFactory.Program[0]
      Warning format message from Program
dbug: LoggerFactory.XMLWriter[0]
      <msg>First Message (LogDebug/SeverityLevel: Information)</msg>
dbug: LoggerFactory.XMLWriter[0]
      <msg>Second Message (LogDebug/SeverityLevel: Information</msg>
dbug: LoggerFactory.Program[0]
      Process finished!
```

But, now, we have more messages. If we take a look at the `Output` window, we'll see the new entries depending on the `LogLevel` we configure--some of them will be presented, and others, ignored (as you can see, in this version, only the first four messages are replicated in the Output window, and all XMLWriter messages are ignored):

```
Output
Show output from: Debug
'dotnet.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\1.0.3\System.Text
'dotnet.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\1.0.3\System.Text
'dotnet.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\1.0.3\System.Diag
LoggerFactory.Program:Critical: Critical format message from Program
LoggerFactory.Program:Error: Error format message from Program
LoggerFactory.Program:Information: Information format message from Program
LoggerFactory.Program:Warning: Warning format message from Program
The thread 0x1d18 has exited with code 0 (0x0).
The thread 0x38d8 has exited with code 0 (0x0).
```

This is only a first approach to some of the services that adopt the DI architecture and that are available inside .NET Core. We will see much more on these implementations when dealing with ASP.NET Core coding, in `Chapter 4`, *Dependency Injection in ASP.NET Core*.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we have made a first approach to how Dependency Injection technologies are supported and implemented in the current version of the .NET Core (2.0) and how we can use them, even outside of ASP.NET projects.

To summarize, we've seen the main characteristics of NET Core and its installation and usage from Visual Studio, especially focused on the latest version, Visual Studio 2017, together with the distinct types of deployment linked to this version of the framework, along with the main features and functionality included in the namespaces related to DI and, especially, inside the `Microsoft.Extensions.DependencyInjection` and `Microsoft.Extensions.Logging` containers.

We've also analyzed the classes and interfaces related to this architecture, together with its implementation through a bunch of samples, ending with some real implementations, that are already part of .NET Core 2.0, such as the logging services, and how to use them, from any class.

In `Chapter 4`, *Dependency Injection in ASP.NET Core*, our approach will be more real, since it deals with web applications and how the new architecture and configuration of websites manage these new concepts, such as middleware and services configuration, in which DI plays an important role since the very beginning.

WOW! eBook
www.wowebook.org

Dependency Injection in ASP.NET Core

In Chapter 3, *Introducing Dependency Injection in .NET Core 2.0*, we were focusing on .NET Core exclusively, analyzing the possibilities the platform offers in relation to Dependency Injection and distinct ways to implement it. In this chapter, we'll continue analyzing DI, but, this time, focusing on the implementation of ASP.NET Core and the possibilities offered to the programmer at the time of configuring websites and other related functionality which spreads along the whole life cycle.

The idea is to start with the CLI (command-line tools) and see how to modify a console application and turn it into a web app, so you can better understand the concept of middleware and the way it's used in ASP.NET Core.

With that, we will be ready to move onto an analysis of the default templates that Visual Studio 2017 offers for ASP.NET Core apps, and the specific features linked to DI in those initial models.

Finally, we'll see how to tune up our own services and how to use them via Dependency Injection in registration options, controllers, and views.

In all, we will cover the following topics in this chapter:

- Building ASP .NET Core apps using command-line tools.
- Middleware architecture in ASP .NET Core.
- Analyzing the default templates offered by Visual Studio.
- Features of DI in ASP.NET applications. Customized services.

TIP

Notice the official documentation site uses the same code as the templates offered by Visual Studio 2017. You can find it at https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc.

WOW! eBook
www.wowebook.org

ASP.NET Core from the command-line tools

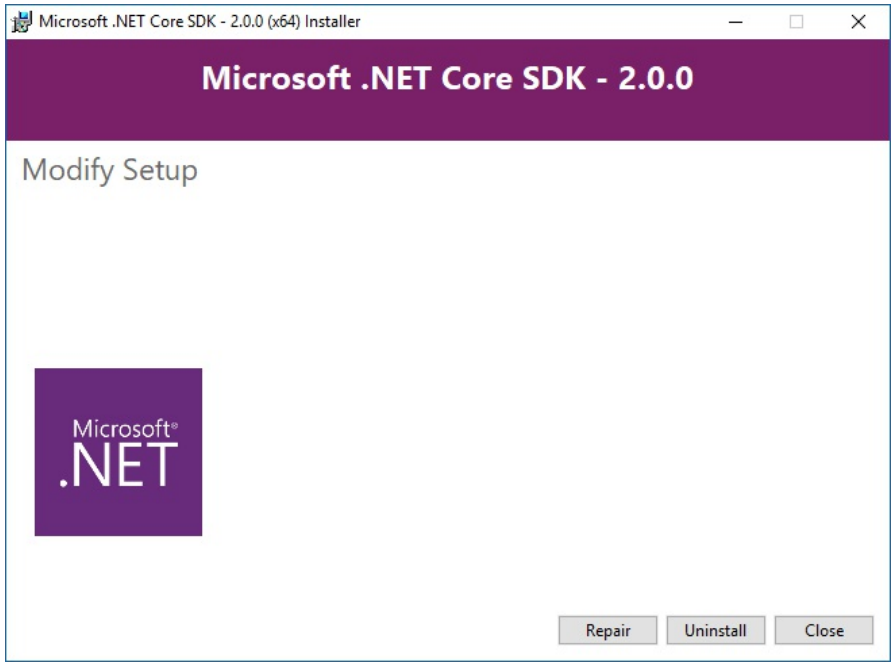
Once we have installed .NET Core in our boxes and updated (if necessary) to version 2.0, we can start a very simple, but explanatory, website from scratch using command-line tools and see how we can move from that minimum .NET Core Console app to an ASP.NET Core app in a few steps.

This process will help you understand the profound changes in architecture that ASP.NET Core carries, and how some of the SOLID principles that we have seen are applied in a variety of ways to achieve this goal.

Thus, the first step should be to check out our installed version of .NET Core, which we can do in a command-line window (remember that Visual Studio 2017 installs several links to these windows in the Windows menu, and that the Developer Command Prompt already has environment variables defined to suit the main needs of a programmer).

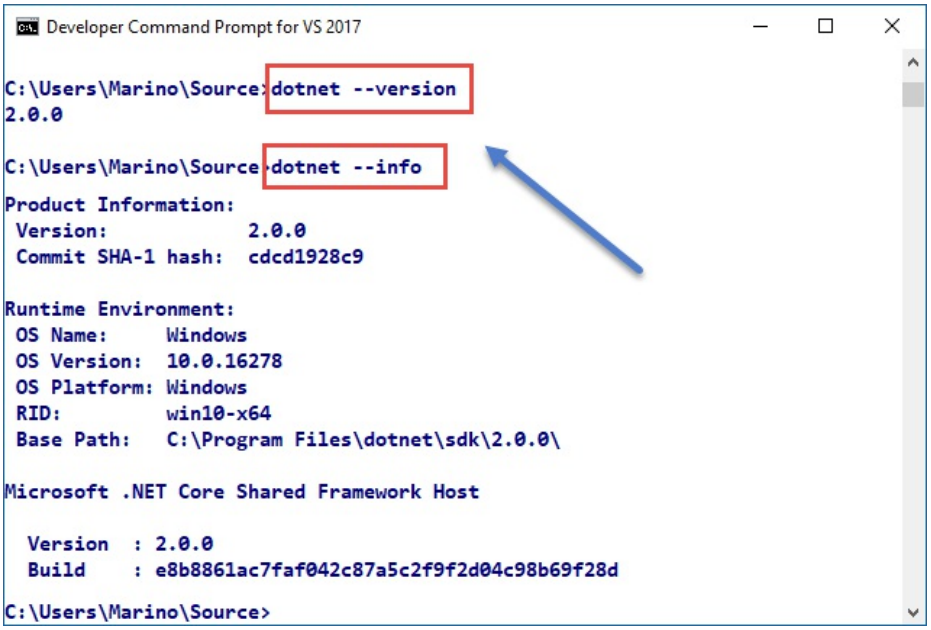
If you do not already have the .NET Core command-line tools installed, remember that you can do it in a separate install from the site `https://www.microsoft.com/net/download/core` in the Command-Line/Other section, which lets you download the installer for all versions currently supported. When run, the installer should look as follows:

WOW! eBook
www.wowebook.org



After installation, a new folder should appear at `C:\Program Files\dotnet\sdk` named 2.0, (or whichever is the latest version available at the time).

So, we can just check the current version's details with the `dotnet -version and dotnet --info` commands, and see if we have version 2.0 installed:



The window will show some basic commands, such as `--help` and `--version`. To check the types of templates available for the latest version, just type `dotnet new` (with no extra parameters), and you'll be presented a list of this kind:

WOW! eBook
www.wowebook.org

```
---------------------------------------------------------------------------------------------------
Console Application                                console          [C#], F#, VB      Common/Console
Class library                                      classlib         [C#], F#, VB      Common/Library
Unit Test Project                                  mstest           [C#], F#, VB      Test/MSTest
xUnit Test Project                                 xunit            [C#], F#, VB      Test/xUnit
ASP.NET Core Empty                                 web              [C#], F#          Web/Empty
ASP.NET Core Web App (Model-View-Controller)       mvc              [C#], F#          Web/MVC
ASP.NET Core Web App                               razor            [C#]              Web/MVC/Razor Pages
ASP.NET Core with Angular                          angular          [C#]              Web/MVC/SPA
ASP.NET Core with React.js                         react            [C#]              Web/MVC/SPA
ASP.NET Core with React.js and Redux               reactredux       [C#]              Web/MVC/SPA
ASP.NET Core Web API                               webapi           [C#], F#          Web/WebAPI
global.json file                                   globaljson                         Config
Nuget Config                                       nugetconfig                        Config
Web Config                                         webconfig                          Config
Solution File                                      sln                                Solution
Razor Page                                         page                               Web/ASP.NET
MVC ViewImports                                    viewimports                        Web/ASP.NET
MVC ViewStart                                      viewstart                          Web/ASP.NET
```

Notice that the first time you type that command it will decompress some files (it takes a few seconds), to initially populate your local package cache. That will only happen once, improving restore speed and enabling offline access.

WOW! eBook
www.wowebook.org

Creating the minimum possible application

We have 11 default types of templates (you can install extra ones from GitHub) for as many project types, plus a solution option that creates a `.sln` file including the projects inside the depending folders, and distinct options related to configuration and Web/ASP.NET solutions.

To test this latest version, inside the Command Prompt, create a new folder, move to it, and just type a `dotnet new console` command. This creates two files that define the simplest app (console app) for .NET Core.

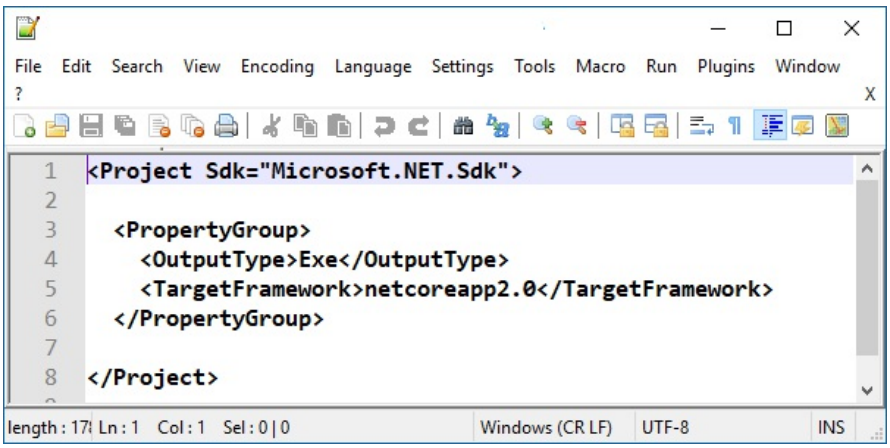
At that moment, you should see a `program.cs` and a `[NameOfDirectory].csproj` file, which contains dependency information for your app.

Notice that previous versions used a `project.json` file instead, with the same information, but if you open any previous project with V. Studio 2017 it will recognize it and migrate it automatically.

The contents of the `.csproj` file, contain a few basic directives in XML format.

To keep working with very basic resources in this initial demo, I'm going to use Notepad++ for code coloring and some other editing features.

You should see the following content in the `.csproj` file:



WOW! eBook
www.wowebook.org

As you see, it indicates we're using the .NET SDK, the output is an `exe` file, and we're targeting NET Core 2.0.

The contents of our `program.cs` file are as expected (no changes from a typical Console application, in a classical .NET Framework):

```
 class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Hello World!");
   }
}
```

But, still, we need to download the dependencies, a task that is performed using a `dotnet restore` command previous to any other options. After issuing that command, you'll see how it downloads all required packages, and a new `obj` directory shows up.

And that's it. You can finally issue the `dotnet run` command, which, in turn, compiles and executes the app, showing the expected message in the console (nothing new, I omited the output).

WOW! eBook
www.wowebook.org

Changing to ASP .NET Core

Now, to change our application into an ASP.NET Core app, the first thing to do is to install the package called `Microsoft.AspNetCore`. We can do this by issuing the `dotnet add package Microsoft.AspNetCore` command.

When we do that, the command-line tool will download the appropriate package and modify our `.csproj` accordingly, so it is included in our solution as well (see the new version of `csproj` after issuing the command):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  </ItemGroup>
</Project>
```

We see the presence of a new `<ItemGroup>` tag, indicating the reference to include and the version that has been downloaded and added to the project.

We're now ready to create our web starting now that will take the form of a class named `Startup.cs` (of course you can name it as you want as long as you configure it later. It's not a convention name).

Inside that file, we'll reference the three extra namespaces required to create a website (although a basic one):

- `Microsoft.AspNetCore.Builder`: To actually build a web server using the `config` parameters we've defined
- `Microsoft.AspNetCore.Hosting`: To hold the web application
- `Microsoft.AspNetCore.Http`: For all HTTP-related activities

WOW! eBook
www.wowebook.org

With those references, we need to add a method named `Configure` (and this is by convention), where we will indicate the minimum actions to perform when the server is launched.

Here, we will start seeing DI in action, because the shape of this very basic method is as follows:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

namespace WebApplication1
{
  public class Startup
  {
    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env)
    {
      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }
      app.Run(async (context) =>
      {
        await context.Response.WriteAsync("This is a first
              web app...");
      });
    }
  }
}
```

But, before we start explaining the inner details of that file, let's make sure it compiles correctly and that we redirect our application to the new website.

So, we will issue another `dotnet restore` command, so all new references are located properly, and the next step will be to modify our main entry point, to create a new web host that uses the `Startup` class just created.

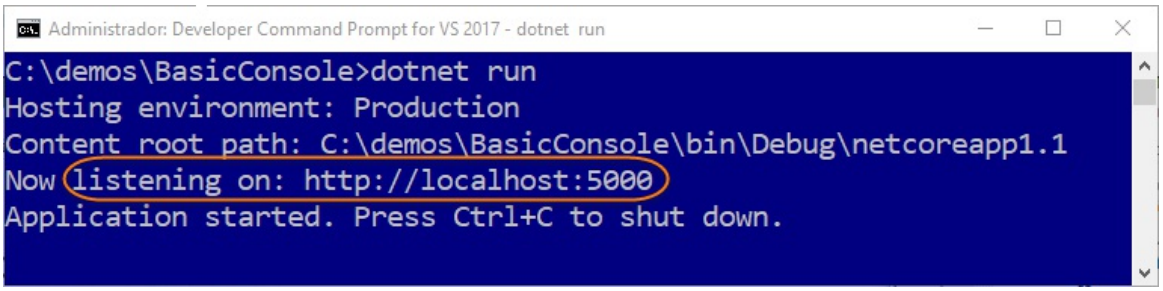
With that purpose, the new `Main()` entry point will use a reference to `Microsoft.AspNetCore.Hosting` namespace, and define the following:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication1
{
  public class Program
  {
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }
```

WOW! eBook
www.wowebook.org

```
        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

Finally, we can repeat the `dotnet run` command and we will see two different outputs. On one side, the CLI environment, we will generate a web host and run it (by default, using port number `5000`), indicating that in the console output:



That host will keep on running and listen on that port until we stop it using *Ctrl + C*. Now, we can open a browser and type the URL to see the following page:

Of course, that page doesn't contain any HTML, only the text we order the server to send back to the user when a request is received on that port. Just take a look at the source code if you want to check this out.

But, before we continue the explanation of the previous code, it's important to understand the concept of Middleware, present from the beginning in ASP.NET Core.

WOW! eBook
www.wowebook.org

Middleware

Of course, middleware refers to software, but to a software that is assembled using an application pipeline in a manner that facilitates handling requests and responses.

Obviously, you may wonder what an application pipeline is. Wikipedia defines the terms like this: "*In software engineering, a pipeline consists of a chain of processing elements (processes, threads, coroutines, functions, etc.), arranged so that the output of each element is the input of the next; the name is by analogy to a physical pipeline*".

These pipelines are common in many software languages, even in modern constructions as in JavaScript Promises, which define asynchronous pipelines for a sequence of calls, improving execution control.

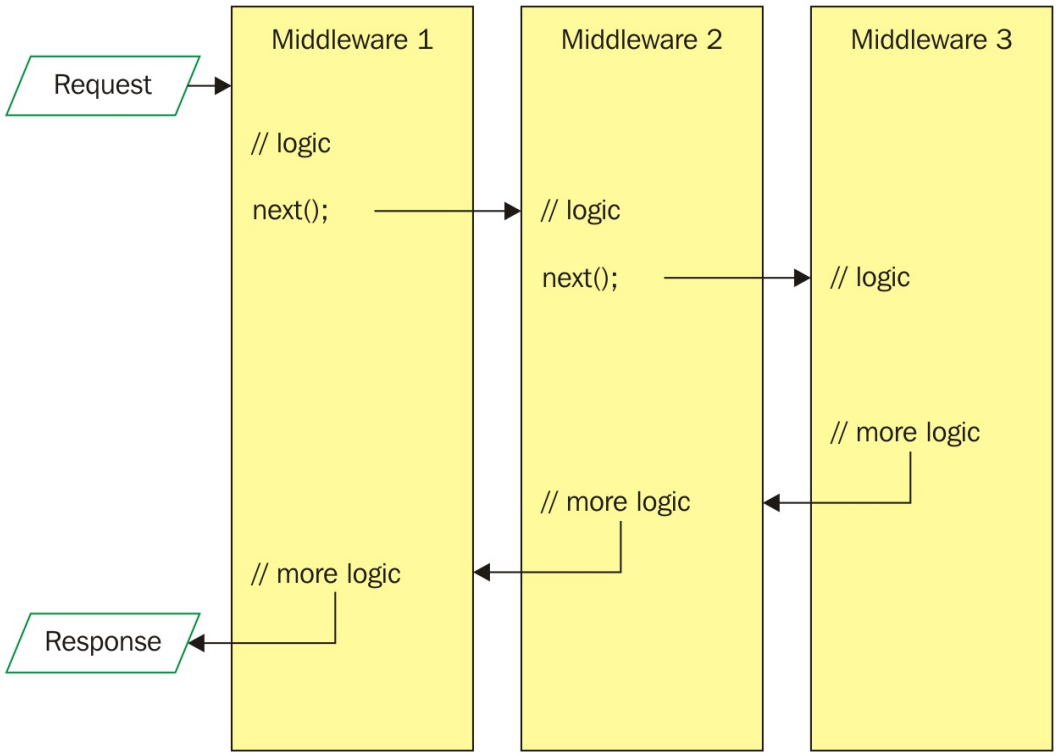
One important feature of pipelines is that each component of the pipeline can decide whether it passes information to the next component or returns directly, and has the ability to perform custom actions before and after that component is invoked.

In ASP.NET Core, to build a request pipeline, we use delegates to handle each request. And, again, we find some of the SOLID principles present in this architecture.

Every time you need to configure one of these delegates, you use one method belonging to the method's families called `Use*`, `Run*` and `Map*` (they are sets of predefined extension methods starting with these prefixes, each one with a specific purpose).

These methods are linked to the `IApplicationBuilder` object that the `Configure` method receives via Dependency Injection. The following schema visually explains this structure (notice that the thread of execution follows the black arrows):

WOW! eBook
www.wowebook.org



In this manner, it's totally up to the programmer to keep on passing the `IApplicationBuilder` object to another middleware piece (notice the call to `next()`) or avoid any other call, in which case it returns to the calling method.

If one of these delegates decides not to pass a request to the next delegate, that's called **short-circuiting** the request pipeline. This is often convenient to avoid having to perform any unnecessary work.

In the case of exception handling delegates, they need to be called early in the pipeline, to allow them to catch exceptions that might occur later in the sequence.

Let's go now for the explanation of the previous code with all this in mind.

WOW! eBook
www.wowebook.org

The Startup class and Dependency Injection

The `Startup` class coded previously is, therefore, the component in charge of configuring the request pipeline and handling all requests made to the application.

This class is mandatory (although, as mentioned, it could be named differently) because the runtime will look inside it for the methods that take care of configuration aspects, and that includes services as well. You can consider it as a convenient way to achieve independent configurations via the first SOLID principle (SoC).

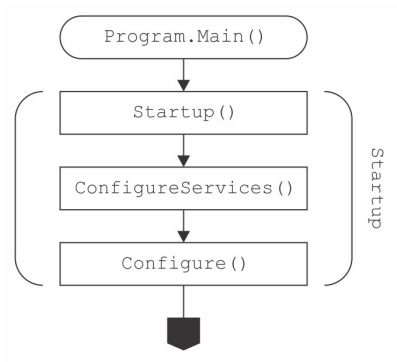
One of the interesting features linked to this separation of responsibility is that you can define distinct **Startup** classes depending on the environment (development, production, and so on). The appropriate one will be selected at runtime.

This class is defined in such a way that accepts dependencies provided via Dependency Injection, as well. For instance, you can declare variables of types `IHostingEnvironment` or `ILoggerFactory` to gain the ability to configure diverse functionality related to hosting or logging (remember in, `Chapter 3`, *Introducing Dependency Injection in .Net Core 2.0*).

So, what is mandatory in this class and what is optional? The `Configure` method is mandatory. No matter what the name of the class, the runtime will look for it inside and call it to make sure the required conditions apply.

However, the `ConfigureServices()` method is optional, but if it is present, it will be called prior to `Configure()` during that initial process. The following schema illustrates this sequence:

WOW! eBook
www.wowebook.org



(Image courtesy: http://developer.telerik.com/featured/understanding-asp-net-core-initialization/)

Before going any further in demos, let's explain the previous code in more detail.

WOW! eBook
www.wowebook.org

Code explanation

Starting with the `Main()` method, after a reference to the `Hosting` subspace that we mentioned previously, we build a new web server by calling the `WebHostBuilder` class. This class permits configuration and building of a web server using various middleware pieces and entry point conditions.

Thus, after calling the constructor, we piped another three calls:

- One to `UseKestrel()`, which is the default, light-weight server for development that Visual Studio 2017 (and the CLI) uses (we'll explain that in a while)
- Another call to `UseStartup<Startup>()`, to indicate where the server will find the `Configure()` method to initiate its process (remember the name of the class is irrelevant, as far as it contains the method `Kestrel` looks for initially)
- And, finally, another one to `Build()`, which creates and initializes the new server with those previous values

With all that in place, that last sentence simply calls the `Run()` method to launch the process.

When launching that process, the `Configure` method turns on. Its only parameter (in this case), is of type `IApplicationBuilder`, and, as you see, is served to this method by Dependency Injection (no previous references or instantiation were made by our code).

Consequently, when a server is created, the main, basic objects implied in the communication process are served in this manner, expecting from the user all required behaviors in the form of subsequent calls to its methods.

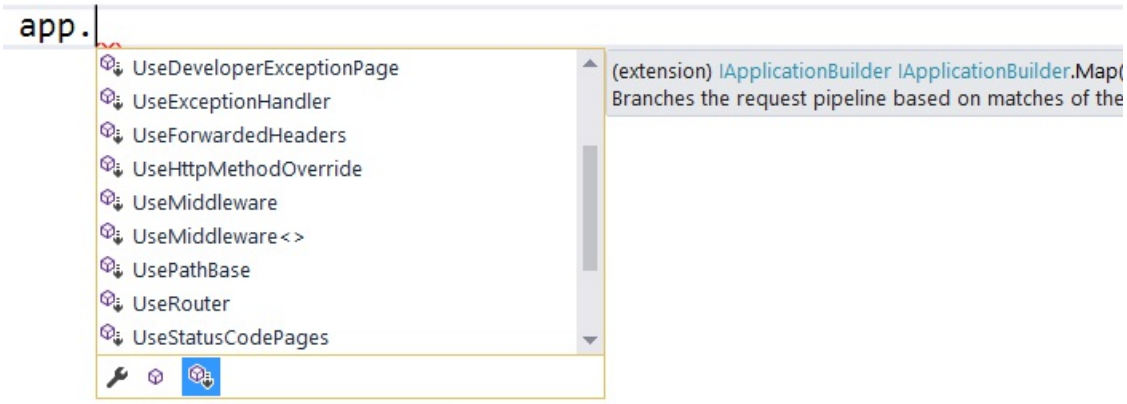
It's pretty self-explanatory if we take a look at the interface definition:

WOW! eBook
www.wowebook.org

```
namespace Microsoft.AspNetCore.Builder
{
    public interface IApplicationBuilder
    {
        IServiceProvider ApplicationServices { get; set; }
        IFeatureCollection ServerFeatures { get; }
        IDictionary<string, object> Properties { get; }

        RequestDelegate Build();
        IApplicationBuilder New();
        IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware);
    }
}
```

By the time the `Configure` method receives an instance of the `IApplicationBuilder` class, a bunch of extra functionality is available. As mentioned previously, it has been made available via extension methods, adopting the form of `Use*`, `Run*`, and `Map*` methods, which help the programmer in coding configuration aspects and obtaining more independence and granularity.

Notice how the following screenshot shows different configuration options and suggests the use of middleware:



Each of these extension methods supplies a way to call the implied delegate. Notice that while the `Use*` family of methods calls `Next()` implicitly many times to keep on going in the pipeline (actually it depends on other features), the `Run*` family stops the propagation and short-circuits the pipeline (therefore, it doesn't call a next request delegate).

Also, the `Map*` family allows branching the pipeline, making calls that return to that point and extending functionality accordingly.

WOW! eBook
www.wowebook.org

The New ASP.NET servers

Let's quickly review some important aspects of the servers used when programming for ASP.NET Core since that is one of the main changes linked to this new platform.

First of all, an ASP.NET Core application runs an in-process HTTP server implementation. That implementation is listening for HTTP requests and sends those requests to the application wrapped in an object named `HttpContext` that holds a set of features composed into it.

This version of ASP.NET provides two distinct server implementations: `Kestrel` and `WebListener`. As the official documentation reminds us:

WOW! eBook
www.wowebook.org

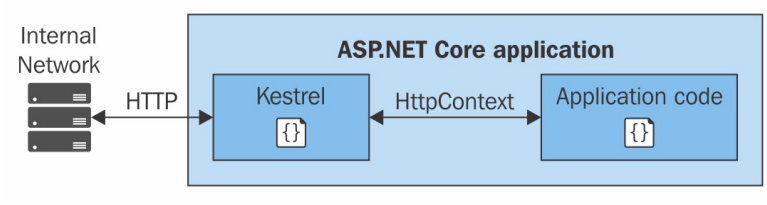
Kestrel

`Kestrel` is a cross-platform HTTP server based on `libuv`, a cross-platform asynchronous I/O library:

`libuv` is defined as a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others.

Kestrel is the web server included by default in the ASP.NET Project templates. The advantage is that if your application accepts requests exclusively from an internal network, it can be used by itself.

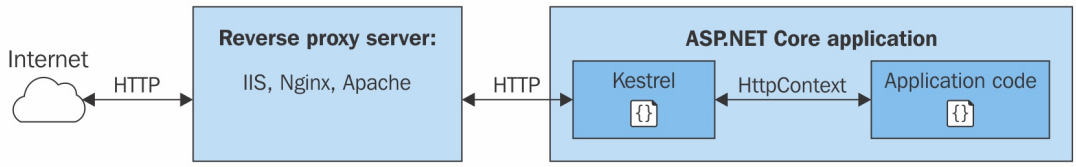
This is the working scheme of a default scenario with Kestrel:



(Image courtesy: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/)

However, if you expose the application to the internet, there are some security issues for which Kestrel is not prepared to cope (it's relatively new and still doesn't have the whole required set of defense resources). For those cases, the recommended configuration is to use a reverse proxy server, such as IIS, Apache, or Nginx, to provide functionality.

Notice that a reverse proxy server, as the documentation states, *receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling* (see the following screenshot):



(Image courtesy: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/)

WOW! eBook
www.wowebook.org

Another important point is that you can't use any of those reverse proxy servers without Kestrel or a custom server implementation. This is because ASP.NET Core was designed to run in its own process so it can behave consistently across platforms.

The problem that we might face here is that IIS, Nginx, and Apache prescribe their own startup process and environment. The consequence is that to use them directly it should adapt to the requirements of each one.

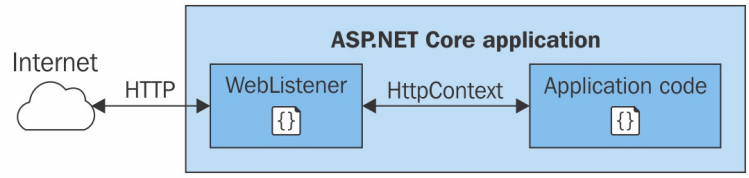
In this way, Kestrel gives ASP.NET Core the ability to code `Program` and `Startup` classes in any required form to suit the user's needs, while avoiding another dependency on a concrete, specific server. That's one of the reasons why middleware is so important in this context.

WOW! eBook
www.wowebook.org

WebListener

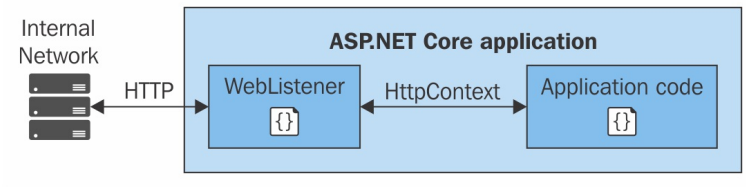
`WebListener` is a Windows-only HTTP server based on the `Http.Sys` kernel driver. It serves as an alternative for those scenarios where exposing our app to the internet is mandatory, but we don't wish to use IIS if we cannot do it for some reason.

The following schema represents the role of WebListener in a similar scenario as the one presented previously with Kestrel:



(Image courtesy: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/)

In the same way, if you need some of the features that Kestrel doesn't support while working on an internal network, you could use WebListener in a quite comparable configuration:



(Image courtesy: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/)

Finally, keep in mind that for internal network scenarios, Kestrel is recommended because it provides improved performance. In any case, if you want to know more about the features provided by WebListener, the official documentation is available at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/weblistener.

WOW! eBook
www.wowebook.org

Dependency Inversion in the architecture - OWIN

It's also possible to create your own server implementation to work with ASP.NET Core. **Open Web Interface for .NET** (**OWIN**) is the third possible implementation supported by the platform.

In some ways, the implementation of OWIN also relates to the Dependency Inversion Principle. Its main goal is to make web applications decoupled from web servers.

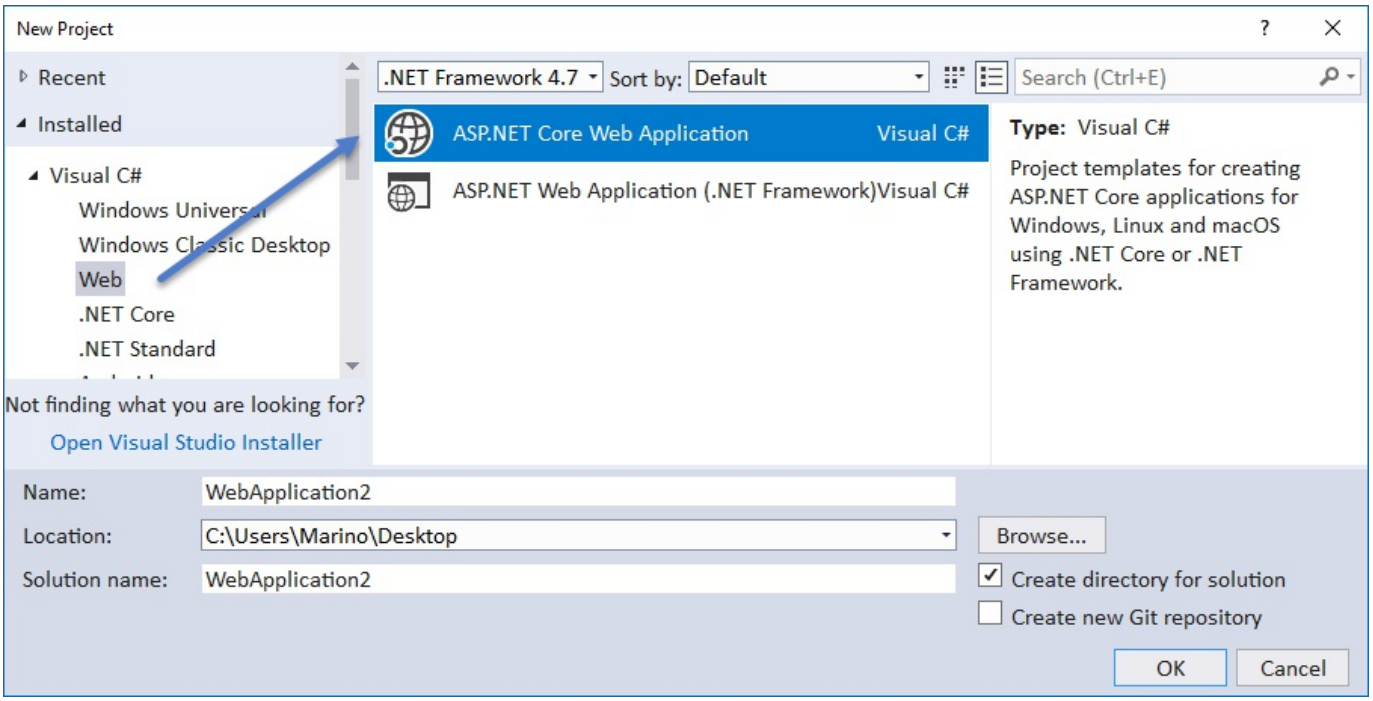
With that purpose, it defines standard ways to create middleware that can be used in the pipeline, to configure and tune your server.

WOW! eBook
www.wowebook.org

Analyzing the default template

Now it's time to open Visual Studio 2017 (or 2015, with ASP.NET Core installed) and take a look at it and explain how the default templates behave.

Remember that ASP.NET Core can execute using the classical.NET Framework as well, so, when you create a new Web project you'll be asked, initially, to choose between three main options: classical ASP.NET, ASP.NET Core with NET Core, and ASP.NET Core with classical .NET Framework:



Once selected, you'll be offered another choice window: Empty, Web API, Web Application, Web Application (Model-View-Controller), and some new options added to version 2.0 for third-party libraries, which include Angular, React.js, and React.js and Redux.

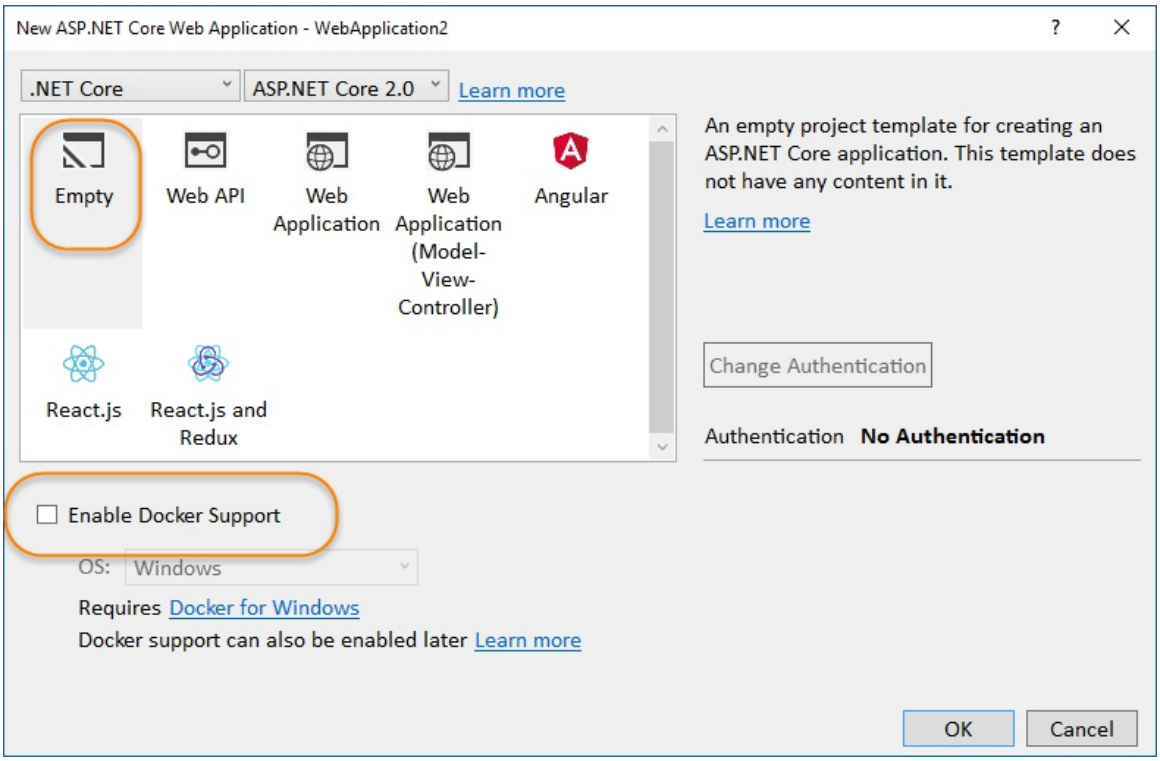
In the first case, the one we're using now creates an application with the minimum configuration to be able to create and run a web server using Kestrel and present some text in the browser. It's simple, but it allows us to check out how it is done with more detail and to make some changes.

The other three (Web API, Web Application, and Web App MVC) remind their

WOW! eBook
www.wowebook.org

counterpart in classic ASP.NET, with the difference that they use the new architecture and configuration files. In this manner, we'll be able to better appreciate the migration process needed to pass from the old architecture to the new one.

You should see the following dialog box:



Notice that you can also change authentication as in previous versions and that you have a checkbox that allows enabling Docker support.

The resulting project is a bit more complex than the basic demo we did previously, although the basic pieces are the same. However, there are some noticeable changes in configuration.

WOW! eBook
www.wowebook.org

Configuration files

Once you compile the application and activate the See all files option in the Solution Explorer, you'll notice some extra configuration files that take care of a few tasks, such as launching the browser on the default URL. These options are defined inside the `launchSettings.json` file available inside the `Properties` directory.

A look at it is also quite explanatory:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:57539/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "ASPNETCoreDemo1": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
       "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:57540"
    }
  }
```

As you see, three main areas of configuration apply here: `iisSettings`, to indicate IIS behavior, including the URL to use, a `profiles` section that only contains one profile for `IISExpress`, indicating that a browser should be launched and a hint on the development mode, and a final one under the name of the application itself (`ASPNETCoreDemo1`), with similar information.

Of course, if you dig into the `\bin` or `\obj` directories, you'll see some more, such as `ASPNETCoreDemo1.runtimeconfig.json` with extra information. And, finally, if you examine the `.csproj` you'll see some additions as well:

WOW! eBook
www.wowebook.org

Remember that on the project's context menu option at the Solution Explorer you now have an option that allows you to open it directly in Visual Studio 2017.

```
 <Project Sdk="Microsoft.NET.Sdk.Web">
 <PropertyGroup>
  <TargetFramework>netcoreapp1.1</TargetFramework>
 </PropertyGroup>

 <ItemGroup>
  <Folder Include="wwwroot\" />
</ItemGroup>
<ItemGroup>
  <PackageReference Include= "Microsoft.ApplicationInsights.AspNetCore"
       Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
</ItemGroup>
</Project>
```

Not many changes, but now it indicates the use of the `wwwroot` folder and it adds the `ApplicationInsights` call. And, obviously, there's no indication of filetype, since, by default, the compiled assembly is a DLL.

WOW! eBook
www.wowebook.org

The Entry Point

Let's start with `program.cs`. Its `main()` method is similar, but it contains new middleware:

```
public static void Main(string[] args)
{
  var host = new WebHostBuilder()
  .UseKestrel()
  .UseContentRoot(Directory.GetCurrentDirectory())
  .UseIISIntegration()
  .UseStartup<Startup>()
  .UseApplicationInsights()
  .Build();
  host.Run();
}
```

The three main differences are: `UseContentRoot()`, `UseIISIntegration()`, and `UseApplicationInsights()`.

`UseContentRoot(Directory.GetCurrentDirectory())`, indicates that whenever the user requests a physical resource, that's the directory to search in. By default, it's going to point to the `wwwroot` directory.

`UseIISIntegration()` is there to indicate that IIS will be used as a reverse proxy (as we mentioned previously), and finally, `UseApplicationInsights()` helps monitor and audit your application.

As the official documentation states (https://github.com/Microsoft/ApplicationInsights-aspnetcore/wiki/Getting-Started-with-Application-Insights-for-ASP.NET-Core) it allows us to *Monitor your live ASP.NET Core applications with Visual Studio Application Insights. Application Insights is an extensible analytics platform that monitors the performance and usage of your live web applications. With the feedback you get about the performance and effectiveness of your app in the wild, you can make informed choices about the direction of the design in each development lifecycle.*

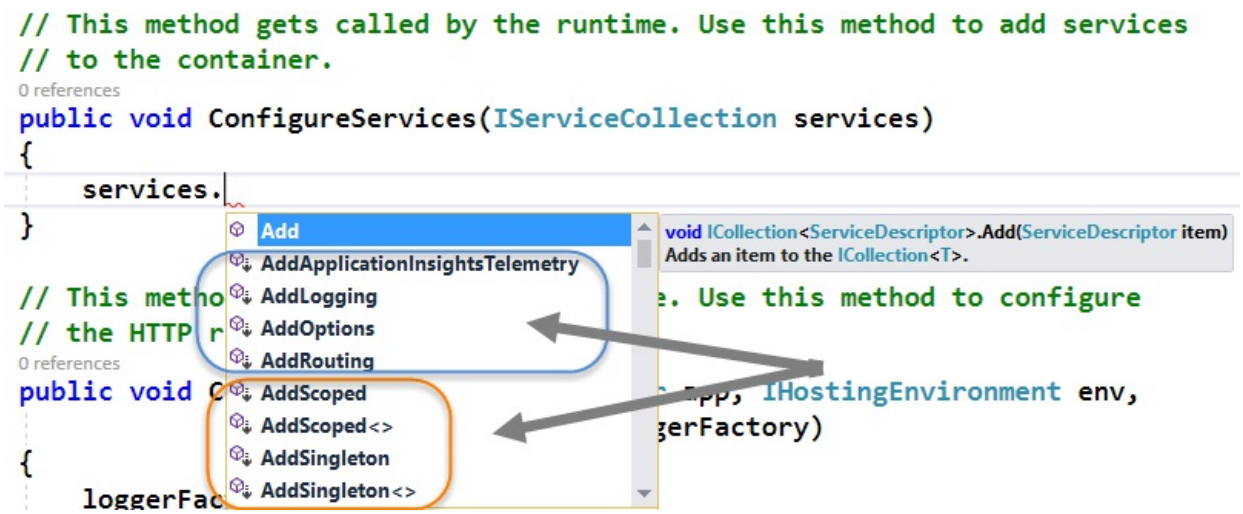
So, we have an entry point reinforced with new middleware, and thus, we can use DI from the start. Let's see what it's done in the `Startup` class (the configuration).

WOW! eBook
www.wowebook.org

The default Startup class

The first point to notice is the presence of `ConfigureServices` (even if it's empty). As you can see in the following screenshot, it allows adding distinct services to our pipeline, storing them in the `services` collection. This will be the place to register our own services as well.

Also notice that some of these methods are ready to add services with distinct lifetime configuration (`AddSingleton`, `AddScoped`, and `AddTransient`). Later, we'll see how to add services at this point to allow applications to use a given functionality, such as Database Access, and so on, using this `IServiceCollection` object that it received through Dependency Injection (see the following screenshot):



As for the `Configure()` method, this time it receives (via Dependency Injection, of course), three instances of types `IApplicationBuilder`, `IHostingEnvironment`, and `ILoggerFactory` as the following code shows:

```
// This method gets called by the runtime. Use this method to //
// configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app,
   IHostingEnvironment env, ILoggerFactory loggerFactory)
{
  loggerFactory.AddConsole();

  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.Run(async (context) =>
```

WOW! eBook
www.wowebook.org

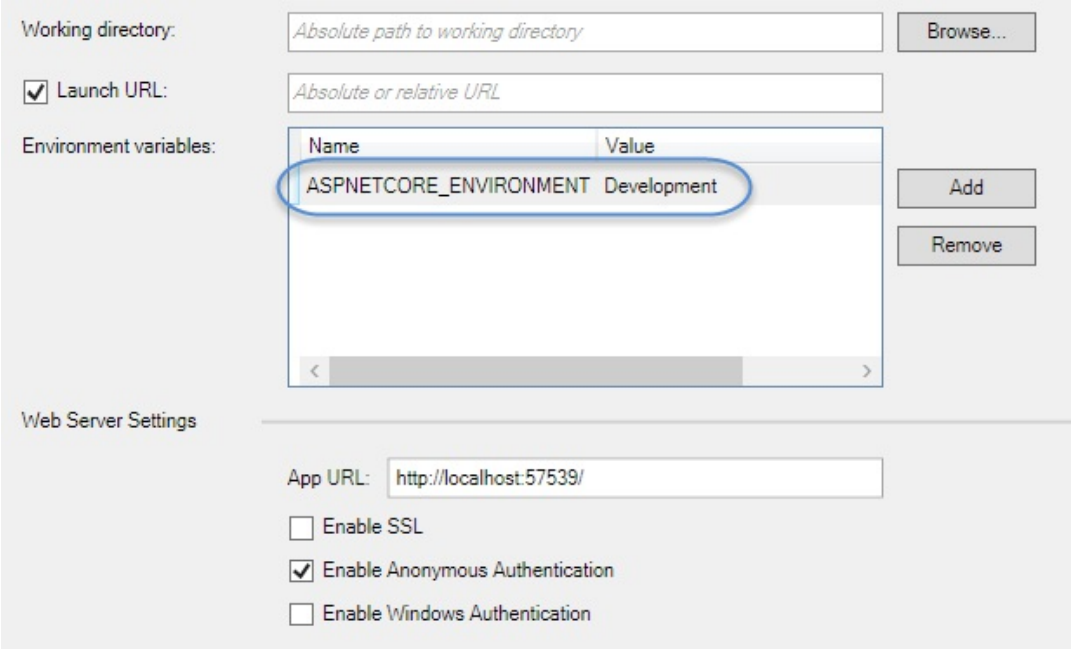
```
        {
          await context.Response.WriteAsync("Hello World!");
        });
      }
```

The first one is used at the end to launch the application, only this time using the `async/await` structure to guarantee a formal asynchronous call, returning a string.

`ILoggerFactory` is used in a similar manner as we did in `Chapter 3`, *Introducing Dependency Injection in .NET Core 2.0*, to configure messages that will be output to the console.

Finally, the `IHostingEnvironment` variable is used to check if the application is in development mode, and, in that case, to configure a developer exception page, which we can define elsewhere.

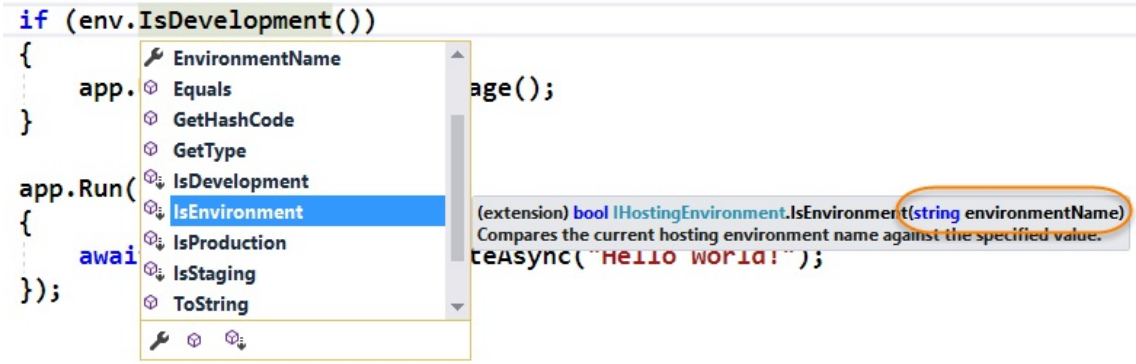
ASP.NET Core distinguishes four possible development modes: `development`, `production`, `staging`, and a fourth one `environment` that allows defining your own. That value is configurable inside the `Project/Properties/Debug` window, where you can now add distinct environment variables, as you can see in the following screenshot:



These values are read by the `IHostingEnvironment` object allowing the insertion of actions prior to the application launching process. Notice it's not a single value, but a collection, so you can add as many environment values as needed and use this object's methods to read whether some of them are true.

WOW! eBook
www.wowebook.org

You could also use something like `ASPNETCORE_ENVIRONMENT"="MyCompany` to define your own and check this value easily, using the `IsEnvironment()` method, as you can see in the following screenshot:



So, if we change the previous `development` value for a customized one, such as `PACKT`, we can modify the exit in the browser with the following:

```
app.Run(async (context) =>
{
   if (env.IsEnvironment("Packt"))
   {
    await context.Response.WriteAsync("We're in PACKT
         development mode!");
   }
   else await context.Response.WriteAsync("Hello World!");
});
```

The output will be different in this case, so we're free to configure anything and mix it with other values to have a totally customized, mode-dependent, experience:

But, there's much more in relation to DI, as we're going to see in the next sections.

WOW! eBook
www.wowebook.org

Dependency Injection in ASP.NET Core

Of course, this behavior is possible thanks to the presence of a DI Container inside the ASP.NET Core engine. The official documentation states it very clearly: *if a given type has declared that it has dependencies, and the container has been configured to provide the dependency types, it will create the dependencies as part of creating the requested instance*.

The container, in this way, manages an object's lifetime and avoids the need for hard-coded object construction.

Besides other built-in implementations, remember that ASP.NET Core provides a simple DI Container (that we already tested in `Chapter 3`, *Introducing Dependency Injection in .Net Core 2.0*), represented by the `IServiceProvider` interface.

As we mentioned, the place to configure services using that interface in this platform is the `ConfigureServices` method, which we will analyze in the following section.

WOW! eBook
www.wowebook.org

Services provided by ASP.NET Core

There's quite a large list of services available inside ASP.NET Core via Dependency Injection. The following table shows these services together with an indication of its lifetime:

Service Type	Lifetime
`Microsoft.AspNetCore.Hosting.IHostingEnvironment`	Singleton
`Microsoft.Extensions.Logging.ILoggerFactory`	Singleton
`Microsoft.Extensions.Logging.ILogger<T>`	Singleton
`Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory`	Transient
`Microsoft.AspNetCore.Http.IHttpContextFactory`	Transient
`Microsoft.Extensions.Options.IOptions<T>`	Singleton
`System.Diagnostics.DiagnosticSource`	Singleton
`System.Diagnostics.DiagnosticListener`	Singleton
`Microsoft.AspNetCore.Hosting.IStartupFilter`	Transient
`Microsoft.Extensions.ObjectPool.ObjectPoolProvider`	Singleton
`Microsoft.Extensions.Options.IConfigureOptions<T>`	Transient
`Microsoft.AspNetCore.Hosting.Server.IServer`	Singleton
`Microsoft.AspNetCore.Hosting.IStartup`	Singleton
`Microsoft.AspNetCore.Hosting.IApplicationLifetime`	Singleton

As you see, this is a pretty comprehensive list of choices, to which we have to add the functionality already available and "installable" via `Run*`, `Use*`, and `Map*` methods commented on previously.

WOW! eBook
www.wowebook.org

So, we can distinguish two flavors of method, here: those already available and ready to include at will (the ones shown in `Program/Main`), and those that you can customize (using the `Startup` class), either by addition or by creating your own classes and interfaces and adding them to the `ConfigureServices` initial process.

WOW! eBook
www.wowebook.org

The relation of services available at startup

In summary, we have several ways to include functionality via Dependency Injection in ASP.NET Core 2.0, via the Startup class:

- By creating your own constructor for the Startup class, which refers to a previously define map Interface => Class
- By referring to the service required in the `ConfigureServices` method
- By using the `Configure()` method in the ways we've seen

If you consider the methods of `Startup` in the order they are executed, the following services are at your disposal:

- Constructor: `IHostingEnvironment`, `ILoggerFactory`
- `* ConfigureServices`: `IServiceCollection`
- Configure: `IApplicationBuilder`, `IHostingEnvironment`, `ILoggerFactory`, `IApplicationLifetime`

WOW! eBook
www.wowebook.org

Identifying Services in the Web Application template

The Web Application template is much more revealing regarding the use of services and the role of Dependency Injection, so we're going to create a project equivalent to the classic ASP.NET MVC 5 available for a classic .NET Framework so we can compare both, identifying this functionality.

Once you create one of these new projects, you'll probably identify many of the elements that were present in the previous version: folders for controllers and views, Razor files defining the distinct views in the application, and so on.

But, now that we've seen some of the main changes, a review of the critical files is quite enlightening. There are no changes in `Program/Main`, but we'll find a lot in the `Startup` class. The references (`using` statements) are the same as in the basic demo, so I'm omitting the code here.

The most interesting come inside the `Startup` class itself:

```
public class Startup
{
  public Startup(IHostingEnvironment env)
  {
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false,
                     reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                     optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
  }

  public IConfigurationRoot Configuration { get; }

  // This method gets called by the runtime. Use this method to
  // add services to the container.
  public void ConfigureServices(IServiceCollection services)
  {
    // Add framework services.
    services.AddMvc();
  }
```

Up to this point, the default template only initializes the MVC engine, considering it

WOW! eBook
www.wowebook.org

an extra service, totally optional to the user.

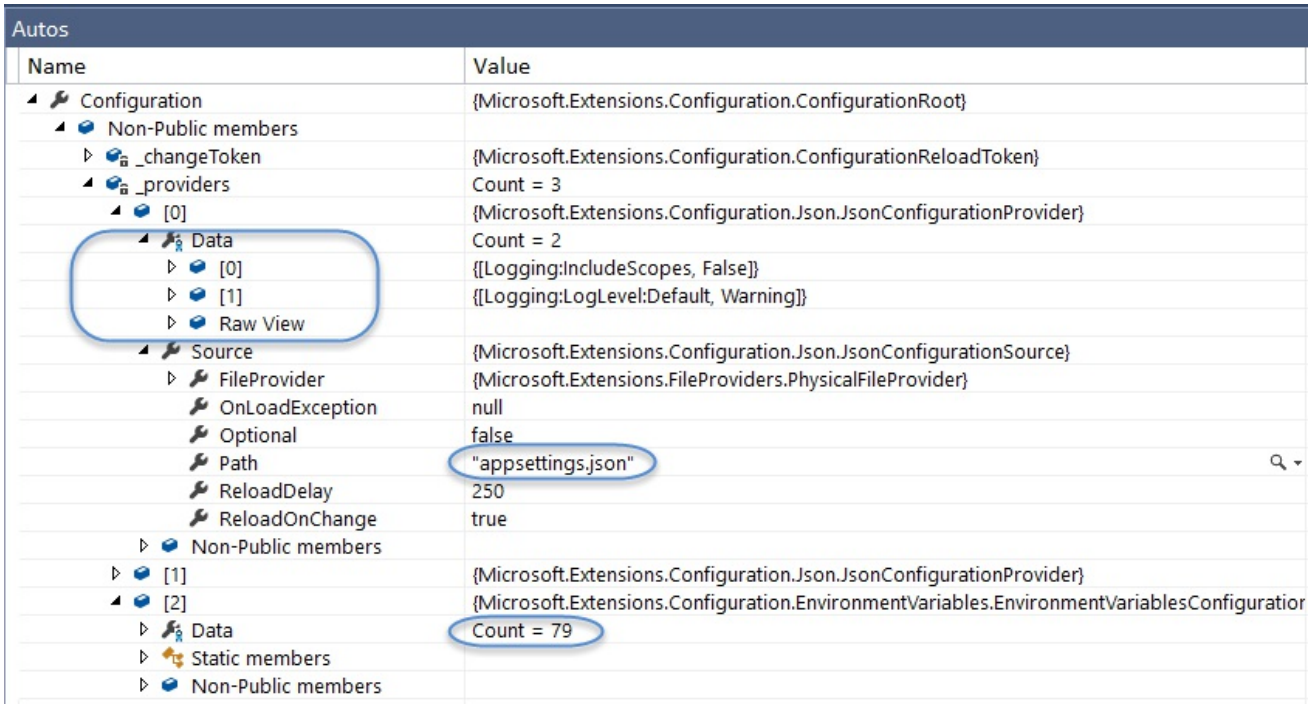
Further on, we'll see how to use this method to register and configure other built-in or customized services:

```
// This method gets called by the runtime. Use this method to
// configure  the HTTP request pipeline.
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute( name: "default",
            template:{controller=Home}/{action=Index}/{id?}");
    });
 }
```

First of all, we now have a constructor and a read-only private property (`Configuration`) of type `IConfigurationRoot`. Well, it happens that the `builder.Build()` method called at the end of the constructor, is of this type and provides a convenient way to include and access all configuration information loaded from several `.json` files in the distinct application's folders.

All that information is collected at the very beginning of the process of initialization, so whatever has to be read from the outside is ready before `ConfigureServices` and `Configure` come into play (see the following screenshot showing the value of `Configuration` after loading):

WOW! eBook
www.wowebook.org



As you see, the `ConfigurationBuilder` class is created in the first sentence of the constructor, and takes care of loading and reading the content of all these `.json` files, and adding environment variables. This allows further access to any `json`-defined value, and the ability to tune-up the application with external information.

With all that info collected, the `ConfigureServices` method comes and registers a set of services, such as MVC, by adding a `UseMvc` call (yes, it's not available by default, and has to be explicitly added to the pipeline if we want to use that architecture).

As you've seen along these samples, this ASP.NET middleware, such as MVC, follows a convention of using a single `Add`*`ServiceName`* extension method to register all of the services required by that feature. Also, notice that this call only adds the service to the collection of services, but it does not configure it (that comes later).

Then, the `Configure()` method comes in. It first recovers information from the `Configuration` object (`Logging` section) and adds debugging capabilities in the way we saw in the previous chapter.

It follows with error handling, by checking if we are in development mode or another one (it also enables `BrowserLink` capabilities), and goes on with a call to `UseStaticFiles()`, to allow recovering and serving local files (you guessed it, it's not available by default, either).

WOW! eBook
www.wowebook.org

Notice that this feature can be configured to point someplace else (relative request path) on the server (see the screenshot):

```
app.UseStaticFiles();
▲ 3 of 3 ▼ (extension) IApplicationBuilder IApplicationBuilder.UseStaticFiles(string requestPath)
           Enables static file serving for the given request path
           requestPath: The relative request path.
{
```

The last step is configuring MVC routes with a call to `UseMvc()`, in which the route's configuration is provided. Notice how different this approach is in comparison to classical ASP.NET MVC 4/5, in which you would register these routes in a different file. However, the syntax is similar, only this time you define it via lambda expressions.

As you can deduct from the code, the first lambda argument is given by Dependency Injection and it's of type `IRootBuilder`. Take a look at the Intellisense information provided by the IDE to check this feature, which is shown in the following screenshot:

```
app.UseMvc(routes =>
                    (parameter) Microsoft.AspNetCore.Routing.IRouteBuilder routes
{
```

The default route is then configured to point to the `HomeController` class and the `Index` action method if no other combination is supplied by the request (the syntax has also been a bit simplified).

So, our conclusion after this trip through the `Startup` class is that the most important architectural aspects of ASP.NET Core and its MVC variant are provided via Dependency Injection, and that it is the developer's duty to tune up their application in this manner having the ability to include/exclude only those pieces that are needed by the application, with the minimum overload.

WOW! eBook
www.wowebook.org

Using Dependency Injection in ASP.NET MVC Views

Steve Smith proposes a demo (http://ardalis.com/how-to-list-all-services-available-to-an-asp-net-core-app) that can clarify the total of services available at a given moment.

This gave me an idea for another demo about how to include the namespace `Microsoft.Extensions.DependencyInjection` inside a view, using MVC to obtain a list of all services available. So let's start with the default template we just analyzed, and make the proper modifications.

First, if we want to integrate the list as a new option inside the main menu of the Web Application template, we need to add a new link inside the `_Layout.cshtml` file that points to the corresponding view that will show all our services. That's quite straightforward (notice that there's no `ActionLinks` here as in the previous version):

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
        <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
        <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
        <li><a asp-area="" asp-controller="Home" asp-action="ServicesAvailable">List of Services</a></li>
    </ul>
</div>
```

With this new line, we're creating a new application menu element and the corresponding view (later named `ServicesAvailable`) will take advantage of Bootstrap classes loaded in the `_Layout.cshtml` header, to format the output and make it more readable.

Now, if we think about the controller (`HomeController` in this case), we can add a new action method, following the syntax of the other methods, using the `ViewData` object to pass the required information to our new view.

The information we need is stored in the instance of the `IServiceCollection` object defined inside `Startup.cs`, and we want to make it available for the controller so we can later assign it to our `ViewData` messenger.

Let's revisit the `Startup`, and make a few changes (not many). Actually, only these are

WOW! eBook
www.wowebook.org

required:

```
public static IServiceCollection _services { get; private set; }
public void ConfigureServices(IServiceCollection services)
{
  // Add framework services.
  services.AddMvc();
  _services = services;
}
```

We create a public static variable (`_services`) to hold all services once they have been configured and assign the `services` variable to it with the sole purpose of accessing its content from the controller.

Now, back to the `HomeController`, we can add a new controller with this code:

```
public IActionResult ServicesAvailable()
{
  ViewData["Services"] = Startup._services;
  return View();
}
```

With these few lines, our services are now available inside the controller, and we can pass them to the view in a very simple way (notice there's no data model here since it's just a simple collection of data and the `ViewData` object serves just fine for this purpose).

Finally, the last step will be to add the `ServicesAvailable` view. The code will be the following (I'll start with the header's explanation):

```
@using Microsoft.Extensions.DependencyInjection;
@{
  ViewData["Title"] = "Services Available";
  var _services = @ViewData["Services"];
}
<h2>@ViewData["Title"]</h2>
<h1>All Services</h1>
```

First, recall that when we make a reference to a namespace inside a view, the `using` sentence shall not belong to a block of code. Instead, it will be a standalone sentence headed by the `@` sign (we need the `DependencyInjection` namespace to cast the information passed inside the `ViewData` object into a real `IServiceCollection` object).

Next, we recover that info inside a variable (that will be available locally in the whole view). Observe that I'm using the `var` keyword instead of the interface name because otherwise the compiler will complain. It's easier to do it this way and make the casting later in the code.

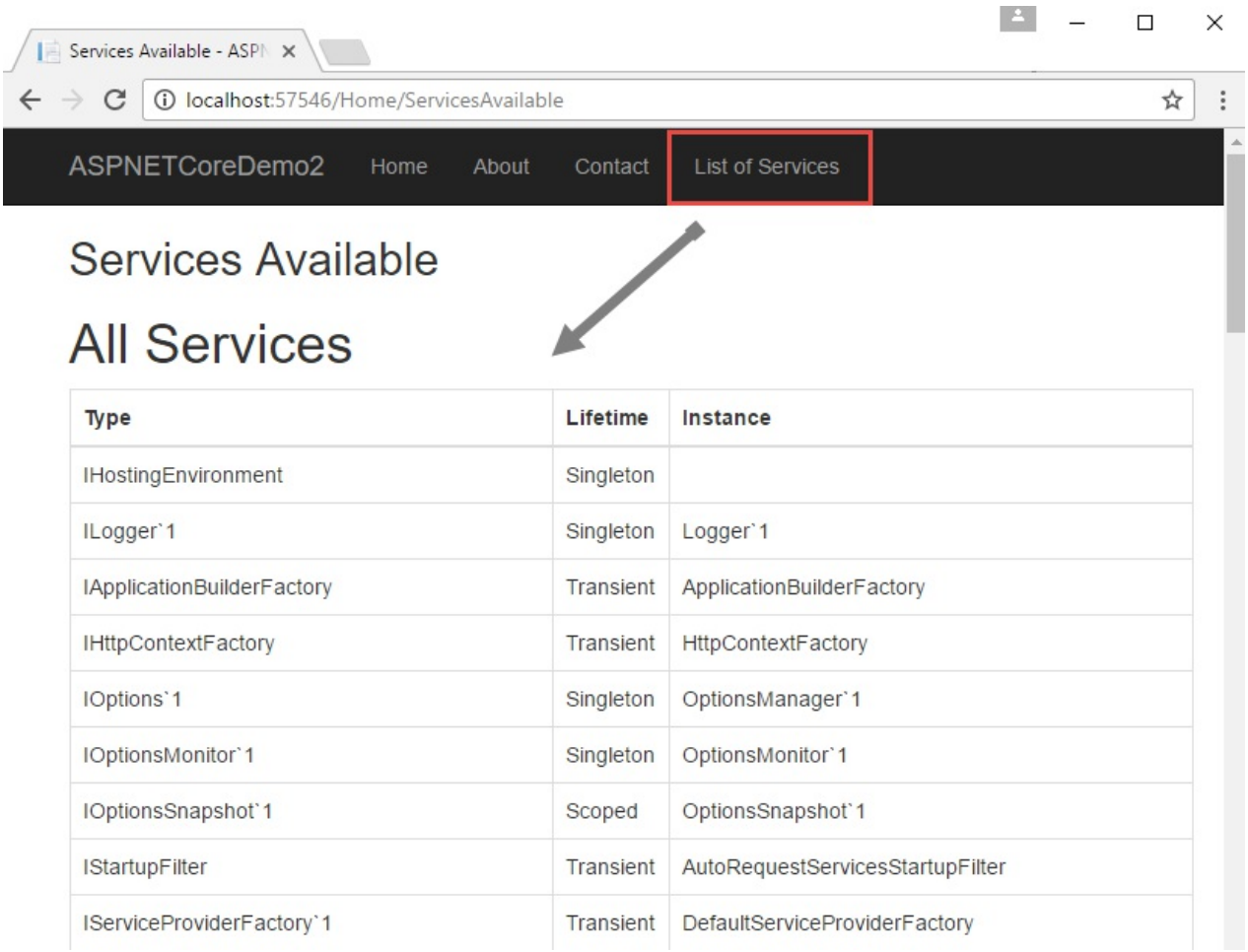
WOW! eBook
www.wowebook.org

Finally, we'll use a table to present some information hold in three of the services of `IServiceCollection` (called `_services` in the view). Notice also the casting is made here with the `as` operator to get a real `IServiceCollection` object:

```
<table class="table table-bordered">
<thead>
    <tr>
        <th>Type</th>
        <th>Lifetime</th>
        <th>Instance</th>
    </tr>
</thead>
<tbody>
 @foreach (var svc in _services as IServiceCollection)
 {
   <tr>
   <td>@svc.ServiceType.Name</td>
    <td>@svc.Lifetime</td>
    <td>@svc.ImplementationType?.Name</td>
    </tr>
 }
</tbody>
</table>
```

And that's all! Now we can launch the application and select the new menu element *List of Services*. A new view will show up containing a, pretty long list of services available in this architecture, showing at the same time how we can make use of the `DependencyInjection` namespace (or any other) inside a view (see the following screenshot):

WOW! eBook
www.wowebook.org



Type	Lifetime	Instance
IHostingEnvironment	Singleton	
ILogger`1	Singleton	Logger`1
IApplicationBuilderFactory	Transient	ApplicationBuilderFactory
IHttpContextFactory	Transient	HttpContextFactory
IOptions`1	Singleton	OptionsManager`1
IOptionsMonitor`1	Singleton	OptionsMonitor`1
IOptionsSnapshot`1	Scoped	OptionsSnapshot`1
IStartupFilter	Transient	AutoRequestServicesStartupFilter
IServiceProviderFactory`1	Transient	DefaultServiceProviderFactory

WOW! eBook
www.wowebook.org

Garbage Collection and Custom Services

In relation to Garbage Collection behavior, ASP.NET MVC automates some operations and leaves others to the user's discretion.

The main rule is as follows: if you register a service and call its constructor inside the process, it's your obligation to dispose of that object, otherwise if the container is in charge of creating the object, it will call `Dispose()` on any object it implements.

In practice, imagine we have a couple of services (`Service1` and `Service2`) located in a folder (`Services`), and that both implement the `Disposable` interface. To simplify this part, I collapsed the code, so I'm just showing the relevant lines to understand this idea (see the following screenshot):

```
namespace ASPNETCoreDisposeDemo.Services
{
    1 reference
    public class Service1 : IDisposable {
        IDisposable Support
    }
    1 reference
    public class Service2 : IDisposable {
        IDisposable Support
    }
}
```

The behavior will be different depending on the instantiation of each service (I've commented on each case so you'll see the difference):

```
publicvoidConfigureServices(IServiceCollectionservices)
{
 // Add framework services
 services.AddMvc();
 // container will create the instance(s) of these types
 // and will dispose themservices
 AddScoped<Service1>();
 // Here, the container did not create instance (we do it)
 // so it will NOT dispose itservices
 AddSingleton(newService2());
}
```

In all, it is a pretty convenient way to manage garbage collection of services.

WOW! eBook
www.wowebook.org

Using Custom Services through Dependency Injection

With this architecture, it is very easy to create any class belonging to our data model, or any other part of the application, and have it ready and available for Dependency Injection, with all the advantages linked to it.

For example, imagine that we want some adornment in one of our views in the form of a random sentence (about programmers and software engineers, for instance), and that we would like the classes that hold the information to be treated as a service, via Dependency Injection, so we can use that content in distinct application's views.

We can rename the classes used in the previous sample as `ProgrammerSentenceSvc` and `EngineerSentenceSvc` and add some very simple information: a `List<string>` of sentences that will be used later inside our controller to randomly retrieve a couple of these sentences and pass them to the corresponding view.

So let's modify our services to hold the required information (I'm just showing these captures, to focus on the relevant code):

```
public class ProgrammerSentenceSvc : IDisposable {
    public List<string> programmersSentences = new List<string>()
    {
        "The trouble with programmers is that you can never tell what" +
        " a programmer is doing until it’s too late",
        "Walking on water and developing software from a specification" +
        " are easy if both are frozen.",
        "A programming language is low level when its programs" + "" +
        " require attention to the irrelevant.",
        "In theory, theory and practice are the same. In practice, they’re not.",
        "Perl – The only language that looks the same before" +
        " and after RSA encryption."
    };

    IDisposable Support
}
```

WOW! eBook
www.wowebook.org

```
public class EngineerSentenceSvc : IDisposable {
    public List<string> engineersSentences = new List<string>()
    {
        "The best way to get a project done faster is to start sooner",
        "Even the best planning is not so omniscient" +
        " as to get it right the first time.",
        "Any fool can write code that a computer can " +
        "understand. Good programmers write code that" +
        "humans can understand.",
        "Program testing can be used to show the presence" +
        " of bugs, but never to show their absence!",
        "Prolific programmers contribute to certain disaster."
    };
    IDisposable Support
}
```

The next step will be to register these classes in the `Startup` class via `ConfigureServices`, as we've seen before (I'm using two distinct syntaxes just to show):

```
publicvoidConfigureServices(IServiceCollectionservices)
{
 // Add framework services.
 services.AddMvc();
 // container will create the instance(s) of these types
 // and will dispose them
 services.AddScoped<ProgrammerSentenceSvc>();
 // Here, the container did not create instance (we do it)
 // so it will NOT dispose it
 services.AddSingleton(newEngineerSentenceSvc());
}
```

That's all we need to have our services available inside any controller. So, let's revisit our `HomeController`, and add the following action method (remember that we have to make a reference to our Services namespace, via `using ASPNETCoreDisposeDemo.Services;`):

```
publicIActionResultSentences(ProgrammerSentenceSvcsvc,
    EngineerSentenceSvcsvc2
 {
  Randomrnd = newRandom();ViewData["ProgSentence"] =
      svc.programmersSentences[rnd.Next(1,5)];
  ViewData["EngSentence"] = svc2.engineersSentences[rnd.Next(1,5)];
  returnView();
 }
```

And that's it! Any service registered in that manner is automatically available inside a controller via Dependency Injection, merely by making a reference to it as a parameter of the corresponding action method.

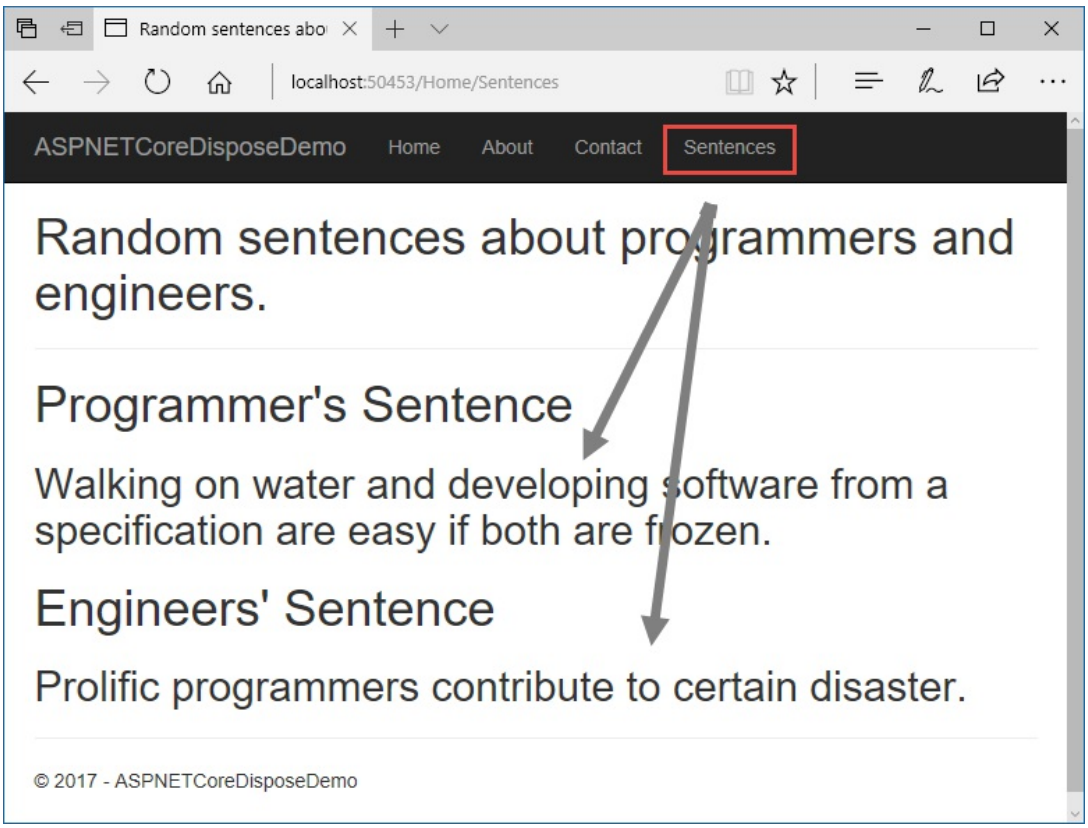
The last step is to create the view named `Sentences` to recover the information and present it to the user:

```
@{ViewData["Title"] = "Random sentences about programmers
```

WOW! eBook
www.wowebook.org

```
        and engineers";
    }
     <h1>@ViewData["Title"].</h1>
     < hr />
     < h1 > Programmer's Sentence </ h1 >
     <h2>@ViewData["ProgSentence"]</h2>
     < h1 > Engineers' Sentence </ h1 >
     <h2>@ViewData["EngSentence"]</h2>
```

If we have added (just like before), a new link next to the default menu pointing to the name of this action method, we'll see the following output:



As you see, the output is the expected one, and the general architecture of the application with respect to the MVC model is quite similar to previous versions.

WOW! eBook
www.wowebook.org

Services and data management

Although data management is not the target of this book, I'd like to mention that the architecture proposed when accessing data inside ASP.NET Core MVC applications is relatively alike.

To test this feature and highlight those parts directly related to Dependency Injection, follow the initial instructions at https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/existing-db. That creates a very simple database using SQLLocalDb called `Blogging`. It creates a `Blogs` table and adds three registers, just to have some data to work with. You'll also find a link to the complete sample it proposes using several approaches: Database First, New Database, and so on.

What I would like to point out here is those fragments of code that imply the use of DI inside that solution.

In the `Models'` application folder you'll find the definition of the model based on that database in a similar way to the usual result when you scaffold an existing database using classic Entity Framework. The main class inherits from `DbContext`, defines public virtual properties pointing to the existing entities, and names the `DBContext` instance `BloggingContext`.

Consequently, the first step is to register that context inside the `ConfigureServices` method, for which it does the following:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    var connection = @"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;";
    services.AddDbContext<BloggingContext>(options => options.UseSqlServer(connection));
}
```

As you see, the connection string is defined directly so that it points to the newly created database (`Blogging`).

Next, the services variable is populated with a new entry via the generic method `AddDbContext<BlogginContext>`. And, to link that `DbContext` to the `Blogging` database and `Action` delegate is passed to the method, allowing it to override the `OnConfiguring`

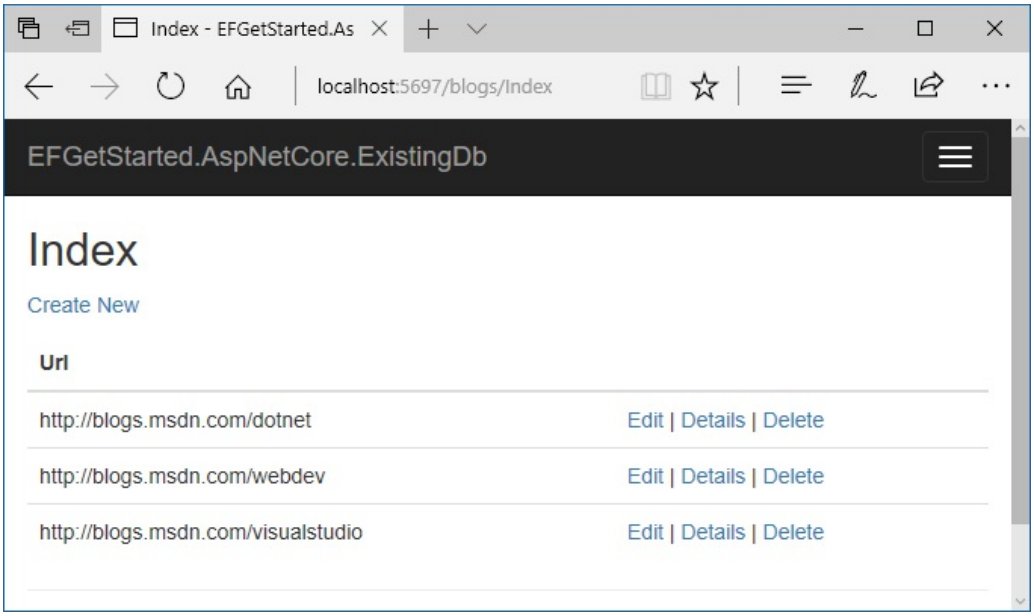
WOW! eBook
www.wowebook.org

method of the `BloggingContext` class. In this manner, the real connection configuration can be delayed until this method is called.

Finally, how does the `BloggingController` class access this `BloggingContext`? By Dependency Injection in its constructor. It operates in a similar way to what we did in the previous sample:

```
public class BlogsController : Controller
{
    private readonly BloggingContext _context;

    0 references | 0 exceptions
    public BlogsController(BloggingContext context)
    {
        _context = context;
    }
```

The output shows the three entries loaded in the `Blogs` table, just as expected, with the typical CRUD options, presenting the information via Razor views, but it's interesting to highlight the role of DI in this architecture as well (see the following screenshot):



Again, you've seen how DI is also present in .NET Core-related platforms, such as Entity Framework Core.

However, there's an interesting point we haven't covered yet. We saw how to make a reference to the `DependencyInjection Namespace` inside a view, but we were not really using DI. Let's see how to use the new `@inject` directive inside a view with a simple demo.

WOW! eBook
www.wowebook.org

Using Dependency Injection inside Views

For completeness, I'm going to explain a simple demo of how to use DI inside a Razor view. It's a quite interesting feature and it can be useful to access data, service-related information, or even your own Razor helpers.

The idea is to create another view that is able to access the same data as our previous `Services` View, but this time without implying the controller in the business logic. Besides that, I'll create a Razor helper just to show how we can do both tasks thanks to the `@inject` directive.

First, inside a new folder (`Helpers`), let's create a simple helper that provides some kind of information (for instance the current system's time):

```
namespaceASPNETCoreDisposeDemo.Helpers{
  publicclassDateTimeHelpers
  {
    publicDateTimeHelpers()
    {
      LocalTime = DateTime.Now.TimeOfDay.ToString();
    }
      publicstringLocalTime { get; privateset; }
  }
}
```

With this new service in place, we need to register it as we did with the `Sentences*` services. The new version of the `ConfigureServices` method will look as follows:

```
publicvoidConfigureServices(IServiceCollectionservices)
{
  // Add framework services.
  services.AddMvc();
  // container will create the instance(s) of these types
  // and will dispose them
  services.AddScoped<ProgrammerSentenceSvc>();
  // Here, the container did not create instance (we do it)
  // so it will NOT dispose it
  services.AddSingleton(newEngineerSentenceSvc());
  services.AddTransient<DateTimeHelpers>();
}
```

This is all we need on the business logic side since we can use the existing services, only in a different way. So, I'll create another action method as part of the

WOW! eBook
www.wowebook.org

`HomeController` named `ServicesDI`, and try to replicate the previous functionality.

We can also use the `[Route("key")]` attribute to redirect URL queries to this action method. Actually, it's even simpler than the others:

```
[Route("SentencesDI")]publicIActionResultSentencesDI()
{
   returnView();
}
```

The rest of the programming logic is deferred to the view itself. Thus, the `SentencesDI` View will have to reference the namespaces related to the services it is going to use, and declare any service required:

```
@usingASPNETCoreDisposeDemo.Helpers
@usingASPNETCoreDisposeDemo.Services
@inject DateTimeHelpersTimeHelpers
@inject ProgrammerSentenceSvcPSentences
@inject EngineerSentenceSvcESentences
@{
  Randomrnd = newRandom();
   ViewData["Title"] = "Random sentences obtained via
        Dependency Injection";
 }
 <h1>@ViewData["Title"]</h1>
 <h3>Local Time: @TimeHelpers.LocalTime</h3>
 < hr />
 < h2 > Programmer's Sentences (DI) </ h2 >
 <h3>@PSentences.programmersSentences[rnd.Next(1,5)]</h3>
 < h2 > Engineers' Sentences (DI) </ h2 >
 <h3>@ESentences.engineersSentences[rnd.Next(1,5)]</h3>
```

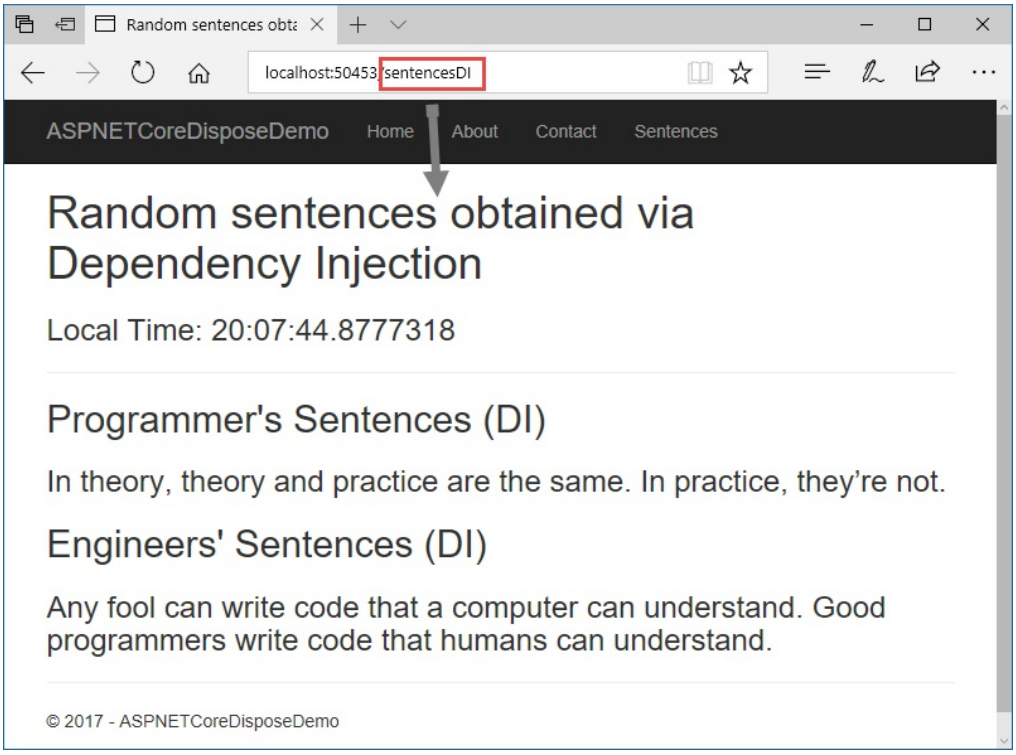
As you can see, the rest of the code is pretty self-explanatory. Once services are registered and references made, we can use the `@inject` directive to instruct the DI container about the resources that our view is going to need. The syntax is:

```
@inject [Type] [Name/Alias]
```

In this way, any functionality related to a service that is injected in a view can access the data inside the service without having to process it in the controller. You might say that this breaks in some way the architecture of the MVC foundations, but there are some circumstances in which it might be interesting to separate certain data from the controller if it is only related to a particular view.

And, surely, you'll find more useful possibilities. By the way, you can type `localhost:[port]/ServicesDI` in the browser to get the corresponding output:

WOW! eBook
www.wowebook.org



In short, another feature related to Dependency Injection, this time inside the Razor views, that we can use inside the ASP.NET Core platform.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we focused on ASP.NET Core and Dependency Injection, analyzing how the whole architecture and configuration process is based on the internal DI Container of ASP.NET Core.

We also saw how to move from a simple Console app to an ASP.NET Core app, and finally, how the distinct servers for development and production can be tuned up.

Then, we analyzed the main templates offered by Visual Studio 2017 in relation to ASP.NET Core apps and reviewed how they used DI in order to configure and manage the required information and functionality.

Finally, we saw how to use our own custom services and integrate them inside ASP.NET Core using DI, either in the controllers or in the Razor Views.

In `Chapter 5`, *Object Composition*, we will analyze Object Composition and its use within DI contexts.

WOW! eBook
www.wowebook.org

Object Composition

In `Chapter 4`, *Dependency Injection in ASP.NET Core*, we got an insight into Dependency Injection with .NET Core and ASP.NET Core with default DI Container. We explored how to apply DI to different components of the application like controllers and views. Now it's time to deep dive into the actual fundamentals behind Dependency Injection.

Before moving forward with the main topic, we need to first understand why do we even care to read about this topic. The programming world is surrounded with objects and their interactions. We achieve certain solutions or build features getting help from classes that we generate for two basic reasons that are **code reuse** and **maintainability**.

Now you might ask me why create classes at all! Yes, I second you, unless you ask me to alter code after some days. Then, even I can't help you because that will be a nightmare for me. And you know why? That is because I might have to repeat the codes (you have already written) in my new classes.

Suppose you have a class named `Customer` with the properties `CustomerId`, `FirstName`, `LastName`, `Email`, `MobileNumber`, `Address1`, `Address2`, `City` and so on. I come in and start working on another entity named as `Seller` with the properties `SellerId`, `FirstName`, `LastName`, `Email`, `MobileNumber`, `Address1`, `Address2`, `City` and so on.

I think you are smart enough to identify the problem. We can see the properties which are *italicised* are getting repeated. One solution to this problem would be to create another class, something named `User` with all those common properties. Then, the `Customer` and `Seller` classes become its children. Thus, we will achieve a simple design to deal with future entities as well. I can easily reuse the `User` class to create some other child class, which would inherit the common properties if needed.

Therefore, this chapter plays an important role in helping you grab these relationships. Designing a complex class with smaller flexible and reusable classes is an essential part of a software project, which will be the focused of this chapter. The following are the key topics that we will cover:

- Relationships

WOW! eBook
www.wowebook.org

- Composition
- Aggregation
- Association
- Inheritance
- Composition over Inheritance
- The importance of Object Composition in Dependency Injection
- Object Composition in .NET Core 2.0 Console and MVC Apps

WOW! eBook
www.wowebook.org

Understanding object relationships

Let's try to understand object relationships by first considering human relationships. The examples may not be the aptest but if we want to learn about object relationships, then why not look into it?

- You depend on your parents till you get a job
- You have a wife and two children
- Trees have blossom and leaves on them
- A motherboard is part of a computer

All these relationships possess unique characteristics. Let me simplify my statement. You depend on your parents. However, if they become jobless, you are not going to die. You will find some way to deal with the situation. On the contrary, if a tree dies, its blossom and leaves will die eventually. The relationship between the tree and its parts are tightly coupled. After the tree gets a life (instantiated), within a short period of time, its parts come to life as well. A motherboard without a computer is useless. It comes into play when it becomes a component of a computer.

Let's consider the following code snippet to understand the relationship between classes:

```
class Organisation
{
  public Organisation() { }
  public string Name { get; set; }
  public string OfficialFacebookLink { get; set; }
}

class Packt : Organisation
{
  public Packt() { }
  public int TotalBooksPublished { get; set; }

  public void PrintPacktInfo()
  {
    Console.WriteLine($"This is {Name}!\n" +
        $"Our official facebook page link is
          {OfficialFacebookLink}.\n" +
        $"We have published {TotalBooksPublished} books.\n");

    Account account = new Account();
    account.PrintAcountInfo(1, "Packt Account");
  }
}
```

WOW! eBook
www.wowebook.org

```
public class Account
{
  public int AccountId { get; set; }
  public string AccountName { get; set; }

  public void PrintAcountInfo(int accId, string accName)
  {
    Console.WriteLine("Account Id: " + accId + "
        and Account Name: " + accName);
  }
}
```

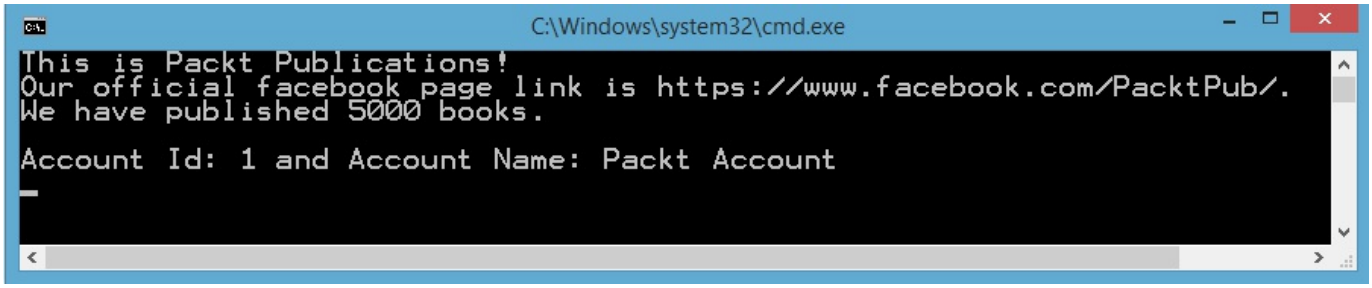
We have one `Organisation` class and `Packt`, being an organization, derives the parent class `Organisation`. This relationship is denoted as *is a* relationship because `Packt` is an organization. `Account` is a class, which can become *a part of* a `Packt` class. Thus, there is another relationship between `Packt` and `Account`. The name of the relationship is *a part of*.

Notice the method `PrintPacktInfo()` inside `Packt` class, which prints all information regarding `Packt`. However, that is not all, because you can see there is an `Account` class instance generated inside the method by which we are able to print the account information of `Packt`.

The `Main` method looks as follows, where we create an instance of `Packt` and provide any necessary details through properties and then call `PrintPacktInfo()`:

```
static void Main(string[] args)
{
  Packt packt = new Packt
  {
     Name = "Packt Publications",
     OfficialFacebookLink = "https://www.facebook.com/PacktPub/",
     TotalBooksPublished = 5000
  };

  packt.PrintPacktInfo(); // Prints the Account information.
  Console.ReadKey();
}
```

The code produces the following output:



WOW! eBook
www.wowebook.org

The important thing to take away from this example is the way the `Packt` class depends on the `Account` class. `Dependency` is generated inside the `Packt` class.

Like the preceding one, we can find patterns, relationships, and hierarchies in programming. Let's investigate these in more detail and learn how to improve code reusability and elasticity of classes.

WOW! eBook
www.wowebook.org

Object Composition

When I searched for the word composition in Google, the first thing I saw was:

the nature of something's ingredients or constituents; the way in which a whole or mixture is made up.

Now it's very simple to guess what Object Composition would be. Objects collectively mix themselves to constitute (be a part of) complex objects.

A simple real life example would be a car whose whole body consists of different types of components like engine, break, gear, battery, door, etc. Thus, these parts are actually the building blocks of the car and are composed in a very innovative way by the maker of the car.

Likewise, as we discussed, the `Account` class reference inside the `Packt` class method in the last section, which generates a relationship among them. We can consider that as a dependency too, as we can't execute the method of the `Packt` class without an instance of `Account` class. Clearly, we can say that the `Packt` object composes itself with the help of the `Account` instance.

Did you notice the bracketed phrase in the first paragraph *be a part of*? Confused! Let's discuss again and get hold of this phrase. In other words, this phrase can also be represented as *has a,* if I form the sentence from the complex object point of view. Refer the lines below.

- A computer *has a* keyboard. (Keyboard *is a part of* computer)
- A car *has an* engine. (Engine *is a part of* car)

It's pretty clear now, as you can see how these complex objects are made up of small objects conceptualizing Object Composition.

As you already know, different fundamental data types like `int`, `string`, boolean, and so on, or other class types, can be packaged into one structure or another class type, therefore, classes are often considered composite types.

The most important benefit of applying this concept to your projects is to get more

WOW! eBook
www.wowebook.org

easily manageable parts. Not only does it reduces the complexity of the application, but it also helps us to write code faster. Another noticeable advantage is code reuse, which leads to fewer errors as you will be using codes which have already been verified after testing.

WOW! eBook
www.wowebook.org

Types of Object Composition

There are two subtypes of Object Composition, **Composition** and **aggregation**. Let's discuss one by one.

WOW! eBook
www.wowebook.org

Composition

A Composition is a way through which you can bind objects together. One object can contain another object of the same/another class as a building block.

In other words, in our last example, `Packt` was dependent upon the `Account` class for its operation. The instance is created by the `Packt` class, giving life to it and then operated some function with the instance. You can add another class like `Packt` and do the same using an `Account` instance. Thus, you are trying to compose the objects to form a more complex one, which enables us to execute behavior (methods) of all composed/part objects with one composite object.

The following are the relationships, which is satisfied by the object and it's member or part, qualify as a Composition.

- **The part (member) is a component of the object (class)**: As we have already discussed, the part or smaller class should be a part of the bigger complex class. For example, your kidney is a part of your body.
- **The part (member) can only belong to one object (class) at a time**: If the smaller class is referred to the complex class at a certain period, then it can't be a part of other classes at the same time. For example, your kidney, which is a part of your body, can't be a part of someone else's body at the same time.
- **The part (member) has its existence managed by the object (class)**: The object is responsible for the existence of the parts in a Composition relationship. In simple words, the part is created when the object is created and destroyed when the object is destroyed. This means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the kidney is created too. When a person's body is destroyed, their kidney is destroyed too. Because of this, the Composition is sometimes called a **Death Relationship**.
- **The part (member) does not know about the existence of the object (class):** A particular part in a Composition doesn't know about the existence of the whole object. Your kidney is unaware that it is part of a larger structure but works as expected. This is called a **Unidirectional Relationship**. For example, the body knows about the kidney, but not the other way around.

If you are thinking that as body parts can be transferable, why not

WOW! eBook
www.wowebook.org

member classes, then you are assuming right. Member classes can also be transferred. Thus, the new larger class is now the owner of the member class and the member class is no more related to the previous owner unless that is transferred again.

WOW! eBook
www.wowebook.org

Consider an example

Our favorite `Student` class:

```
public class Student
{
  private int Id { get; set; }
  private string Name { get; set; }
  private DateTime Dob { get; set; }
  private Address Address { get; set; }
  private ICollection<Book> Books { get; set; }

  public void PrintStudent()
  {
    Console.WriteLine("Student: " + Name);
    Console.WriteLine("City: " + Address.City + "");
    Console.WriteLine("-----------------------");
  }
}
```

Come on, don't be so surprised and please don't scold me now. I know this looks like a very basic class, but that is what Composition is all about, in short. Don't believe me? Alright, allow me to match those relationships with this `Student` class:

- **Rule 1: Part of a complex class**: You can see the class members which are of different types such as `Integer`, `string`, `DateTime`, `Class` and `List<Class>` type. `Integer`, `string`, and `DateTime` are the data types already defined inside the `System` namespace of .NET Framework, whereas the `Address` and `Book` classes are user-defined classes. All are part of a complex class `Student`. Thus, the first condition is satisfied.
- **Rule 2: Members should belong to one object**: If I create an instance of the `Student` class, with a constructor, the members will only belong to the student object at that time. It can't be a member of another instance. Moreover, the members are private, which prevents them being used by any other class.

```
Student student = new Student(1, "Bhagirathi Panda",
  new DateTime(1990, 4, 23));
```

The constructor will look like the following:

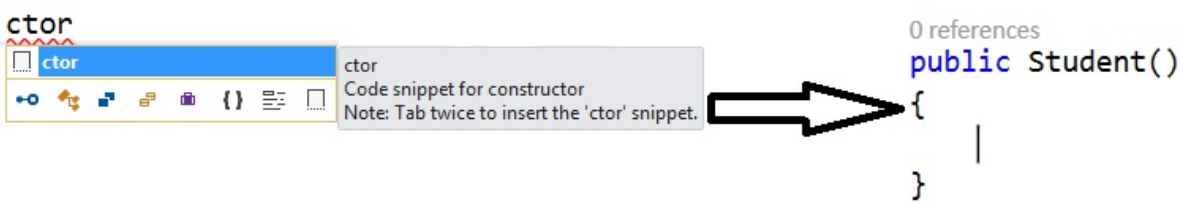
```
public Student(int id, string name, DateTime dob)
{
  Id = id;
  Name = name;
  Dob = dob;
}
```

WOW! eBook
www.wowebook.org

- **Rule 3: Members gets a life by the complex class (Death Relationship)**: As you can see, members don't have any existence until we instantiate the `Student` class and they get destroyed when the object is dead; proving our rule 3.
- **Rule 4: Members are unaware of the existence of the complex object (Unidirectional Relationship):** The members are very obedient. They just store whatever value is assigned to them and don't even bother about who is assigning them for what reasons. The complex instance is their parent but these members behave like orphans and don't recognize it. Likewise, we don't mind what they are doing because our rule 4 is proved.

Another important thing to note here is that the complex class can have a multiplicative member like `List<Book> Books`.

*Do you know the shortcut to create a constructor? Just type **ctor** at the line you want the constructor and then hit the Tab key two times. You will see a constructor block available for you. Moreover, Visual Studio tooltip tells y handle the command:*



WOW! eBook
www.wowebook.org

Contrasting features of Composition

The parts are created with the composite class's creation. That means the composite class is responsible for the creation. Also, the destruction of the part depends upon the destruction of their creator composite class. However, rules are meant to be broken and that happens in the case of Composition too.

Consider the following scenarios:

- The creation of a part is postponed till it is actually used. For instance, our `Student` class doesn't create the list of books until the user or any other method assigns some data to it.
- A Composition assigns the responsibility of destructing the part to some other object. We are already aware of such routine named **Garbage Collector**, who manages the destruction of unused objects from time to time.

Composition acts like a guardian, where all care of the members is taken by the composite class. Creation, assignment, and destruction; all are mostly managed by the composite class.

WOW! eBook
www.wowebook.org

Why are subclasses inside Composition?

There is always a decision you need to take while you program. If you closely analyze our `Student` class, you will realize a few things. The properties of the `Address` class be can directly be declared inside the `Student` class instead of declared as a different entity.

So, instead of the following:

```
public class Student
{
  // Other properties.

  int AddressId {get; set;}
  string City {get; set;}
  string State {get; set;}
  string Country {get; set;}
}
```

We did this. Basically, we just separated out the address properties in a container class named `Address`. The following code block shows how we can extract out `Address` class:

```
public class Student
{
  // Other properties.
  private Address Address { get; set; }
}
public class Address
{
  public int AddressId { get; set; }
  public string City { get; set; }
  public string State { get; set; }
  public string Country { get; set; }

  // Constructor. Just initialising City for now.
  public Address(string city)
  {
    City = city;
  }
}
```

These are some of the advantages of doing a subclass instead of directly adding them to the composite class:

WOW! eBook
www.wowebook.org

- The subclass is independent and simplifies the architecture. The code seems very easy to understand. The actual benefit you get out of it is when you try to do something extra on it, in the future. Suppose, I tell you to include another member Street for `Address`. If you have designed address properties directly inside composite classes, then you have to manually go to each class and add another member. However, in the case of the subclass, you just need to add that to the subclass and all composite classes using that will get the access automatically.
- The more you break down the classes into subclasses, the more you make them be reusable. The address class, for instance, can be reused in some other class as well. It is no more tightly coupled with the Composition.
- Due to the introduction of subclasses, the composite class is not complex anymore. Moreover, the subclasses can have methods defined in them which makes the life of the composite object easy. That is because the subclass can define it's related functions which can be called by the composite class. For example, if we want to get the full address, we can easily have a method inside the `Address` class, which will return a string using all those address properties. Thus, the composite class does not need to do anything on `Address` related stuff.

WOW! eBook
www.wowebook.org

Important notes

Thinking of a better design, such that it distributes responsibilities among different entities, is tough. But it is not impossible. When we start building the app, we don't always have full specifications or realize what could happen next. Eventually, when more specifications come in, we face problems as we ignore building subclasses for a certain common usage. Therefore, you will also see bad and repeated codes in many classes which can be packaged in a subclass and could easily be reused.

Composite or main classes should be responsible for what it is designed. A `Student` class is responsible for managing student information. There is no doubt that the address of the student is something it should handle. But, what if we get another entity like `Teacher` who also has an address? We have to repeat the same set of properties inside the `Teacher` class to achieve this. Code smell isn't it! That is when you should decide to separate `Address`--related information to another class.

WOW! eBook
www.wowebook.org

Aggregation

Aggregation is another type of object Composition. Let's explore this in details.

It is the process by which you bring existing objects together to form a new object. At first, it looks similar to Composition. But in reality, it has differences. To qualify as an aggregation, the complex object and its parts must satisfy the following relationships:

- **The part (member) is a component of the object (class)**: It is similar to Composition. That means the smaller object is a part of the complex object. For example, a person has a Google Drive folder.
- **The part (member) can belong to more than one object (class) at a time**: Unlike Composition, here the member is independent of the class. It might be referenced by other classes as well at the same time. For example, a Drive folder can be shared by many users at the same time.
- **The part (member) does not have its existence managed by the object (class):** As the last point states the member is not tied to the complex class, so it's creation and destruction is not managed by it. For example, all those people with whom the folder is shared have not created it. Unless they are provided with admin rights, they can't delete the folder.
- **The part (member) does not know about the existence of the object (class):** The member does not know whether the complex object exists or not, same as Composition. For example, a person does not know if the Drive folder exists or not.

WOW! eBook
www.wowebook.org

Consider an example

Now, we will try to alter the `Student` class which was illustrated in the Composition lesson. I am doing it because we have an `Address` property there. See how the constructor is updated to have the address as an argument:

```
public Student(int id, string name, DateTime dob, Address address)
{
    Id = id;
    Name = name;
    Dob = dob;
    Address = address;
}
```

This is not new to you. Let's try to elaborate it. We are simply passing the `Address` object into the constructor, which is then assigned to the `Address` property of the composite class `Student`.

The next confusion is what is the difference then? Let me explain. The `Address` property is declared inside the composite class gets created/destroyed like other members. But inside the constructor, we are assigning an external `Address` object to it. That means the composite class does not have any control over that incoming external object.

The `Student` class initialization will also be altered as follows:

```
static void Main(string[] args)
{
  Address add = new Address("Dhenkanal");
  Student pallu = new Student(1, "Pallavi Praharaj",
    new DateTime(1990, 6, 12), add);
}
```

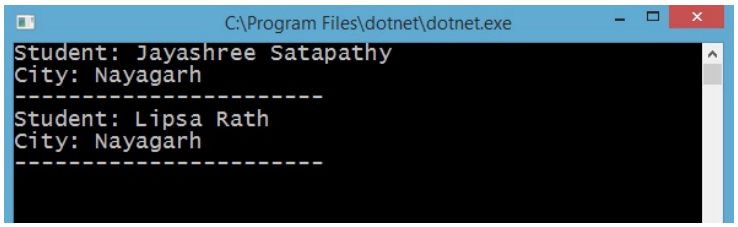
Let's analyze how this change can be considered as an Aggregation.

- **Rule 1: The part (member) is a component of the object (class)**: The `Address` property is referenced inside the `Student` class. Thus, it becomes a part.
- **Rule 2: The part (member) can belong to more than one object (class) at a time**: I passed an `Address` object into the constructor, which is used inside the composite class, for further operation. However, the object `add` is pretty much independent as it is created by the routine (`Main`, as shown in the preceding code). Let me allow it to be used by another `Student` as well:

WOW! eBook
www.wowebook.org

```
        static void Main(string[] args)
        {
          Address add = new Address("Nayagarh");
          Student rinu = new Student(1, "Jayashree Satapathy",
            new DateTime(1995, 11, 14), add);
          Student gudy = new Student(2, "Lipsa Rath",
            new DateTime(1995, 4, 23), add);

         rinu.PrintStudent();
         gudy.PrintStudent();
         Console.ReadKey();
        }
```

I know you might have some reservations about whether this will work or not. Here is the output for you:



Straightforward, isn't it! The `Main` method is the creator, so it can be used inside the scope anywhere till it is destroyed by it or by Garbage Collector. The `add` object is passed as a reference to both of the students. From these facts, we can infer that `Jayashree` and `Lipsa` and are two students who stay at the same address.

- **Rule 3: The part (member) does not have its existence managed by the object (class)**: Since it is wrapped inside the `Main` method, the composite object can't destroy it. By the way, it did not create it either. It was `Main` who gave birth to that.
- **Rule 4: The part (member) does not know about the existence of the object (class)**: The `Address` object does not have any idea about these student objects. It is autonomous inside the `Main` block.

WOW! eBook
www.wowebook.org

Composition versus Aggregation

Now that we have explored both types of Object Composition, let's look at the differences:

Composition	**Aggregation**
Generally, contains built-in type member variables Manages member's creation and destruction	Generally contains reference variables which live out of the scope of the composite class Does not manage member's creation and destruction

While these concepts have such tiny but important differences, they can still be mixed up in a composite class. That means a `Student` class having simple parts like `Id`, `Name`, `Address`, and so on as an Aggregation will be treated as a mix of both these concepts.

WOW! eBook
www.wowebook.org

Advantages and disadvantages

In case of Composition, you have the control to manage those members as you like. However, they are not independent and you can't use them in other places.

Whereas in Aggregation, there is no control of the member's life and if you forget to destroy the member after creation, it will be a culprit of memory leaks.

While both of them have their own pros and cons, it is just a matter of deciding which one to use where, according to the scenario. Moreover, you can mix them up and do amazing things. For example, some properties like `Id`, `Name`, etc. are used exclusively inside the class, so ideally we won't reuse these properties anywhere in the application. Therefore, if you build another class with all these properties and reference (aggregate) in the first class, that becomes unnecessary because the other class has no such usage outside the first class.

On the other hand, when you identify that some properties that can be packed up as a subclass (`AddressId`, `Address`, `State`, `City`, `Country`, and so on), you might have a requirement in future to refer to that new subclass somewhere in the code. Better to aggregate that. Let the caller manage its lifetime. Just get that into your composite class and use as required and forget it's management.

WOW! eBook
www.wowebook.org

Other important relationships

There are a few other relationships which you should be aware of. Unless you know these, you won't be able to visualize how objects typically work with each other.

WOW! eBook
www.wowebook.org

Association

So far we have completed two types of Object Composition, Composition, and Aggregation. Just to confirm that we are on the same page, Object Composition is used to package relationships into a complex object that is from one or more simpler objects (parts/members).

Now, we'll take a look at a type of relationship between two otherwise unrelated objects, called an **Association**. An Association is a bonding between two unrelated objects and they must satisfy the following conditions:

- **Object (member) associated is unrelated to the other object (class):** Here the objects, which are going to be associated, are not related to the object. Rather than being a part of a complex object as in a composition or aggregation, it is totally independent in nature. For example, a `Teacher` and `Student` are two independent objects, but not contained in each other. These entities get associated with each other while in a class for lessons.
- **Object (member) associated can belong to more than one object (class) at a time:** Like aggregation, the associated object can belong to multiple objects at the same time. For example, a `Student` can be associated with many teachers and vice versa.
- **Object (member) associated does not have its existence managed by the other object (class)**: All aggregated objects are independent here. Everyone manages themselves. For example, neither `Teacher` is going to decide what a `Student` will do nor does the `Student` decides what the actions of the `Teacher` should be.
- **Object (member) associated may or may not know about the existence of the object (class)**: Association relationships might be unidirectional or bidirectional. That means the two associated objects may or may not know each other. We will get more clarification on this once we see the code snippet. For example, a `Student` may or may not recognize a `Teacher`.

Association can be defined as *uses...a* relationship. The `Teacher` *uses* the `Student` to teach the lessons. The `Student` uses the `Teacher` to clarify doubts.

WOW! eBook
www.wowebook.org

Example

Let's take an example of a cricket player and bat. Consider the following:

```
public class CricketPlayer
{
  public string PlayerName { get; set; }

  public CricketPlayer(string name)
  {
    PlayerName = name;
  }

  public void Play(Bat bat)
  {
    bat.StartPlay(this);
  }

  public string GetPlayerName()
  {
    return PlayerName;
  }
}

public class Bat
{
  public string BrandName { get; set; }

  public void StartPlay(CricketPlayer player)
  {
    // Do something with the player.
    Console.WriteLine("Player Named as " + player.PlayerName
       + " is playing.");
    Console.ReadLine();
  }

  public string GetBrandName()
  {
    return "Some Brand Name";
  }
}
```

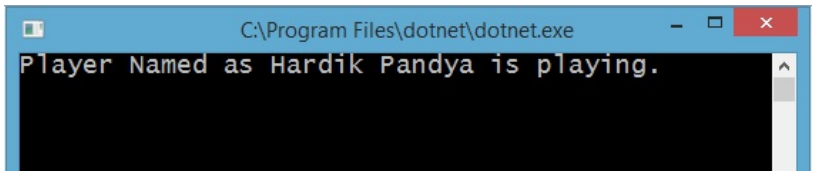
WOW! eBook
www.wowebook.org

Explanation of the preceding code

We have two different classes, `CricketPlayer` and `Bat`. Now let me create the objects as follows:

```
var cPlayer = new CricketPlayer("Hardik Pandya");
Bat bat = new Bat();

cPlayer.Play(bat);
bat.StartPlay(cPlayer);
```

The output is as shown in the following screenshot:



The methods `Play` inside the `CricketPlayer` class and `StartPlay` inside the `Bat` class are the ones, which you should be concentrating on at this point. Both are taking a reference param pointing to the other class's object. The whole point of this type of relationship between the player and bat is the common cause of starting t0 play.

Let's look at the rules which this relationship follows.

- **Rule 1: Object (member) associated is unrelated to the other object (class)**: Neither the bat relates to the player nor the player relates to the bat. But we will gradually see how they get associated.
- **Rule 2: Object (member) associated can belong to more than one object (class) at a time:** The bat can be used by many other players in the team, not just by one.
- **Rule 3: Object (member) associated does not have its existence managed by the other object (class)**: The player is not responsible for managing the bat. It is created even before the player comes to the team. Likewise, the bat is not going to give birth to or be the cause of death of a player unless the player hits himself with the bat, which is not going to happen.
- **Rule 4: Object (member) associated may or may not know about the existence of the object (class)**: The player knows about the bat as the bat is passed as an argument in the `Play` method. Similarly, the bat knows about the player as that got passed to the `StartPlay` method.

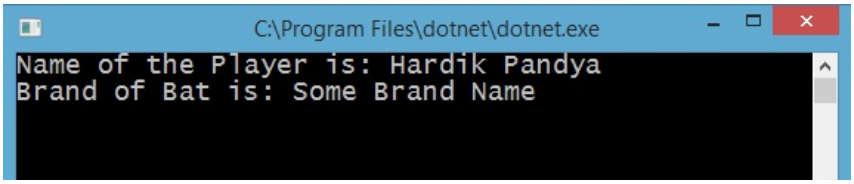
WOW! eBook
www.wowebook.org

If you understand cricket a little bit, you must know that a player uses a bat to bat. That means it depends on the bat object. The bat object, however, can be used by any batsman in the team, which leads to a `StartPlay` method in the `Bat` class. Obviously, the bat object needs to get associated with a player, which eventually commences the game.

Both of the objects depend on each other in order to start the game. They still exist as independent objects. Suppose, you don't call the `Play` and `StartPlay` method, nothing will change. The code will compile. That defines the association with each other for a common cause:

```
CricketPlayer cPlayer = new CricketPlayer("Hardik Pandya");
Bat bat = new Bat();

//cPlayer.Play(bat);
//bat.StartPlay(cPlayer);

Console.WriteLine($"Name of the Player is: {
    cPlayer.GetPlayerName() }");
Console.WriteLine($"Brand of Bat is: { bat.GetBrandName() }");
```

Here, I just commented out codes for play. That did not have any impact on the objects and they still exist. Then I used them to call other methods like `GetPlayerName` and `GetBrandName`.



WOW! eBook
www.wowebook.org

Types of Association

We have learned a bit about Association. However, there are different types and it would be handy for us to know about them.

WOW! eBook
www.wowebook.org

Reflexive Association

When two objects of the same type get associated, that association is called as **Reflexive Association**. Let's consider the class `Medicine`:

```
public class Medicine
{
  public string Name { get; set; }
  public Medicine AlternateMedicine { get; set; }

  public Medicine(string name, Medicine altMedicine)
  {
    Name = name;
    AlternateMedicine = altMedicine;
  }
}
```

This is very useful in many scenarios when we have dependencies of the same entity. We know that a medicine might have one alternate medicine, which again might have another alternate and so on.

WOW! eBook
www.wowebook.org

Indirect Association

Association, when formed by other means and not directly between two objects, is called an **Indirect Association**. We will try to grasp this concept with the help of the following example:

```
public class SoftwareEngineer
{
    public string Name { get; set; }
    public int LaptopId { get; set; }

    public SoftwareEngineer(string name, int laptopId)
    {
            Name = name;
            LaptopId = laptopId;
    }
 }

 public class Laptop
 {
    public int LaptopId { get; set; }
    public string LaptopName { get; set; }

    public Laptop(int id, string name)
    {
       LaptopId = id;
       LaptopName = name;
    }
  }

  public class AvailableLaptops
  {
    public static List<Laptop> Laptops { get; set; }
    static AvailableLaptops()
    {
      Laptops = new List<Laptop>
      {
        new Laptop(1, "Laptop1"),
        new Laptop(2, "Laptop2"),
        new Laptop(3, "Laptop3"),
        new Laptop(4, "Laptop4"),
      };
    }

    public static Laptop GetLaptop(int id)
    {
      return Laptops.Find(l => l.LaptopId == id);
    }
  }
  static void Main(string[] args)
  {
    SoftwareEngineer softEng = new SoftwareEngineer("Tworit Dash", 3);
    // Get the Laptop object from AvailableLaptops class by id.
    Laptop usedLaptop = AvailableLaptops.GetLaptop(3);
```

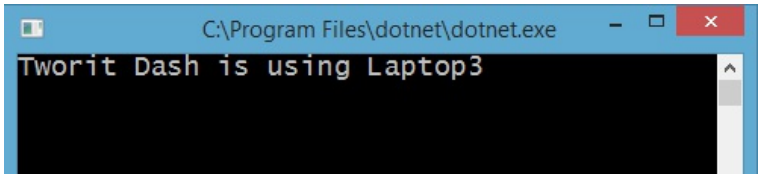
WOW! eBook
www.wowebook.org

```
        Console.WriteLine(softEng.Name + " is using " +
          usedLaptop.LaptopName);
        Console.ReadLine();
      }
```

We have two classes named `SoftwareEngineer` and `Laptop` which are meant to be associated with each other. In this case, we are trying to relate them indirectly via a static class `AvailableLaptops` which has a list of `Laptop` objects containing their `Id` and `Name`.

Let's focus on the `Main` method now. A `SoftwareEngineer` is instantiated (`Name Tworit Dash` with a `LaptopId 3`). We need to print his name and the laptop name he is using. We have the `LaptopId`. If you closely look at the `AvailableLaptops` class, there is a static method `GetLaptop` which takes `laptopId` as an argument and then finds the laptop from the list it already has.

So, `Laptop usedLaptop = AvailableLaptops.GetLaptop(3);` would get us the required `Laptop` object which has ID `3` and `Name Laptop3`. Now, it is just a matter of printing the engineer's name and the name of the laptop returned by the static class:



This is called **Indirect Association** because the association is established with the help of another class which can interact with a class and returns the results to another class requesting any data from the first class.

To summarise the relationships, let's do a quick recap.

Composition:

- Part of a complex class.
- Members can't be a part of multiple classes.
- Members are created and destroyed by a complex class.
- Unidirectional: Members do not know about the complex object.
- *A part of* relationship

Aggregation:

- Part of a complex class

WOW! eBook
www.wowebook.org

- Members can be a part of multiple classes
- Members are neither created nor destroyed by a complex class
- Unidirectional: Members do not know about the complex object
- *Has a* relationship

Association:

- Classes are not related but require each other when needed
- Associated members or objects can be used by multiple objects
- Members are neither created or destroyed by a complex class
- Unidirectional or Bidirectional: Objects may or may not know each other
- *Uses a* relationship

WOW! eBook
www.wowebook.org

Composition over Inheritance

This topic is very interesting and is discussed online in many websites and blogs. As the topic headline says **Composition is preferred over Inheritance**, we need to understand why it is so important by identifying the problems. The better the design from the beginning of your software development, the better the maintainability and reuse.

Inheritance and Composition are two pillars of OOPs concepts. Unless they are used wisely according to your architecture, they will create problems in the future when you start adding complexity to your application.

We have already discussed Composition, now, let's discuss Inheritance before we move towards the main topic of discussion.

WOW! eBook
www.wowebook.org

Inheritance

As the name suggests, the act of acquiring or deriving some behavior from someone is called **Inheritance**. In the programming world, when a certain class is inherited from another class, it creates an Inheritance. Some basic examples are written as follows:

- A car *is a* vehicle. The vehicle has a certain behavior which the car acquires when it is built.
- A rectangle *is a* shape.
- An `HourlyEmployee` *is an* employee. A `MonthlyEmployee` *is an* employee too.
- Chicken curry *is a* dish.

Notice, they all have something in common. That is the phrase *is a*. Inheritance is defined as an *is a* relationship.

The car is a vehicle, but it might have a music system as well, which is not a common behavior of a vehicle. Thus, the derived or child classes can have their own behavior as well. `HourlyEmployee` and `MonthlyEmployee` are employees of a certain company who share many benefits from the company. However, their salary is not the same.

WOW! eBook
www.wowebook.org

Example on User class

Let's see how we can implement Inheritance.

```
public abstract class User
{
    public int Id { get; set; }
    public int RoleId { get; set; }
    public string Name { get; set; }
    public string EmailId { get; set; }
    public string MobileNumber { get; set; }

    public int SaveUser(int userId)
    {
       // Database operation to save the user.
       return userId;
    }
  }

  public class Admin : User
  {
    public string CompanyDepartment { get; set; }
    public Admin()
    {
      RoleId = 1;
    }
  }

  public class Manager : User
  {
    public List<TeamLead> TeamLeads { get; set; }
    public Manager()
    {
      RoleId = 2;
    }
  }

  public class TeamLead : User
  {
    public List<string> Projects { get; set; }
    public TeamLead()
    {
      RoleId = 3;
    }
  }
```

We have different types of users in our company such as `Admin`, `Manager`, `TeamLeads`, `HR`, and more. Although these entities are different, they have some properties in common. They must have an `Id`, `RoleId`, `Name`, `EmailId`, `MobileNumber`, and so on.

As they have common properties and behavior we make an abstract base class `User` which has all those declared inside it. We won't be instantiating this class, so it is

WOW! eBook
www.wowebook.org

restricted by using an abstract keyword. Every type of user is going to have some operation. The simplest operation is `SaveUser()`, which is defined inside the base class so that it will be available for all child classes.

In this example, we have different properties declared inside child classes as well.

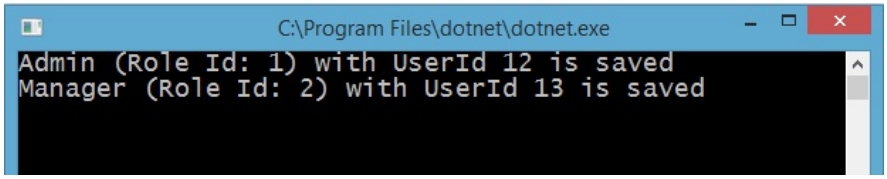
Consider the following:

- Admin--`public string CompanyDepartment { get; set; }`
- Manager--`public List<TeamLead> TeamLeads { get; set; }`
- TeamLead--`public List<string> Projects { get; set; }`

We can also define different methods specific to them inside these child classes. When we try to create objects, or in other words, create an `Admin` and `Manager`, it will look something like as follows:

```
static void Main(string[] args)
{
    Admin admin = new Admin()
    {
      Id = 12
    };
    admin.SaveUser(admin.Id);

    Manager manager = new Manager
    {
      Id = 13
    };
    manager.SaveUser(manager.Id);
     Console.WriteLine("Admin (Role Id: {0}) with UserId {1}
        is saved", admin.RoleId, admin.Id);
    Console.WriteLine("Manager (Role Id: {0}) with UserId {1}
        is saved", manager.RoleId, manager.Id);
}
```

You can see we are creating each type of user and then sending their IDs to the save method for further database processing. `RoleId` is assigned inside the constructors of each type of `User` class. This produces output as follows:



WOW! eBook
www.wowebook.org

New User Type

The company decides to have a new type of employee named Delivery Manager, who will be having certain privileges, but not all. This role will take partial responsibilities from `Manager` as well as from `TeamLead`.

A Delivery Manager can `CreateProject` (like a TeamLead) and ;`AssignProjectToTeamLead` (like a `Manager`). Also while doing all these, he/she can `SendNotificationToCto` which is a new method. So, those two methods will have additional codes which are not an exact copy from the `Manager` and `TeamLead` classes:

```
public class DeliveryHead : Manager, TeamLead
{
}
```

class DependencyExamples.TeamLead

Class 'DeliveryHead' cannot have multiple base classes: 'Manager' and 'TeamLead'

Multiple Inheritance *is a language-specific feature which enables a class to inherit features of more than one parent class. This feature can introduce complexities in the design and the supported languages have their own way of handling such scenarios. C#, Java, Swift etc. does not support Multiple Inheritance, but they allow implementing multiple protocols which are named* ***Interfaces****. I just wanted to show you that we have to go with some alternative approach to solve the problem in hand instead of inheriting from multiple classes as shown above, which is anyway not supported by C#.*

WOW! eBook
www.wowebook.org

Problem which we come across

Visual Studio complains to me that I can't code like this, which is actually termed Multiple Inheritance. Moreover, when a company introduces more roles, the system gets complex to manage when we have responsibilities (methods) defined in each class. Because when we have to mix responsibilities for a certain user role, we will have to create duplicate codes that would have been already written in some class.

To explain what I'm saying, imagine a `Car` class, which gets inherited by classes like `Toyota`, `BMW`, `Volkswagen`, and more. For some reason, I got mad and brought both `Toyota` and `Volkswagen` to my workshop and then built a new brand out of them. I will name it VolksTaditToy. Please don't kill me for this name.

VolksTaditToy now has a combination of functions from both of these cars. But there is no way to deal with them in my program. Don't be so surprised, if you see more of such cars on the road because there is no scarcity of such fools in this world. Gradually your program will be in a situation where there is no escape.

How can we write this class with the concept of inheritance? Impossible right! Let's make that possible for our initial `User` problem.

WOW! eBook
www.wowebook.org

The solution to the problem

The composition is our savior here. Let's see how we can solve this problem using this relationship. We will introduce a `Role` class. Obvious right! `Manager`, `TeamLead`, and `DeliveryManager` are different roles played by employees:

```
public class Role
{
    public int RoleId { get; set; }
    public string RoleName { get; set; }
}
```

Now every other user type class will derive from this class:

```
public class Admin : Role
{
    public string CompanyDepartment { get; set; }
    public Admin()
    {
      RoleId = 1;
    }
}

public class Manager : Role
{
    public List<TeamLead> TeamLeads { get; set; }
    public Manager()
    {
      RoleId = 2;
    }
}
public class TeamLead : Role
{
    public List<string> Projects { get; set; }
    public TeamLead()
    {
      RoleId = 3;
    }
}
public class DeliveryHead : Role
{
   public DeliveryHead()
    {
      RoleId = 4;
    }
}
```

Alright, what is the next step? The remaining class is `User`. We need to do a couple of modifications in that, shown as follows:

```
public class User
```

WOW! eBook
www.wowebook.org

```
{
    public int Id { get; set; }
    public List<Role> RoleIds { get; set; }
    public string Name { get; set; }
    public string EmailId { get; set; }
    public string MobileNumber { get; set; }

    public int SaveUser(int userId)
    {
       // Database operation to save the user.
       return userId;
    }
}
```

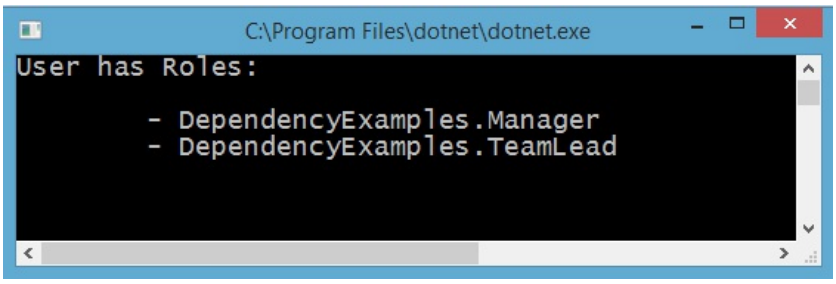
The first modification is to remove the abstract keyword because we will now create objects of this class. Next is to have a property `public List<Role> RoleIds { get; set; }` instead of `public int RoleId { get; set; }`. We did this to allow assigning multiple roles to a user/employee.

Observe, how we can create a user with multiple roles in the following main method:

```
static void Main(string[] args)
{
    User deliveryManager = new User()
    {
        RoleIds = new List<Role>
        {
           new Manager(),
           new TeamLead()
        }
    };
    Console.WriteLine(string.Format("User has Roles:\n\n\t-
       {0}", string.Join("\n\t- ", deliveryManager.RoleIds)));
}
```

While creating a `DeliveryManager` type of `User`, we are assigning multiple roles to the user by creating a list of `Manager` and `TeamLead` types. As they inherit from `Role` base class, so `RoleIds` recognize these types.

This piece of code produces the following output:



We conclude that Composition takes over Inheritance in many of such cases. That

WOW! eBook
www.wowebook.org

means you need to be very careful when you start designing the classes. Otherwise, the situation will get worse in the future when your system grows as you will create a mess. You should definitely avoid duplicate codes. When you see that you are writing something which has already been written before, stop and think before you proceed. Try to normalize as much as possible at that moment.

WOW! eBook
www.wowebook.org

Role of Object Composition in Dependency Injection

Now that we understand the concept of Object Composition, let's analyze an actual software project problem and how that can be trapped using Object Composition. In the process, we will find out the importance of this concept in a DI context:

```
class Mail
{
  protected bool SendMail(string mailId, string message)
  {
     // Logic to send an email
     return true;
  }
}
class Notification : Mail
{
   void SendNotification(string mailId, string message)
   {
      SendMail(mailId, message);
   }
}
```

So, the `Notification` class is inheriting the `Mail` class, so that it can call `SendMail()`. This structure is not wrong, but it will create complications going forward.

Imagine another class for `Sms`, where we can have a method something like `SendSms()`. There is no way here for the `Notification` class to call that method as multiple inheritances is not possible.

To solve this problem, we can easily use Object Composition along with Dependency Injection. Let's alter the code first. This is shown as follows:

```
interface IMail
{
    bool SendMail(string mailId, string message);
}

interface ISms
{
    bool SendSms(string mobile, string message);
}

public class Mail : IMail
{
   public bool SendMail(string mailId, string message)
```

WOW! eBook
www.wowebook.org

```
      {
         // Logic to send an email
         Console.WriteLine("SendMail Called");
         return true;
      }
    }

    public class Sms : ISms
    {
       public bool SendSms(string mailId, string message)
       {
         // Logic to send a Sms
         Console.WriteLine("SendSms Called");
         return true;
       }
     }

     class Notification
     {
        private readonly IMail _mail;
        private readonly ISms _sms;

        public Notification(IMail mail, ISms sms)
        {
          _mail = mail;
          _sms = sms;
        }

       public void SendNotification(string mailId, string mobile,
          string message)
       {
          _mail.SendMail(mailId, message);
          _sms.SendSms(mobile, message);
       }
     }
```

`IEmail` and `ISms` are interfaces which have `SendMail()` and `SendSms()` methods respectively. The next thing is to implement these interfaces in classes `Mail` and `Sms`. We will write our sending logic inside the methods implemented in these classes.

Notice the `Notification` class, which is not inheriting any class, rather it is referencing new interfaces. Then inside a parameterized constructor, we have `IMail` and `ISms` as arguments. The `SendNotification()` method now takes the necessary details like `mailId`, `mobile` and `message` to call the methods using the interfaces.

What's the advantage then? We wrote more code, didn't we? The point here is very interesting. If you look at the code to instantiate `Notification` class, you will get some hint. Let's look at that:

```
    static void Main(string[] args)
    {
        Notification notify = new Notification(new Mail(),
           new Sms());
        notify.SendNotification("taditdash@gmail.com",
```

WOW! eBook
www.wowebook.org

```
            "9132994288", "Hello Tadit!");
        Console.ReadLine();
    }
```

Got the hint? Let me explain. We injected the `Mail` and `Sms` class instances into the `Notification` constructor which is assigned to the properties `_mail` and `_sms`. It would automatically call the methods inside `Mail` and `Sms` classes. So, we composed `Notification` class with `IMail` and `ISms` references. This is where Object Composition came into the picture along with Dependency Injection.

Suppose at some point in time you want to include another class for mail sending (something like `SmtpMail`). You just need to write that class implementing the same `IMail` interface and define the `SendMail` method. Bang, it's done. No need to make the `Notification` class dirty anymore. It will work as expected.

```
    public class SmtpMail : IMail
    {
       public bool SendMail(string mailId, string message)
       {
          // Logic to send an email
          Console.WriteLine("SmtpMail Called");
          return true;
       }
    }
    static void Main(string[] args)
    {
        Notification notify = new Notification(new SmtpMail(),
            new Sms());
        notify.SendNotification("taditdash@gmail.com",
           "9132994288", "Hello Tadit!");
        Console.ReadLine();
    }
```

As you can see, I just declared the new class and injected the object like `new SmtpMail()` instead of `new Mail()`. That is the only change. The rest will work as expected because I have injected the object instead of directly referencing it inside the `Notification` class.

To summarise, the following is what we achieved:

- We introduced flexibility through interfaces for concrete dependencies.
- We can easily plug in new concrete class dependency just by implementing the interface abstraction.
- We composed `Notification` object with the dependency objects just with one go.
- We moved all initialization code to one place inside the `Main` method

When I said we moved the initialization code to one place, that location is denoted as

WOW! eBook
www.wowebook.org

the **Composition Root** of the application.

WOW! eBook
www.wowebook.org

Composition Root

Composition Root composes all the independent modules of the application. During runtime, Object Composition is the first thing that happens before any other operations. As soon as the object graph is wired up with dependencies, Object Composition is finished, then the components related to the application can take over. Object Composition should happen as close as possible to the application's entry point.

The entry points in .NET Core 2.0 Console App and ASP.NET Core 2.0 MVC app are the same and that is inside the `Main` method of the `Program.cs` class. .NET Core 2.0 Console App's `Main` method is clean, but on the other hand, ASP.NET Core 2.0 MVC has some bootstrapping codes inside the `Main` method. However, we usually write composition codes inside a `ConfigureServices` method which can be called inside the `Main` method.

When you do a File | New | Project | .NET Core | Console App (.NET Core) in Visual Studio 2017, you will see the following in the `Main` method:

```
namespace PacktConsoleApp
{
  class Program
  {
    static void Main(string[] args)
    {
      // We will do all Object Composition here directly or
          calling a ConfigureServices method.
      Console.WriteLine("Hello World!");
    }
  }
}
```

In the case of ASP.NET Core 2.0 MVC App, when you select File | New | Project | Web | ASP.NET Core Web Application (in the next screen, select appropriate template), the template for Web Application generates the `Main` method in the `Program` class and `Startup` would look like the following:

WOW! eBook
www.wowebook.org

Main Method

```
namespace PacktCoreWebApp
{
    0 references
    public class Program
    {
        0 references | 0 exceptions
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        1 reference | 0 exceptions
        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

Startup Class

```
namespace PacktCoreWebApp
{
    2 references
    public class Startup
    {
        0 references | 0 exceptions
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        1 reference | 0 exceptions
        public IConfiguration Configuration { get; }

        // This method gets called by the runtime.
        // Use this method to add services to the container.
        0 references | 0 exceptions
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        0 references | 0 exceptions
        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env)...
    }
}
```

The way we composed the `Notification` object in the last section is known as **Poor Man's DI**. Instead of doing that, we should apply DI Containers to compose and manage objects in that location.

Object Composition is the fundamental building block of DI and the easiest one to understand because we already know how to compose objects with many examples. Now it's time to learn what are the challenges we would face composing objects for DI due to the framework capabilities. The issues are related to a framework and nothing to do with the Object Composition concept. Let's find out in the upcoming sections.

WOW! eBook
www.wowebook.org

Composing .NET Core 2.0 Console application

Inside the `Main` method, we can easily compose objects with built-in DI Container. If you remember, we have already discussed that from container initialization, registration of objects to the container, resolving dependencies and then release the components from the container, everything should happen inside the Composition Root, which is considered as the `Main` method here.

Consider an example of the `Main` method inside a console application:

```
static void Main(string[] args)
{
  // Setup container and register dependencies.
  var serviceProvider = new ServiceCollection()
  .AddTransient<IEmployeeService, EmployeeService>()
  .BuildServiceProvider();

  // Get the service instance from the container and
  do actual operation.
  var emp = serviceProvider.GetService<IEmployeeService>();
  emp.HelloEmployee();

  Console.ReadKey();
}
```

This is simple code which leverages the extension method available, `Add***`, to register the dependency with the container. Then we used the `GetService` method to get the implemented type by the interface. With the instance, we then can do everything we want further in the application.

Recall that Composition Root is the place where we should do all operations related to Dependency Management. It's not recommended to use the `serviceProvider` outside the Composition Root or `Main` method to be specific. The same rule applies to ASP.NET Core MVC as well. We are going to explore that in a while.

You can, of course, introduce another method and you can name it `ConfigureServices` (shown as follows) for a cleaner code structure. You can give any name to the method, but this name resembles the name of the method specially dedicated in ASP.NET Core MVC app for dependency injection configurations as we saw in the picture in the last section. This new method added is shown in the

WOW! eBook
www.wowebook.org

following code snippet:

```
static void Main(string[] args)
{
  ConfigureServices(new ServiceCollection());
  Console.ReadKey();
}

public static void ConfigureServices(IServiceCollection
  serviceCollection)
{
  // Setup container and register dependencies.
  var serviceProvider = serviceCollection
  .AddTransient<IEmployeeService, EmployeeService>()
  .BuildServiceProvider();

  // Get the service instance from the container and
     do actual operation.
  var emp = serviceProvider.GetService<IEmployeeService>();
  emp.HelloEmployee();
}
```

Notice that we are not manually releasing the objects or the container. The reason is that release is automatically handled by the DI Container according to the life style you decide. `AddTransient`, `AddSingleton` and `AddScoped` are readily available methods which helps to perform different types of lifetime for the objects. We will explore more about Object Lifetime in `Chapter 6`, *Object Lifetime*.

WOW! eBook
www.wowebook.org

Object Composition in ASP.NET Core MVC 2.0

Like Console Application, we can follow the same procedure to deal with dependencies inside ASP.NET Core MVC 2.0 apps. Unlike the console app, the `Main` method inside the `Program.cs`, in this case, is populated with default codes to initiate the MVC App with required configurations. It is that location from which it instructs the framework to load the Startup class. The `host` inside the `Main` method executes the `Startup` class `ConfigureServices` method.

ASP.NET Core MVC is designed to be DI-friendly. But it does not force you to apply DI always. To deal with dependencies in ASP.NET MVC, we can take the Poor Man's DI approach to manually manage them or leverage built-in/third party DI Container's technique to register, resolve, and release dependencies. Let's dive a little deep into the controller initiation process and see if we find anything useful.

The heart of MVC lies in controllers. Controllers handle requests, process them, and return the response back to the client. Thus, the controllers should delegate their responsibility to the other modules as required. That means the controller will refer to other classes for certain tasks. It will use the `new` keyword for the dependency object creation inside the action methods, which we can easily avoid as we are using DI Container. With the dependency injection technique, we should be able to inject dependencies into the controllers with Constructor Injection.

`IControllerFactory` is an interface inside `Microsoft.AspNetCore.Mvc.Controllers` namespace which enables us to create and release controllers. The interface contains two methods, shown as follows:

```
namespace Microsoft.AspNetCore.Mvc.Controllers
{
  /// <summary>
  /// Provides methods for creation and disposal of controllers.
  /// </summary>
  public interface IControllerFactory
  {
    object CreateController(ControllerContext context);
    void ReleaseController(ControllerContext context,
     object controller);
  }
}
```

WOW! eBook
www.wowebook.org

ASP.NET Core MVC 2.0 ships with a `DefaultControlFactory` which implements this interface. Let's have a look at the source code:

```
namespace Microsoft.AspNetCore.Mvc.Controllers
{
  /// <summary>
  /// Default implementation for <see cref="IControllerFactory"/>.
  /// </summary>
  public class DefaultControllerFactory : IControllerFactory
  {
    private readonly IControllerActivator _controllerActivator;
    private readonly IControllerPropertyActivator[]
        _propertyActivators;
    public DefaultControllerFactory(
        IControllerActivator controllerActivator,
        IEnumerable<IControllerPropertyActivator> propertyActivators)
        {
          if (controllerActivator == null)
          {
            throw new ArgumentNullException(nameof(
             controllerActivator));
          }

          if (propertyActivators == null)
           {
            throw
              new ArgumentNullException(nameof(propertyActivators));
           }

           _controllerActivator = controllerActivator;
           _propertyActivators = propertyActivators.ToArray();
        }
        public virtual object CreateController
          (ControllerContext context)
        {
           // Codes removed just for book.
               You can find codes in Github.
         }

         public virtual void ReleaseController(ControllerContext
           context, object controller)
        {
          // Codes removed just for book. You can
              find codes in Github.
        }
  }
}
```

`DefaultControllerFactory` has a constructor injection with the required dependencies for the `ControllerActivator` and `PropertyActivators`. Thus, this factory is composed of activators. Like a factory, there is an interface for `Activator` also named `IControllerActivator`. There are providers for Factory and Activators named `ControllerFactoryProvider` and `ControllerActivatorProvider` respectively.

Now, the most important part. The Object Composition of these factories is actually

WOW! eBook
www.wowebook.org

done inside the `AddMvcCore()` method of class `MvcServiceCollectionExtensions` which stays inside `namespace Microsoft.Extensions.DependencyInjection`. The `namespace` name contains `DependencyInjection`, which itself gives us a hint that we are going to do some injection for sure to initiate these activators and factories. Let's see a snapshot of the `AddMvcCoreServices()` method (which is another method called from `AddMvcCore()`) responsible for composing all required dependencies for controller activation and initialization:

```
//
// Controller Factory
//
// This has a cache, so it needs to be a singleton
services.TryAddSingleton<IControllerFactory, DefaultControllerFactory>();

// Will be cached by the DefaultControllerFactory
services.TryAddTransient<IControllerActivator, DefaultControllerActivator>();

services.TryAddSingleton<IControllerFactoryProvider, ControllerFactoryProvider>();
services.TryAddSingleton<IControllerActivatorProvider, ControllerActivatorProvider>();
services.TryAddEnumerable(
    ServiceDescriptor.Transient<IControllerPropertyActivator, DefaultControllerPropertyActivator>());
```

You can see how the interfaces are registered as the concrete classes. This method contains many other service registrations for all the background work needed for the controller process. But we got an idea about how Object Composition is implemented inside the framework itself.

If we want to design our own custom controller factory, we can do that as well by registering the required factory and provider inside this method for initiation.

WOW! eBook
www.wowebook.org

Summary

This chapter touched upon important relationships between objects in programming. We created very basic classes and tried to grasp the concepts. Then we moved our focus to Object Composition, and it's types, Composition, and Aggregation.

Moreover, we discussed Association. With code examples and outputs, we saw how these relationships matter a lot while coding.

Finally, we were introduced to Inheritance with an example. Once we completed all these, we moved towards a very important saying *Composition over Inheritance*. This is a real-time issue which developers face while applying new requirements to existing class structures.

Then, we discussed the important role played by Object Composition in Dependency Injection. Also, we saw how this pattern is followed in ASP.NET Core MVC 2.0.

It's time to see how objects are created, how they live and are then destroyed in `Chapter 6`, *Object Lifetime*. We will relate the object lifetime with what we already know from this chapter.

WOW! eBook
www.wowebook.org

Object Lifetime

Object Lifetime is the time duration between an object's creation and destruction. In Functional Programming Languages, data stored in constant variables have a scope defined which is immutable in nature. This means that their lifetimes have a functional scope (no destruction) as long as the app is not stopped. On the other hand, the objects in Object Oriented Programming are mutable and have different types of scope which leads to different lifestyles.

Memory plays a vital role in application life cycle. All objects and variables use memory space. Therefore, it is important to learn the concepts which deal with an object's flow during application execution. Unless we know how to free up space by using proper codes or patterns, it would result in **Memory Leaks**.

If a computer program exposes bugs and erroneously manages the memory allocations, then resources become unavailable. This condition is called Memory Leaks.

To avoid Memory Leaks, we should take proper care while designing our classes, so that the resources are available when needed. That would only happen if and only if resources attached to objects are released by the program as soon as objects go out of scope. Thus, the app runs seamlessly as unused space is regularly cleared. However, this process is not automatic in all scenarios. The reason, we are going to explore this topic, is to understand how exactly the DI technique manages the lifetime of objects differently in different scenarios, which in return helps us take appropriate decisions while designing our classes.

An object is born when it's related class is instantiated. The newly born object stays around for a limited period of time as long as the application keeps a reference of that and continues to use that. If your application is closed or the reference of the object goes out of scope in the code, then .NET Framework will flag that object to be removed from memory.

In this particular chapter, we will learn how objects are managed by .NET Core. Also, we will look into techniques to determine when an object is disposable and how to work with it.

WOW! eBook
www.wowebook.org

Topics we will cover in this chapter are:

- Managed and unmanaged resources.
- Object's creation and destruction
- `IDisposal` interface
- Object lifetime management in .NET Core

WOW! eBook
www.wowebook.org

Manage the object life cycle

There are two basic places in the memory--stack and heap.

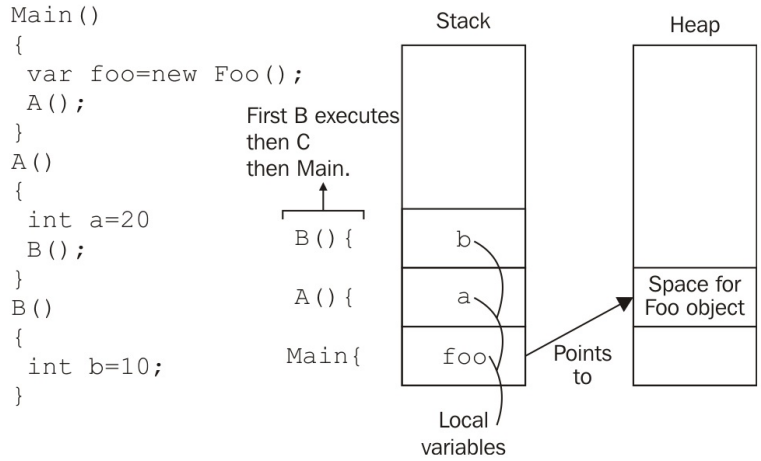
WOW! eBook
www.wowebook.org

Stack versus heap

Let's learn a little bit about these memory space types.

Stack	**Heap**
Static memory: Fixed memory space allocated for the app.	Dynamic memory: No fixed space.
When a method is called, a chunk of stack is reserved to store the method's information like a method name, local variables, return type, and so on.	Can store anything. The developer has the flexibility to manage this space.
Memory allocation has a pattern in **LIFO (Last In First Out)** order. So, when the `Main` function calls method `A()` and then `A()` calls `B()`, `B()` will be stored at the top and executes first. Refer to the following diagram:	No such pattern to store data.

The following diagram shows how the data gets stored in stack and heap for one program:



`Main()` calls `A()` which then calls `B()` method. According to stack properties, it first moves the last one out, which is `B()`. So, `B()` executes first. Then `B()` is removed from the stack and then `A()` is processed and removed from memory. After that,

WOW! eBook
www.wowebook.org

the `Main()` method executes and removed. The reference type variable named as `foo` inside `Main()` is stored in stack, but the actual object is allocated memory in heap.

Consider the following example:

```
static void Main(string[] args)
{
  string name = "Dependency Injection";
  SomeClass sc = new SomeClass()
}
```

The variable `name` is a value type, which is directly stored on the stack. But, when we write `SomeClass sc = new SomeClass()`, it actually tells the framework to store the object in heap. In addition to this, it also allocates a memory space for a variable `sc` in the stack, which holds the reference to this object.

Now, when the `Main` method execution is completed, the variables `name` and `sc` will be released and memory space becomes free. Here is the catch though. The variable `sc` (reference type) is released from the stack, but the actual object is still on the heap. It's just the reference, which is removed. As the reference is removed from the stack, so there is no way to actually know if there is an object existing on the heap., related to it. We have ended up with a management problem now.

To get rid of this (in C++), we could have done something like this `delete sc;`. However, in C#, which is a managed language, there is a service known as **Garbage Collector** (**GC**), which automatically cleans up the unused memory by analyzing all those flagged *out of scoped* objects.

Managed Language *is a high-level language that depends on services provided by a runtime environment to execute, such as Garbage Collection service, Security Service, Exception Handling, and more. It uses Common Language Runtime (CLR) to execute in .Net languages or* ***Java Virtual Machine (JVM)*** *in Java.*

WOW! eBook
www.wowebook.org

Managed and unmanaged resources

Pure .NET code is called **managed resources** as that can be directly managed by the runtime environment. On the other hand, unmanaged resources are those that are not in direct control of the runtime such as, file handles, COM objects, database connections, and so on. For example, if you open a connection to a database server this will use resources on the server (for maintaining the connection) and possibly other non .NET resources.

The managed resources are targeted directly by CLR, therefore, Garbage Collector cleans them up, which is an automatic process. As a developer, you don't usually need to explicitly call GC. However, there is a catch, when we consider the unmanaged resources like database connections. We have to deal with them ourselves as the CLR can't. We have to release them manually using the `Finalize` method.

WOW! eBook
www.wowebook.org

Generations

The heap is categorized into three generations so it can handle long-lived and short-lived objects. Garbage collection basically reclaims short-lived objects that typically occupy only a small part of the heap.

There are the following three generations of objects on the heap:

- **Generation 0**: When an object is initialized, it's generation gets started. It first falls into Generation 0. This generation's objects are usually short-lived. These objects are more prone to GC for destruction. GC collects these short-lived objects so that they can be released to free up memory space. If the objects survive from being collected by GC, that means they are going to stay for a longer time, thereby being promoted to the next generation.
- **Generation 1**: This generation objects live for a longer duration than Generation 0 objects. GC does collect the objects from this generation, but not so regularly as it does for Generation 0 because their existence is extended by the application for more operations. Survivors of this generation go to Generation 2.
- **Generation 2**: These are the longest existing objects in the application. Prolonged objects which successfully pass through the previous two generations are automatically considered for Generation 2. GC rarely comes into the picture when it comes to releasing these objects.

WOW! eBook
www.wowebook.org

Object creation

Constructors are responsible for giving birth to the object of a particular class. If we create any class without a constructor, the compiler will automatically create a one default constructor for that class. At least one constructor is present in every class.

The constructors can be overloaded too which provide a convenient way to build the object with different attributes, meaning, it can instantiate the object by taking certain parameters (like a normal method) and assign them to its properties inside its body (also called as a parameterized constructor). A constructor should have the same name as the class name.

WOW! eBook
www.wowebook.org

The Copy constructor

There is another type of constructor called the **Copy constructor**. As the name suggests, it can copy one object to the new object (of the same class) which is going to be instantiated. In other words, it is a parameterized constructor that contains a parameter of same class type. The main purpose of a copy constructor is to initialize a new instance to the values of an existing instance. We will see how this can be done with an example in a while.

WOW! eBook
www.wowebook.org

Object destruction

We have different ways to destroy an object. Let's explore them one by one.

WOW! eBook
www.wowebook.org

Finalize

Finalizers are used to destroy the objects. We can design Finalizers using destructor methods using the tilde (~) sign with the class name. We will see it in action soon:

```
~Order()
{
  // Destructor or Finalizer
}
```

Garbage Collector has full control on the finalization process as it internally calls this method when objects go out of scope. We can, however, write codes inside the destructor to customize our requirements, but we can't just say to someone to call the destructor. Even if you are very sure that the object is not required anymore and decide to release it, you can't explicitly execute destructor in order to free up space. You have to wait until GC collects the object for destruction.

The finalization process has two collection cycles. In the first cycle, short-lived objects are flagged for finalization. In the next cycle, it calls the finalizers to release them completely from memory space.

WOW! eBook
www.wowebook.org

The IDisposable interface

Unmanaged resources, as we discussed, are not in direct control of the framework. We can easily reclaim these resources inside Finalizers, as we discussed. That means, they will be released when the objects are destroyed by the garbage collector. However, the GC destroys the objects only when the CLR demands more free memory. So, resources may still exist even long even after the object goes out of scope.

Thus, there is a need to free up resources as soon as we are done with them. Your classes can provide a mechanism to actively manage system resources if they implement the `IDisposable` interface. This interface exposes one method, `Dispose()`, which clients should call when they finish using an object. You can use the `Dispose` method to immediately release resources to perform tasks such as closing files and database connections etc. Unlike the `Finalize` destructor, the `Dispose` method is not called automatically. Objects must explicitly call `Dispose` when they want to release the resources.

This method is the single method in an `IDisposable` interface and can be used to release unmanaged resources manually.

```
public interface IDisposable
{
  void Dispose();
}
```

Now it's just a matter of calling the `Dispose()`. But wait. You can only call this method from an object of a class which have implemented this interface and defined the `Dispose()` method.

For example, the `SqlConnection` class has implemented this interface and provided us with the `Dispose()` method, which can be used as follows. As soon as you are finished using the connection object, call `Dispose`:

```
var connection = new SqlConnection("ConnectionString");
// Do Database related stuff here.

// After we are done, let's call Dispose.
connection.Dispose();
```

WOW! eBook
www.wowebook.org

There is another beautiful way to deal with the disposal of objects in .NET. Instead of calling `Dispose` directly, we can use the `using` block. The same statement can be decorated with the `using` block coded as follows:

```
using (var connection = new SqlConnection("ConnectionString"))
{
  // Use the connection object and do database operation.
}
```

When we do this, it converts the code to `try...finally` intermediate code. It disposes the connection object inside the `finally` block, which we created. Unless you do this, the connection object stays in memory. In due course, when we get a huge number of connections, memory starts to leak.

If you are using the finalizer (destructor) method, then make sure to call `Dispose()` inside it for the resources you want to deallocate. Thus, you will be doubly sure that your resources will be cleaned by GC even if someone forgets to release them in their code while using your class.

Out of curiosity, you must be asking yourself, what would happen if some exception occurs before `Dispose()` is called? Will that object be disposed of? The solution to this is to wrap that with a `try...finally` block, so that no matter what happens to the program, finally will be called and you can dispose of objects inside that. To make it simple, the framework has a beautiful thing called the `using` block which can be used as follows:

`Dispose()` versus `Close()`: Are you confused which one to call for an `SqlConnection` object? They are two different methods which solve different problems. `Close()` just closes the connection. You can reopen the connection using the same object. However, `Dispose()` closes the connection (calls `Close()` under the hood) and then deallocates the object from memory. You can't use the object anymore.

You can find out more about Garbage Collector at https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/.

WOW! eBook
www.wowebook.org

Consider an example

A simple `Order` class can have a default, parameterized, copy constructor along with a destructor (to destroy the object):

```
class Order
{
    public string ProductName { get; set; }
    public int Quantity { get; set; }

    public Order()
    {
       // Default Constructor
    }

    public Order(string productName, int quantity)
    {
       // Constructor with two arguments
       ProductName = productName;
       Quantity = quantity;
    }

    public Order(Order someOrder)
    {
       // Copy constructor
       ProductName = someOrder.ProductName;
       Quantity = someOrder.Quantity;
    }

    ~Order()
    {
       // Destructor or Finalizer
    }
}
```

You can see how the constructors are formed using and without the parameters. Notice the copy constructor, which takes an object of the same class as an argument and inside the body, assigns it's properties to the object in making.

The finalizer implicitly calls `Finalize` on the base class of the object. Therefore, when GC calls the Finalizer, that might call a method something like shown in the following:

```
protected override void Finalize()
{
  try
  {
    // Cleanup statements...
  }
  finally
```

WOW! eBook
www.wowebook.org

```
    {
       base.Finalize();
    }
  }
```

Let's verify this behavior with a code snippet inside a console app using .NET Core 2.0:

```
  class BaseClass
  {
    ~BaseClass()
    {
      System.Diagnostics.Trace.WriteLine("BaseClass's destructor is called.");
    }
  }
  class DeriveClass1 : BaseClass
  {
    ~DeriveClass1()
    {
      System.Diagnostics.Trace.WriteLine("DeriveClass1's destructor
          is called.");
    }
  }

 class DeriveClass2 : DeriveClass1
 {
    public DeriveClass2()
    {
      System.Diagnostics.Trace.WriteLine("DeriveClass2's constructor is called.");
    }

    ~DeriveClass2()
    {
       System.Diagnostics.Trace.WriteLine("DeriveClass2's destructor
        is called.");
    }
 }

 class Program
{
  static void Main(string[] args)
  {
     DeriveClass2 t = new DeriveClass2();

     // Unlike .NET Framework, .NET Core 2.0, as of now,
     // does not call GC on application termination
     // to finalise the objects.
     // So, we are trying to manually call GC
     // to see the output.
     System.GC.Collect();
  }
}
```

The `DeriveClass2` object is created first which logs the constructor messages. Then the `DeriveClass2` destructor is executed. Thus, it is destroyed first when the `Main` function execution completes. Additionally, the parent classes have destructors. As the child class (`DeriveClass2`) object is already destroyed, it runs the parent class

WOW! eBook
www.wowebook.org

destructors too. The following screenshot is taken from the output window of visual studio. Make sure you run the app in Release mode:

```
DeriveClass2's constructor is called.
DeriveClass2's destructor is called.
DeriveClass1's destructor is called.
BaseClass's destructor is called.
```

WOW! eBook
www.wowebook.org

Implementing the IDisposable Interface

It's important to learn how to implement `IDisposable` interface as you might work with a user-defined class in your project which will deal with unmanaged resources. The system defines classes using unmanaged resources implement `IDisposable` and expose Dispose so that we can easily call that method to dispose of the object like we saw in the code snippet for `SqlConnection` class.

There is a pattern called **Dispose Pattern**, which the developers must follow while implementing `IDisposable`. Let's explore that. I will go through it step by step.

WOW! eBook
www.wowebook.org

Step1 - Basic structure of class

We will have an `ExampleIDisposable` class which implements the `IDisposable` interface. I am not going to demonstrate usage of unmanaged resources, as our intention is to learn the pattern. I just have a console line inside the constructor telling that we are acquiring unmanaged resources.

```
class ExampleIDisposable : IDisposable
{
    public Dictionary<int, string> Chapters{ get; set; }
    public ExampleIDisposable(Dictionary<int, string> chapters)
    {
       // Managed resources
       Console.WriteLine("Managed Resources acquired");
       Chapters = chapters;

       // Some Unmanaged resources
       Console.WriteLine("Unmanaged Resources acquired");
    }

    public void Dispose()
    {
       Console.WriteLine("Someone called Dispose");

       // Dispose managed resources
       if (Chapters != null)
       {
          Chapters = null;
       }

      // Dispose unmanaged resources
    }
 }
```

You can see that the class contains one managed property which is initialized inside the constructor. We will print a line for it. Likewise, we might have some property using unmanaged resources declared in the class and given life by the constructor. Since we implemented `IDisposable`, we are bound to define the only method `Dispose()`. For now, we just have a console line inside this.

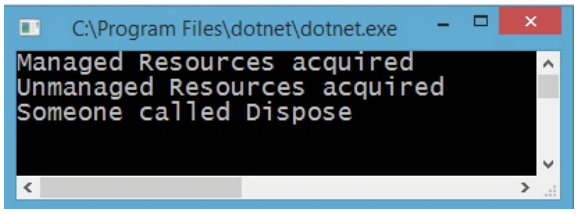
Let's try it out:

```
static void Main(string[] args)
{
   ExampleIDisposable disposable = new ExampleIDisposable(new Dictionary<int,
     string> {{ 5, "Object Composition" },
             { 6, "Object Lifetime" }
     });
```

WOW! eBook
www.wowebook.org

```
        disposable.Dispose();
        Console.ReadLine();
    }
```

This produces the following output:



We need to understand two important points before moving forward.

What happens when Finalizer is called by GC? When an object goes out of scope, that will be added to the Finalizer queue for Garbage Collector to take action on them to release it from memory. And we don't know when that will happen. If you kill the managed resources inside `Dispose()`, then we need to restrict the objects going into the finalizer queue thereby informing GC not to take action on them as they don't exist anymore. Also, that is an overhead for GC.

What if the developer forgets to call Dispose? Suppose, the developer who is using our class does not dispose of it. Still, we need to handle this situation. We can easily do that by calling `Dispose()` inside Finalizer, but wait! We need to ask the `Dispose()` method to kill unmanaged resources only, not managed ones because GC is there to automatically deal with them.

This is where another method comes into the picture.

WOW! eBook
www.wowebook.org

Step2 - Defining a Dispose Overload method

The `Dispose()` method we defined inside the class is going to help us while calling it directly through the object of the class. However, we need another overload of `Dispose()` inside the class which would answer the questions we previously discussed. Let's introduce that to our class.

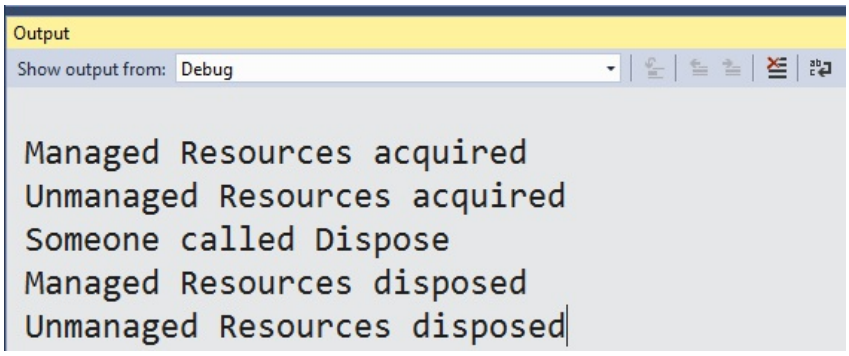
```
class ExampleIDisposable : IDisposable
{
  public Dictionary<int, string> Chapters { get; set; }
  public ExampleIDisposable(Dictionary<int, string> chapters)
  {
    // Managed resources
    System.Diagnostics.Trace.WriteLine("Managed Resources acquired");
    Chapters = chapters;

    // Some Unmanaged resources
     System.Diagnostics.Trace.WriteLine("Unmanaged Resources acquired");
  }
  public void Dispose()
   {
     System.Diagnostics.Trace.WriteLine("Someone called Dispose");
     Dispose(true);
     GC.SuppressFinalize(this);
   }
   public void Dispose(bool disposeManagedResources)
    {
      if (disposeManagedResources)
      {
        if (Chapters != null)
        {
          Chapters = null;
        }

        System.Diagnostics.Trace.WriteLine("Managed Resources disposed");
      }
      System.Diagnostics.Trace.WriteLine("Unmanaged Resources disposed");
    }

    ~ExampleIDisposable()
    {
      System.Diagnostics.Trace.WriteLine("Finalizer called: Managed
          resources will be cleaned");
      Dispose(false);
    }
  }
```

Modifications done to the `Dispose` methods are explained as follows:

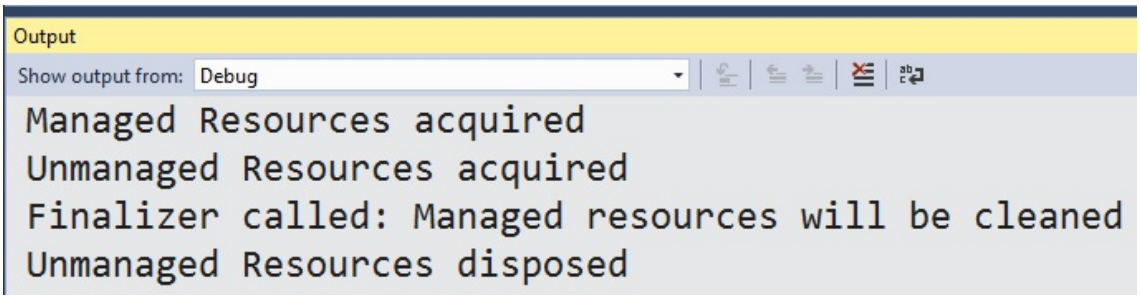
WOW! eBook
www.wowebook.org

- `public void Dispose()`: Now, we are asking the `Dispose(bool)` method to release all type of resources. `public void Dispose(). GC.SuppressFinalize(this)`; suppresses the GC finalizer call, as we have already disposed of everything inside the `Dispose(bool)`
- `public void Dispose(bool)`: This method is the important part of this pattern. With the `bool` param, it decides whether there is a need to kill the managed resources or not

I have replaced the console lines with trace lines so that the `main` method ends and we can see the lines on the output screen. If you just remove `Console.ReadLine();` from the `Main` method and run the app again, the output produced will be as follows:



Removal of the `Dispose()` calls from the `Main` method that is `disposable.Dispose();`, will result in something as follows. Note to call `GC.Collect();` at the end of `Main` method as we did in Step 1:



That means whenever developer forgets to dispose, the finalizer is called where we have called `Dispose(false);`, which eventually released unmanaged resources. Of course, finalizer will remove the managed resources automatically. You can see that the lines Someone called Dispose and Managed Resources disposed are missing in the last case.

WOW! eBook
www.wowebook.org

Step3 -Modifying the Dispose(bool) for Derived classes

As we have `Dispose(bool)` overload, it will be available directly on the object to call. There is no need to expose `Dispose(bool)` to the object for the direct call as we are internally calling it from `Dispose()` and Finalizer.

The user should not pass boolean and decide what to dispose of and how. The only thing they will do is to call `Dispose()` which will release all type of resources. For that reason, we will restrict the call to `Dispose(bool)` by changing the access specifier from `public` to `protected`.

The `Dispose(bool)` is a logical block for the class which is implementing `IDisposable`. Any class which is going to derive a base class implementing `IDisposable` might have their own custom logic for disposal. So, instead of adding another dispose method, they can just override the base class `Dispose(bool)`. To make that happen, we need to prepend a `virtual` keyword before the method name.

The preceding paragraphs call for a modification of our very well known method `Dispose(bool)`:

```
protected virtual void Dispose(bool disposeManagedResources)
{
  if (disposeManagedResources)
  {
    if (Chapters != null)
    {
       Chapters = null;
    }
    System.Diagnostics.Trace.WriteLine("Managed Resources disposed");
  }
  System.Diagnostics.Trace.WriteLine("Unmanaged Resources disposed");
}
```

WOW! eBook
www.wowebook.org

Step 4 - Handling duplicate Dispose calls

We should manage the scenario where the user might call Dispose more than once. If we don't tackle this situation, the subsequent calls will just be unnecessary executions for the runtime because the runtime will try to release an object which has been already disposed of.

We can easily put a flag inside the class which will indicate whether the object was disposed or not.

```
bool disposed = false;
protected virtual void Dispose(bool disposeManagedResources)
{
  if (disposed)
  {
    System.Diagnostics.Trace.WriteLine("Dispose(bool) already called");
     return;
  }
  if (disposeManagedResources)
    {
      if (Chapters != null)
      {
         Chapters = null;
      }
      System.Diagnostics.Trace.WriteLine("Managed Resources disposed");
   }
   System.Diagnostics.Trace.WriteLine("Unmanaged Resources disposed");
   disposed = true;
 }
```

Notice the `disposed` variable, which is used inside `Dispose(bool)`. We are checking if that is true inside the method. If true, then we return/exit from the method, otherwise execute the disposal codes. At last, we set it as true. So, for the first time `Dispose(bool)` is going to execute fully, after that, it will just return once called. This way, we are preventing disposing of the same object many times which is an overhead.

Let's modify the code to call `Dispose()` multiple times:

```
disposable.Dispose();
disposable.Dispose();
disposable.Dispose();
```

WOW! eBook
www.wowebook.org

This would give us the following output:

```
Output
Show output from: Debug
Managed Resources acquired
Unmanaged Resources acquired
Someone called Dispose
Managed Resources disposed
Unmanaged Resources disposed
Someone called Dispose
Dispose(bool) already called
Someone called Dispose
Dispose(bool) already called
```

You can see, for the first call, everything works as expected. The the next two subsequent Dispose() calls for the same object results in a simple return from the method. That is why we see two sets of Someone called Dispose and Dispose(bool) already called messages.

Alright, I would like to show you the final code after all these steps:

```
class ExampleIDisposable : IDisposable
{
    public Dictionary<int, string> Chapters { get; set; }
    bool disposed = false;

    public ExampleIDisposable(Dictionary<int, string> chapters)
    {
        // Managed resources
        System.Diagnostics.Trace.WriteLine("Managed Resources acquired");
        Chapters = chapters;

        // Some Unmanaged resources
        System.Diagnostics.Trace.WriteLine("Unmanaged Resources acquired");
    }

    public void Dispose()
    {
       System.Diagnostics.Trace.WriteLine("Someone called Dispose");

       Dispose(true);
       GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposeManagedResources)
    {
       if (disposed)
       {
           System.Diagnostics.Trace.WriteLine("Dispose(bool) already called");
           return;
       }

       if (disposeManagedResources)
       {
          if (Chapters != null)
          {
```

WOW! eBook
www.wowebook.org

```
                Chapters = null;
            }
            System.Diagnostics.Trace.WriteLine("Managed Resources
                disposed");
        }

        System.Diagnostics.Trace.WriteLine("Unmanaged Resources disposed");
        disposed = true;
    }

    ~ExampleIDisposable()
    {
      System.Diagnostics.Trace.WriteLine("Finalizer called: Managed
            resources will be cleaned");
      Dispose(false);
    }
  }
```

Don't forget that you can use the `using` statement with any class which implements `IDisposable`. For instance, let's write that for `ExampleIDisposable`:

```
using (ExampleIDisposable disposable =
    new ExampleIDisposable(new Dictionary<int, string> {
    { 5, "Object Composition" },
    { 6, "Object Lifetime" }
    }))
{
  // Do something with the "disposable" object.
}
```

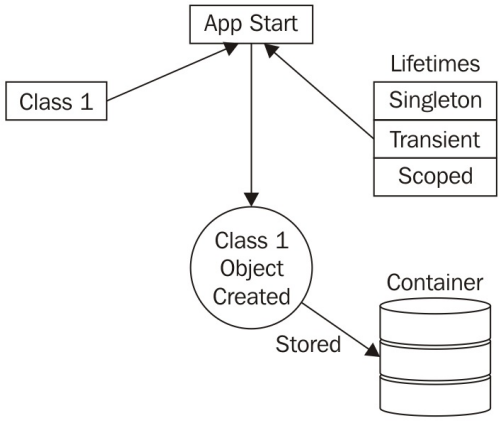
If you run this, it would produce the same result as shown in the first screenshot under *Step 2: Defining a Dispose Overload method* section.

WOW! eBook
www.wowebook.org

Object lifetime management in .NET Core

In previous chapters, we have already explored how Dependency Injection is built into .NET Core. Now that we learned how objects are managed by .NET Framework, let's shed some light on their lifestyle in .NET Core.

With just one line, I can say, in Startup, .NET Core takes one class, tags a lifetime to it as specified in code, then instantiates and stores it in the container or service collection. Consider the following screenshot:



We will be covering how the following is addressed in .NET Core:

- Object Creation.
- Object's Lifestyles.
- Object's Disposal after everything is done.

WOW! eBook
www.wowebook.org

Object creation

Typically in ASP.NET Core 2.0, an injected type is called a **service**. For example, the injected interface is called as `IServiceCollection` and we can add services, as we need, by using the `AddSingleton` method here. We will learn more about it soon:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddSingleton<IExampleService, ExampleService>();
}
```

The built-in DI framework in ASP.NET Core performs two important steps when we perform the preceding code:

- **Instantiation:** The object of the provided service (example: `ExampleService`) is instantiated so that it can be made available to controllers when they are called. The object gets injected through constructor injection or property injection. `IExampleService` will be an argument to the controller. The implementer of this interface `ExampleService` can be instantiated and injected. We will see the constructor in a while.
- **Lifetime Management:** Then it decides the lifetime (creation and disposal) of the object injected to the controller. There are different types of lifestyles served by the framework, which we will learn next.

Let's explore the three-lifetime modes served by *ASP.NET Core* by default. The following is a quick reference table for these lifestyles:

Lifestyle	Description	Disposal
Transient (temporary or short-lived)	A new instance will be created each time the service is requested.	Never
Scoped	An instance will be created for a defined scope. When out of scope is reached, another instance will be created, if requested. A simple scope can be stated as one `Web`	Disposed when the

WOW! eBook
www.wowebook.org

(extent or ranged)	`Request`. Any resource, requesting the instance in a particular `Web Request` will be served the same from the container.	scope ends
Singleton (single or individual)	One instance created by the container which will be used/shared throughout the app for every request.	Disposed when the container is disposed

We will look at each concept in detail now. Don't worry, if you are confused after reading this. After a well-designed example, you will definitely get the concept.

First of all, let's design our ASP.NET MVC Core app to use all these kinds of lifetimes. This example is similar to that given in the official documentation, which can be found at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

The main goal here is to understand how the instances stay during the incoming requests. We will investigate the instances from two places: one from a controller and another from a class. Both these guys will use the same interfaces as dependencies injected to their constructors. Let's go through things step by step.

WOW! eBook
www.wowebook.org

Designing the interfaces

First code an interface `IExampleService` with a `Guid` type property `ExampleId`. Then write four interfaces implementing this interface. We have named them according to a different type of lifetimes so that we can easily recognize them afterward:

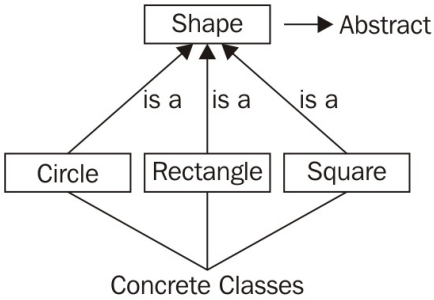
```
public interface IExampleService
{
  Guid ExampleId { get; }
}

public interface IExampleTransient : IExampleService
{
}
public interface IExampleScoped : IExampleService
{
}
public interface IExampleSingleton : IExampleService
{
}
public interface IExampleSingletonInstance : IExampleService
{
}
```

WOW! eBook
www.wowebook.org

The Concrete class

Abstract which means something which can be conceptualized whereas concrete means something which exists in reality and not abstract. Suppose we consider three Shapes like a circle, a rectangle, and a square. The shape seems like a concept, doesn't it? On the other hand a circle, a rectangle, and a square actually have the behavior of shape to represent a concrete idea. An Abstract class is incomplete in nature and when a class takes (or inherits) its behavior, it becomes complete which is denoted as **Concrete**.

Thus, Abstract (or incomplete) concept is meant to be completed by other classes inheriting it, forming *Concrete Classes*. An Abstract class is incomplete and just a concept, so instantiating it becomes meaningless. Shape, for instance, is not meant to be instantiated, rather it's meant to be inherited by actual shape classes. The following diagram shows the relationship:



A single `Example` class implements all these interfaces and defines the `ExampleId` guid. Two constructors are there: one which takes the `Guid` as a param and the other one is the default which initializes a new guid. All those interfaces will be resolved as this class in *Startup*. We are approaching that code soon:

```
using LifetimesExample.Interfaces;
using System;

namespace LifetimesExample.Models
{
  public class Example : IExampleScoped, IExampleSingleton,
     IExampleTransient, IExampleSingletonInstance
  {
    public Guid ExampleId { get; set; }
    public Example()
    {
      ExampleId = Guid.NewGuid();
    }
```

WOW! eBook
www.wowebook.org

```
      public Example(Guid exampleId)
      {
        ExampleId = exampleId;
      }
    }
  }
```

WOW! eBook
www.wowebook.org

The Service class

As we said to have a simple class using these interfaces, here is that class called as `ExampleService`. We have a constructor here, which is waiting for the interfaces to be injected and assigned to local interface type variables.

```
using LifetimesExample.Interfaces;
namespace LifetimesExample.Services
{
public class ExampleService
{
    public IExampleTransient TransientExample { get; }
    public IExampleScoped ScopedExample { get; }
    public IExampleSingleton SingletonExample { get; }
    public IExampleSingletonInstance SingletonInstanceExample { get; }

    public ExampleService(IExampleTransient transientExample,
        IExampleScoped scopedExample,
        IExampleSingleton singletonExample,
        IExampleSingletonInstance instanceExample)
    {
        TransientExample = transientExample;
        ScopedExample = scopedExample;
        SingletonExample = singletonExample;
        SingletonInstanceExample = instanceExample;
    }
  }
}
```

WOW! eBook
www.wowebook.org

The controller

The controller is almost the same as the service class with an additional reference of `ExampleService`. It has a constructor to initialize them. The properties of `ExampleService` will be printed on a view, that's why we are referring to that class:

```
using Microsoft.AspNetCore.Mvc;
using LifetimesExample.Services;
using LifetimesExample.Interfaces;

namespace LifetimesExample.Controllers
{
  public class ExampleController : Controller
  {
    private readonly ExampleService _exampleService;
    private readonly IExampleTransient _transientExample;
    private readonly IExampleScoped _scopedExample;
    private readonly IExampleSingleton _singletonExample;
    private readonly IExampleSingletonInstance _singletonInstanceExample;

    public ExampleController(ExampleService ExampleService,
        IExampleTransient transientExample,
        IExampleScoped scopedExample,
        IExampleSingleton singletonExample,
        IExampleSingletonInstance singletonInstanceExample)
    {
        _exampleService = ExampleService;
        _transientExample = transientExample;
        _scopedExample = scopedExample;
        _singletonExample = singletonExample;
        _singletonInstanceExample = singletonInstanceExample;
    }

    public IActionResult Index()
    {
        // viewbag contains controller-requested services
        ViewBag.Transient = _transientExample;
        ViewBag.Scoped = _scopedExample;
        ViewBag.Singleton = _singletonExample;
        ViewBag.SingletonInstance = _singletonInstanceExample;

        // Example service has its own requested services
        ViewBag.Service = _exampleService;

        return View();
    }
  }
}
```

Inside `Index()` action, we are returning all these values in `ViewBag`.

WOW! eBook
www.wowebook.org

View

Last, but not least, we will simply design a view which can show all these values so that we can start our observation. This would be `Index.cshtml` inside `Views/Example`.

```
@using LifetimesExample.Interfaces
@using LifetimesExample.Services

@{
  ViewData["Title"] = "Index";
 }

@{
  IExampleTransient transient = (IExampleTransient)ViewData["Transient"];
  IExampleTransient scoped = (IExampleTransient)ViewData["Scoped"];
  IExampleTransient singleton = (IExampleTransient)ViewData["Singleton"];
  IExampleTransient singletonInstance = (IExampleTransient)ViewData["SingletonIn
  ExampleService service = (ExampleService)ViewBag.Service;
}

<h2>Lifetimes</h2>

<h3>ExampleController Dependencies</h3>
<table>
 <tr>
    <th>Lifestyle</th>
    <th>Guid Value</th>
 </tr>
<tr>
    <td>Transient</td>
    <td>@transient.ExampleId</td>
</tr>
<tr>
    <td>Scoped</td>
    <td>@scoped.ExampleId</td>
</tr>
<tr>
    <td>Singleton</td>
    <td>@singleton.ExampleId</td>
</tr>
<tr>
    <td>Instance</td>
    <td>@singletonInstance.ExampleId</td>
</tr>
</table>

<h3>ExampleService Dependencies</h3>
<table>
 <tr>
    <th>Lifestyle</th>
    <th>Guid Value</th>
 </tr>
 <tr>
    <td>Transient</td>
    <td>@service.TransientExample.ExampleId</td>
```

WOW! eBook
www.wowebook.org

```
    </tr>
    <tr>
        <td>Scoped</td>
        <td>@service.ScopedExample.ExampleId</td>
    </tr>
    <tr>
        <td>Singleton</td>
        <td>@service.SingletonExample.ExampleId</td>
    </tr>
    <tr>
        <td>Instance</td>
        <td>@service.SingletonInstanceExample.ExampleId</td>
    </tr>
  </table>
```

Finally, we completed the codes. Are you sure? We forgot the main entry point. The one which decides what class will be resolved against which interface.

WOW! eBook
www.wowebook.org

Startup ConfigureServices

Now each type of lifestyle is added to the container resolved using `Example` class. The class `ExampleService` is resolved for itself. That means, whenever we need `ExampleService` anywhere in the app, to be injected, it will automatically assign an object of that class with all its properties:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddTransient<IExampleTransient, Example>();
    services.AddScoped<IExampleScoped, Example>();
    services.AddSingleton<IExampleSingleton, Example>();
    services.AddSingleton<IExampleSingletonInstance, Example>();
    services.AddSingleton(new Example(Guid.Empty));
    services.AddTransient<ExampleService, ExampleService>();
}
```

`Add***` methods (with different lifestyles) make sure that the objects are created according to the desired behavior expected from them. Once these objects are initialized, they can be injected where ever required.

If we run the app now, for the first request, we will get the following output:

WOW! eBook
www.wowebook.org

http://localhost:65472/example

localhost:65472/example

Lifetimes

ExampleController Dependencies

Lifestyle	Guid Value
Transient	9a18e7f1-a37d-4f18-9404-9d4ccd5a9584
Scoped	0af905dc-832e-4de2-b596-6b83e7ecda11
Singleton	f1f1b06b-3f6b-4e07-8cf6-9ead785401c1
Instance	00000000-0000-0000-0000-000000000000

ExampleService Dependencies

Lifestyle	Guid Value
Transient	092cf7e8-07c9-488f-977b-d570fcc72ba1
Scoped	0af905dc-832e-4de2-b596-6b83e7ecda11
Singleton	f1f1b06b-3f6b-4e07-8cf6-9ead785401c1
Instance	00000000-0000-0000-0000-000000000000

I opened the page in another tab (or you can just refresh the tab). I saw the following output:

WOW! eBook
www.wowebook.org

http://localhost:65472/example | http://localhost:65472/example

localhost:65472/example

Lifetimes

ExampleController Dependencies

Lifestyle	Guid Value
Transient	fd33c6ff-cabb-4555-840b-6cb33b05813a
Scoped	6b9f0c9c-33b6-4cef-985e-a69171db7f48
Singleton	f1f1b06b-3f6b-4e07-8cf6-9ead785401c1
Instance	00000000-0000-0000-0000-000000000000

ExampleService Dependencies

Lifestyle	Guid Value
Transient	0bf9a17a-e5ae-4f96-a410-eb35b751f962
Scoped	6b9f0c9c-33b6-4cef-985e-a69171db7f48
Singleton	f1f1b06b-3f6b-4e07-8cf6-9ead785401c1
Instance	00000000-0000-0000-0000-000000000000

WOW! eBook
www.wowebook.org

Object lifetimes

Let's examine all the lifetimes one by one according to the values we see in these screenshots. The values help us recognize the objects.

WOW! eBook
www.wowebook.org

Singleton

Looking at the screenshots and diagrams, the first thing which is clearly seen is the `Singleton` one. No matter how many requests you do after running the app, the object will be the same. It does not depend upon the Controller or the `Service` class.

WOW! eBook
www.wowebook.org

Scoped

For a particular request, the object is same throughout the scope. Notice how the object's guid value is different in the second request. However, the values are same for Controller and `Service` class in both the requests. Here the scope is the web request. When one more request is served, the object is recreated.

Creating a scope in code is possible though. There is an interface called `IServiceScopeFactory` which exposes the method `CreateScope`. `CreateScope` is of the type `IServiceScope`, which implements `IDisposable`. Here the `using` block comes to our rescue for disposal of the scoped instance:

```
var serviceProvider = services.BuildServiceProvider();
var serviceScopeFactory = serviceProvider.GetRequiredService<
      IServiceScopeFactory>();

IExampleScoped scopedOne;
IExampleScoped scopedTwo;

using (var scope = serviceScopeFactory.CreateScope())
{
  scopedOne = scope.ServiceProvider.GetService<IExampleScoped>();
}
using (var scope = serviceScopeFactory.CreateScope())
{
 scopedTwo = scope.ServiceProvider.GetService<IExampleScoped>();
}
```

We created two scoped instances in the preceding code using `CreateScope`. Both are independent of each other and are not shared within a request like a normal scoped instance. That is because we manually assigned the scope instead of the default Web Request scope.

WOW! eBook
www.wowebook.org

Transient

Simple! A new object is always created whenever requested from the container. In the screenshot, you can see the guid values are different in the Controller and the `Service` class even for a single request.

WOW! eBook
www.wowebook.org

Instance

The last one is a special case of Singleton, where the user creates the object and provides that to the `AddSingleton` method. So, we are explicitly creating the object of the `Example` class (`services.AddSingleton(new Example(Guid.Empty))`) and asking DI framework to register it as a Singleton. In this case, we are sending `Guid.Empty`. Thus, an empty guid is assigned which stays unchanged for all requests.

WOW! eBook
www.wowebook.org

Object disposal

When we register a service directly with the `Add***` method, the container is responsible for creating the object and managing the lifetime. Essentially, it calls `Dispose` for the objects, which implement `IDisposable` according to the lifetime.

Consider the following example where `ServiceDisposable` implements `IDisposable` and we told services to manage its lifetime as `Scoped`. Therefore, it will create the instance, then make it available across all resources in the app for a single request. At last, it disposes of it when the request ends:

```
public class ServiceDisposable : IDisposable {}
public void ConfigureServices(IServiceCollection services)
{
  services.AddScoped(ServiceDisposable);
}
```

However, when we create the object explicitly and provide that to the DI, we need to handle it's disposal our self. In the following case, we have to manually call `Dispose()` somewhere to dispose the instance:

```
public class ServiceDisposable : IDisposable {}
public void ConfigureServices(IServiceCollection services)
{
  services.AddScoped(new ServiceDisposable());
}
```

WOW! eBook
www.wowebook.org

When to choose what?

It's important how we deal with different objects in our application so that they consume limited memory space and resources, yet optimize the performance.

- Objects which require more space and utilize large server resources should not be recreated, rather they should be reused. Database objects, for instance, should be reused for all the requests which follow a `Singleton` pattern.
- Operations which runs in a batch or a loop can be reused in a particular request but should be recreated for another. That indicates a Scoped lifetime.
- The `Model` and `View Model` classes should be instantiated every time we try to form them. We can't reuse a Model class object while doing a CRUD operation, else we might end up with the wrong values going into the database. Of course, that is `Transient` as it lives for a short period of time.

WOW! eBook
www.wowebook.org

Relationship and dependencies among lifetimes

In this section, we will dig more to find out the relation between the lifetimes. The reason is pretty clear. While instantiating our classes with a different type of lifestyle, we might face situations where we need dependency of one class in another class. But the catch here is that they may not be following a similar lifestyle. So, what happens to the instance which is following a scoped lifestyle referred inside a `Singleton` class. Does that behave as Scoped?

Let's get our hands dirty and do some code modifications to inject dependencies into one another.

WOW! eBook
www.wowebook.org

Singleton depending on Scoped and Transient

First, we need to add two new properties to the existing `IExampleSingleton` interface:

```
public interface IExampleSingleton : IExampleService
{
    Guid ScopedExampleId { get; }
    Guid TransientExampleId { get; }
}
```

Next, we want to design a new class explicitly for all lifestyles. As we have planned, let's inject Transient and Scoped dependencies into this `Singleton` class through the constructor. The properties defined for dependent lifestyles are assigned values accordingly from the arguments.

```
using System;
namespace LifetimesExample
{
  public class ExampleSingleton : IExampleSingleton
  {
    public Guid ExampleId { get; set; }
    public Guid ScopedExampleId { get; set; }
    public Guid TransientExampleId { get; set; }

    public ExampleSingleton(IExampleTransient transient, IExampleScoped scoped)
    {
        ExampleId = Guid.NewGuid();
        ScopedExampleId = scoped.ExampleId;
        TransientExampleId = transient.ExampleId;
    }
  }
  public class ExampleScoped : IExampleScoped
  {
    public Guid ExampleId { get; set; }

    public ExampleScoped()
    {
        ExampleId = Guid.NewGuid();
    }
  }
  public class ExampleTransient : IExampleTransient
  {
    public Guid ExampleId { get; set; }

    public ExampleTransient()
    {
        ExampleId = Guid.NewGuid();
    }
   }
```

WOW! eBook
www.wowebook.org

```
    }
```

I have defined all classes in a single place just for the readability purposes of this book. You should ideally add them in different files every time.

The controller is the next place where we need to have an action which will return the view where we will show the values.

```
using Microsoft.AspNetCore.Mvc;
namespace LifetimesExample.Controllers
{
  public class ExampleController : Controller
  {
    private readonly ExampleService _exampleService;
    private readonly IExampleTransient _transientExample;
    private readonly IExampleScoped _scopedExample;
    private readonly IExampleSingleton _singletonExample;

    public ExampleController(ExampleService ExampleService,
        IExampleTransient transientExample,
        IExampleScoped scopedExample,
        IExampleSingleton singletonExample)
    {
        _exampleService = ExampleService;
        _transientExample = transientExample;
        _scopedExample = scopedExample;
        _singletonExample = singletonExample;
    }

    public IActionResult SingletonDependencies()
    {
        ViewBag.Singleton = _singletonExample;

        ViewBag.Service = _exampleService;

        return View("Singleton");
    }
  }
}
```

It's similar to what we have done with the `Index` action. The difference is that we have removed the `SingletonInstance` reference and returned a view called as `Singleton`.

The view would look something like the following:

```
@{
  ViewData["Title"] = "Index";
}

@{
  IExampleSingleton singleton = (IExampleSingleton)ViewData["Singleton"];
  ExampleService service = (ExampleService)ViewBag.Service;
}
<h2>Singleton Lifetime Dependencies</h2>
```

WOW! eBook
www.wowebook.org

```
<h3>ExampleController</h3>

<h5><u>Singleton ExampleId: @singleton.ExampleId</u></h5>

<table>
  <tr>
      <th>Dependencies</th>
      <th>Guid Value</th>
  </tr>

  <tr>
      <td>Scoped Dependency</td>
      <td>@singleton.ScopedExampleId</td>
  </tr>
  <tr>
      <td>Transient Dependency</td>
      <td>@singleton.TransientExampleId</td>
  </tr>
</table>

<h3>ExampleService</h3>

<h5><u>Singleton ExampleId: @service.SingletonExample.ExampleId</u></h5>

<table>
  <tr>
      <th>Dependencies</th>
      <th>Guid Value</th>
  </tr>

  <tr>
      <td>Scoped Dependency</td>
      <td>@service.SingletonExample.ScopedExampleId</td>
  </tr>
  <tr>
      <td>Transient Dependency</td>
      <td>@service.SingletonExample.TransientExampleId</td>
  </tr>
</table>
```

So, I am trying to print the `ExampleId` of `Singleton` object and properties related to the dependent objects (`Transient` and `Scoped`). I have omitted the styles from this code which is just to make the table look cool.

It's time to tell the `Startup ConfigureService` to register the classes with the appropriate lifestyle:

```
public void ConfigureServices(IServiceCollection services)
{
        // Add framework services.
        services.AddMvc();

        services.AddSingleton<IExampleSingleton, ExampleSingleton>();
        services.AddScoped<IExampleScoped, ExampleScoped>();
        services.AddTransient<IExampleTransient, ExampleTransient>();
```

WOW! eBook
www.wowebook.org

```
        services.AddTransient<ExampleService, ExampleService>();
}
```

Voila! We are done. Let's check the output. I have pasted the screenshots of two requests done to the controller side by side so that we can easily mark the findings:

localhost:65472/example/SingletonDependencies

Singleton Lifetime Dependencies

ExampleController **Request 1**

Singleton ExampleId: 3506d3ac-3948-4047-b6dd-d0f7eed0cdbf

Dependencies	Guid Value
Scoped Dependency	4173e92d-430a-4fa0-b47c-81f54fbc8630
Transient Dependency	a4fa8b4f-d6d2-4928-8e43-df62ea0d3cd2

ExampleService

Singleton ExampleId: 3506d3ac-3948-4047-b6dd-d0f7eed0cdbf

Dependencies	Guid Value
Scoped Dependency	4173e92d-430a-4fa0-b47c-81f54fbc8630
Transient Dependency	a4fa8b4f-d6d2-4928-8e43-df62ea0d3cd2

localhost:65472/example/SingletonDependencies

Singleton Lifetime Dependencies

ExampleController **Request 2**

Singleton ExampleId: 3506d3ac-3948-4047-b6dd-d0f7eed0cdbf

Dependencies	Guid Value
Scoped Dependency	4173e92d-430a-4fa0-b47c-81f54fbc8630
Transient Dependency	a4fa8b4f-d6d2-4928-8e43-df62ea0d3cd2

ExampleService

Singleton ExampleId: 3506d3ac-3948-4047-b6dd-d0f7eed0cdbf

Dependencies	Guid Value
Scoped Dependency	4173e92d-430a-4fa0-b47c-81f54fbc8630
Transient Dependency	a4fa8b4f-d6d2-4928-8e43-df62ea0d3cd2

Observation: The `Singleton` object is shared across two requests as the underlined `ExampleId` value is same.

Hold on! There is something weird. Dependent objects' values are also the same across request. Notice the values in red blocks. Even though those classes are registered as `Scoped` and `Transient`, they are behaving like a `Singleton`. That means the normal lifestyle of these objects is tampered.

Inference: It's not recommended to refer `Scoped` and `Transient` lifestyle classes inside a `Singleton` class as they will lose their usual behavior and become Singletons.

Obviously, a `Singleton` class can depend on another a `Singleton` class. Likewise, other lifestyles follow the same rule. So, a `Scoped` class can refer another `Scoped` class and a `Transient` can refer another `Transient`. They all will behave as expected when executed.

WOW! eBook
www.wowebook.org

Scoped depending on Singleton and Transient

Similarly, we can test for dependencies inside a `Scoped` class. We will start by adding two properties to the interface `IExampleScoped`:

```
public interface IExampleScoped : IExampleService
{
        Guid SingletonExampleId { get; }
        Guid TransientExampleId { get; }
}
```

`ExampleScoped` should now implement these two properties. Also, the interfaces related to `Transient` and `Singleton` need to be injected into the `constructor`:

```
    public class ExampleScoped : IExampleScoped
    {
        public Guid ExampleId { get; set; }
        public Guid SingletonExampleId { get; set; }
        public Guid TransientExampleId { get; set; }

        public ExampleScoped(IExampleTransient transient, IExampleSingleton singleto
        {
          ExampleId = Guid.NewGuid();
          SingletonExampleId = singleton.ExampleId;
          TransientExampleId = transient.ExampleId;
        }
    }
```

A new action is added which will return the view named as `Scoped`:

```
    public IActionResult ScopedDependencies()
    {
        ViewBag.Scoped = _scopedExample;

        ViewBag.Service = _exampleService;

        return View("Scoped");
    }
```

Looks like we are done. Let's run the app:

WOW! eBook
www.wowebook.org

localhost:65472/example/ScopedDependencies 133% Search

An unhandled exception occurred while processing the request.

InvalidOperationException: A circular dependency was detected for the service of type 'LifetimesExample.IExampleScoped'.

Microsoft.Extensions.DependencyInjection.ServiceProvider.GetServiceCallSite(Type serviceType, ISet<Type> callSiteChain)

Oops! We see an exception screen which says a Circular Dependency was detected.

> ***Circular Dependency**, as the name suggests, is a class which depends on another, which in turn, depends on the first class. We designed everything to test the dependencies of other lifestyles in Scoped lifestyle, but we forgot one thing before doing that. Previously, we added a dependency of `Scoped` class inside a `Singleton` class and now, if you see the preceding `ExampleScoped` constructor, we now inject `IExampleSingleton`, which is resolved as the `Singleton` class `ExampleSingleton`. That's how it became circular.*

Therefore, we need to remove the dependencies from the `Singleton` class to test. We can also test by doing another interface and class for `Singleton`. So, when you fix the code, we will get the following output. I am not going to write the codes of the `View` here. It's pretty much the same as we done in `Singleton`. We just need to print `ExampleId`, `SingletonExampleId` and `TransientExampleId`:

WOW! eBook
www.wowebook.org

localhost:65472/example/ScopedDependencies

Scoped Lifetime Dependencies

ExampleController

Request 1

Scoped ExampleId: 4c2857f5-ae27-417b-b7b3-4a619231ddfc

Dependencies	Guid Value
Singleton Dependency	2081d240-2915-4558-9356-16c19777a43f
Transient Dependency	e67aee53-331f-4a79-8128-3046d0c8edda

ExampleService

Scoped ExampleId: 4c2857f5-ae27-417b-b7b3-4a619231ddfc

Dependencies	Guid Value
Singleton Dependency	2081d240-2915-4558-9356-16c19777a43f
Transient Dependency	e67aee53-331f-4a79-8128-3046d0c8edda

localhost:65472/example/ScopedDependencies

Scoped Lifetime Dependencies

ExampleController

Request 2

Scoped ExampleId: 65e2c65c-281b-4fda-adb3-e390420b84e3

Dependencies	Guid Value
Singleton Dependency	2081d240-2915-4558-9356-16c19777a43f
Transient Dependency	1d0ffa38-c857-4edf-8b72-78b467950968

ExampleService

Scoped ExampleId: 65e2c65c-281b-4fda-adb3-e390420b84e3

Dependencies	Guid Value
Singleton Dependency	2081d240-2915-4558-9356-16c19777a43f
Transient Dependency	1d0ffa38-c857-4edf-8b72-78b467950968

Observation: In the red boxes, we have transient object values. That is because they are not `Transient` anymore and behave like `Scoped`, as the values are the same in a request, which is not what `Transient` looks like. It should be different every time it is requested. But in the case of `Singleton` dependency, it is the same across requests, which not only satisfies the `Singleton` paradigm but also behaves like scoped in a particular request.

Inference: That's why it is recommended to use `Scoped` and `Singleton` dependencies inside a `Scoped` class, but not Transient.

WOW! eBook
www.wowebook.org

Transient depending on Singleton and Scoped

The following the same pattern for `Transient`, we will design the interface and class as required. The interface will look like the following:

```
public interface IExampleTransient : IExampleService
{
        Guid SingletonExampleId { get; }
        Guid ScopedExampleId { get; }
}
```

Next is the `Transient` class depending on `Singleton` and `Scoped`.

```
public class ExampleTransient : IExampleTransient
{
        public Guid ExampleId { get; set; }
        public Guid SingletonExampleId { get; set; }
        public Guid ScopedExampleId { get; set; }

        public ExampleTransient(IExampleSingleton singleton, IExampleScoped scoped)
        {
                ExampleId = Guid.NewGuid();
                SingletonExampleId = singleton.ExampleId;
                ScopedExampleId = scoped.ExampleId;
        }
}
```

Last, but not the least, a new action to render the view `Transient`:

```
    public IActionResult TransientDependencies()
   {
        ViewBag.Transient = _transientExample;

        ViewBag.Service = _exampleService;

        return View("Transient");
   }
```

Running this after designing the view will end up with something like the following:

WOW! eBook
www.wowebook.org

localhost:65472/example/TransientDependencies

Transient Lifetime Dependencies

ExampleController Request 1

Transient ExampleId: b56ea918-5baf-458f-b5cd-2d3fa40a0b8f

Dependencies	Guid Value
Singleton Dependency	6ad2cff8-dabb-4ca1-8a75-ae69328f5819
Scoped Dependency	8a69f773-508f-4610-9041-987b9f4ada0e

ExampleService

Transient ExampleId: df3a3deb-b365-48a6-a326-1ad7d36acd8b

Dependencies	Guid Value
Singleton Dependency	6ad2cff8-dabb-4ca1-8a75-ae69328f5819
Scoped Dependency	8a69f773-508f-4610-9041-987b9f4ada0e

localhost:65472/example/TransientDependencies

Transient Lifetime Dependencies

ExampleController Request 2

Transient ExampleId: 8c20b689-0f72-4204-b947-fe671e2e9a78

Dependencies	Guid Value
Singleton Dependency	6ad2cff8-dabb-4ca1-8a75-ae69328f5819
Scoped Dependency	2dd6c085-6d26-412a-8f93-2935f25c3ff8

ExampleService

Transient ExampleId: 43f0415c-c8c1-473d-a5a6-563001e4318b

Dependencies	Guid Value
Singleton Dependency	6ad2cff8-dabb-4ca1-8a75-ae69328f5819
Scoped Dependency	2dd6c085-6d26-412a-8f93-2935f25c3ff8

You can see how the `Transient ExampleId` differs in inside and outside the request.

Observation: You must be wondering why I don't have red boxes in this picture. That is because everything looks perfect. `Singleton` is the same everywhere and `Scoped` is the same in a particular request, however, changes in the next request.

Inference: That means, both these dependencies carry their usual characteristics when injected inside a *Transient* class. Also, another *Transient* dependency will definitely work if injected to a *Transient* class.

Saying all this, I would say these are the recommended patterns to follow while injecting one lifestyle into another. We should be careful while designing the classes and their lifestyle. Either we will end up in a Circular Dependency or lose the appropriate behavior of the lifestyle injected as we saw in the preceding examples. We can still mix it up as long as we are aware of the consequences.

With just a simple table, I can represent the whole point:

WOW! eBook
www.wowebook.org

Lifestyles	Injected Lifestyle		
	Transient	Scoped	Singleton
Transient	✓	✓	✓
Scoped	✗	✓	✓
Singleton	✗	✗	✓

WOW! eBook
www.wowebook.org

Summary

In this chapter, we learned about how objects are created and destroyed by .NET Framework. Creation and destruction mechanism were discussed. *Garbage Collector* plays an important role in the automatic disposal through Finalizers, which we analyzed through examples.

Most importantly, we saw a step-by-step walkthrough to manually dispose of objects as soon as we are done by implementing the `IDisposal` interface.

After that, we explored the different lifetimes maintained by objects in .NET Core. We saw examples using the controller and service classes. We learned how the objects are created and disposed of. Most importantly, we experimented with the adaptability of different lifestyles with each other.

Interception, another pillar of DI, will be covered in `Chapter 7`, *Interception*.

WOW! eBook
www.wowebook.org

Interception

We have already discussed two pillars of the DI ecosystem in the previous two chapters. Those pillars provided us with ways to identify dependencies, register them and then manage their lifetimes as required. In this chapter, we are going to cover something different which addresses the architectural problem of DI.

An application without security, logging, caching, transaction management, and so on seems very incomplete. When we write codes to cover up all these functionalities for the app, we might end up writing repetitive codes in each module all around the application. Even if Dependency Injection has an existence in our application, solving the purpose of managing the requirements of a class and trying to provide all the types needed for logging, or others as mentioned previously, violates SRP and DRY. Thus, we need a different approach to solve this problem and that is **Interception**. Simply put, Interception will help us dynamically inject blocks of code wherever required.

Now, you might have a question regarding when and how we can inject this code. Let's hold on to the *how* for now and focus on the *when*. We will discuss the *how* soon. To answer *when*, the injection can be done before or after, or it can even totally replace the actual method execution.

In this chapter, we will understand what Interception is and how it can be implemented. We will find out possible techniques to interrupt the flow of execution. Definitely, we will discuss why we even need *Interception*.

Patterns and principles of Interception in .NET Core is the next thing we are going to illustrate. We will even extend the concept of Interception in ASP.NET MVC Core using filters and middleware.

We will cover the following topics in this chapter:

- Cross-cutting concerns and aspects
- **Aspect-oriented programming** (**AOP**): Its types, advantages, disadvantages, and process
- Demos of each type of AOP
- Interception adoption in ASP.NET Core MVC using filters and middleware

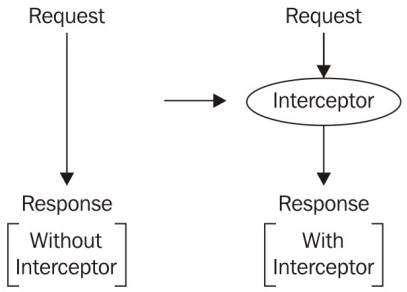
WOW! eBook
www.wowebook.org

Introducing Interception

In this section, I will introduce you to *Interception* along with the connection it has with the *Dependency Injection*.

Interception is another important pillar. Interception is the process through which we can easily intercept the call between a consumer and a service in order to execute certain codes before or after the service is invoked.

The following diagram shows what happens with and without the Interception process:



If we consider the preceding request as a call that comes to the index action of a controller named `CompanyController` and before the company details are served to the client, we can run a few bookkeeping tasks such as *Who requested the company details?*, *When was the call made?*, *Is the user authorized to receive the company details?*, and so on. For all these tasks, we have to stop the flow in the controller to do all these operations, and after we are done, we can again resume the main task of returning data. But all these tasks can't be written inside the controller action, because the main responsibility of the action is to return the company details. If we write everything inside the action method, it would violate SRP.

WOW! eBook
www.wowebook.org

The decorator

Let's find out by a class that requires basic logging as a part of its operations. A `CompanyRepository` might have a method to assign a `User` to a `Company`.

```
public void AssignUserToCompany(int userId, int companyId)
{
  // Database operation to assign User to Company.
}
```

Now, a client asks you to log some important information while this operation takes place, just for bookkeeping and future reference. The information could be the start time of the operation, end time of the operation, who requested for the operation, exceptions if any, and so on. The instant solution that comes to your mind might look something like the following:

```
public void AssignUserToCompany(int userId, int companyId)
{
  _logger.Log("Assign to User started.");
  // Database operation to assign User to Company.
  _logger.Log("Assign to User ended.");
}
```

Notice the bold lines. `ILogger` can be injected to the repository through a constructor for log operations. We can also put `try...catch` block and log exceptions. Everything looks good, but don't you think the class is doing more than what it is intended for and that we are violating the **Single Responsibility Principle** (**SRP**). Therefore, let's think of another solution. How about the following?

```
public class LoggingCompanyRepository : CompanyRepository
{
  private readonly CompanyRepository _companyReposiory;
  private readonly ILogger _logger;
  public LoggingCompanyRepository(CompanyRepository
     companyRepository, ILogger logger)
  {
    this._companyReposiory = companyRepository ??
       throw new ArgumentNullException("companyRepository");
    this._logger = logger ?? throw new
       ArgumentNullException("logger");
 }
 public override void AssignUserToCompany(int userId,
     int companyId)
 {
    _logger.Log("Assign to User started.");
    _companyReposiory.AssignUserToCompany(userId,
        companyId);
```

WOW! eBook
www.wowebook.org

```
        _logger.Log("Assign to User ended.");
    }
}
```

We introduced a decorated `CompanyRepository` named as `LoggingCompanyRepository`, which is managing the logging part. It is initialized by taking repository and logger dependencies and executing the method as needed with log entries. Following the Decorator Pattern, the new class tries to coordinate the work between the repository and logger.

The final task would be to compose the decorator, which has the following steps:

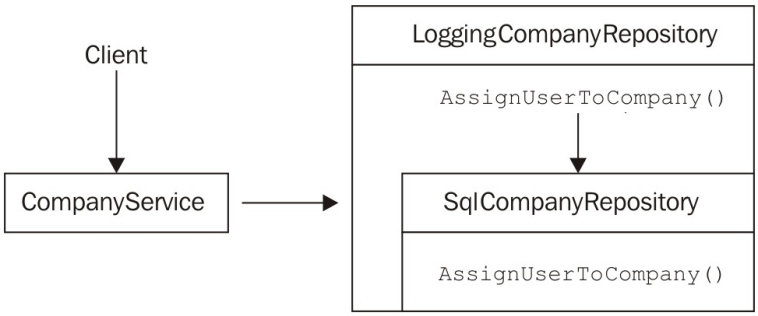
1. We can create the instances of `SqlCompanyRepository` and `SqlLogger` which derive from related abstractions.
2. Then, we create the instance of the decorator by injecting these dependencies.
3. We return the `CompanyService` instance with the decorated repository.

Refer to the following code for these steps:

```
public ICompanyService ResolveCompanyService()
{
  CompanyRepository companyRepository = new
      SqlCompanyRepository("ConnectionString");
  Controllers.ILogger logger = new SqlLogger();
  CompanyRepository loggingCompanyRepository = new
      LoggingCompanyRepository(companyRepository, logger);
  return new CompanyService(loggingCompanyRepository);
}
```

This is what Interception is all about. We are able to interrupt the call made to `AssignUserToCompany` of the concrete class `SqlCompanyRepository` because the `Service` is now composed of a decorator with the concrete class, instead of the direct code block. When you call the method, it would first go to the decorator's method and then call the concrete class method.

The following diagram explains the flow:



WOW! eBook
www.wowebook.org

Patterns and principles for Interception

You must be getting a hint that I want you to taste the DI from the perspective of not only injecting dependencies, but also managing and maintaining code for a good architecture. We explored the Decorator Pattern in the last section, which enabled us to decorate the concrete implementation with a little additional code without even touching any classes in the application. This forms the basis of Interception.

With Interception, we clearly followed SOLID principles of Software Design. The moment we started designing the decorator class, that class played a vital role in the SRP. `CompanyRepository` is managing the database part and the `LoggingCompanyRepository` takes the responsibility for logging.

The **Liskov Substitution Principle** (**LSP**) demands that the consumer should not feel the change of any implementations of the dependency. We achieved this by the decorator implementing the same abstraction that the `Service` demanded, which is `CompanyRepository` so that we would be able to replace the original one `SqlCompanyRepository` with the decorator one `LoggingCompanyRepository`, without any breakage to the service codes.

As we were doing all the preceding steps, we did not change any class for implementing the decorator. Instead, we extended, which strongly followed the Open/Closed Principle. There is no such necessity to touch the `Service` class or the consumer.

The principles behind the Interception following SOLID gives us a hint as to how the Decorator Pattern is closely related to DI, since DI supports Interception. The Decorator Pattern is the fundamental building block of the concept, but the way we implemented it does not solve the architectural problems in hand. Let's analyze the associated benefits and problems.

WOW! eBook
www.wowebook.org

Benefits of the Decorator approach

With the Decorator approach, we achieved many benefits. Refer to the following list:

- The `Service` does not know what repository it is receiving. Notice that `CompanyRepository` and `LoggingCompanyRespository` are both declared as `CompanyRepository` instances.
- Just because the types are the same, there is no need to do any changes to the `Service` class. Moreover, we added logging behavior to the `CompanyRepository` class without even changing it. It supports the Open/Closed principle. All classes are intact.
- We were able to intercept the concrete `SqlCompanyRepository` methods in order to have log entries. Thus, we did not violate the SRP of the repository.

However, we did not realize a few problems with this approach. This structure would lead us to a bad architecture in the long run. Let's find out why.

Instead of dealing with the instances manually, always use DI Container to resolve the dependencies. We will see how we can work with DI Container for Interception in a while.

WOW! eBook
www.wowebook.org

Problems of the Decorator approach

In a typical project, we will have many of these repositories and services. We will definitely encounter the following problems:

- Imagine the number of lines of codes you would have to write to compose the decorators in order to do Interceptions for hundreds of classes
- The next thing is very important, which is actually the basis of our topic in discussion. What if you have to log in a thousand methods of those hundred classes. This is common and we need it in almost every method. Think of the work involved and the size of your app.

Due to all these, we will end up going for a different approach for intercepting calls to methods using Aspects. DI Containers make it easy for us to design aspects and attach Interceptors. Let's move on.

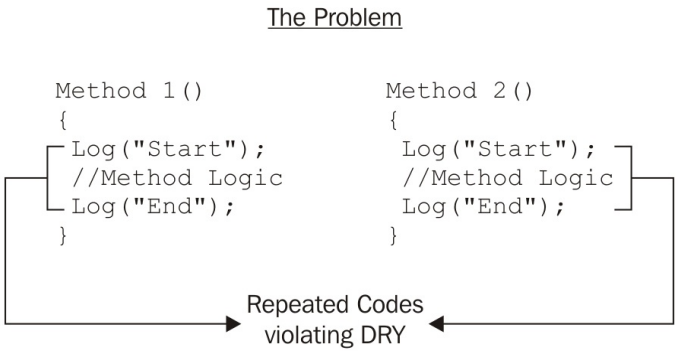
WOW! eBook
www.wowebook.org

Aspect-oriented programming

Object-oriented programming (**OOP**) deals with techniques to solve real-world problems with underlying object models. To design a software, we need to adopt OOP along with good design methodologies in order to make it maintainable, consistent and reliable. There are many such patterns or problems we will encounter during the application development which are of design decisions, that can neither be solved by OOP practices nor with procedural approaches.

In this chapter, we will address one design decision that can be managed easily by Aspects. To clarify, we can consider a simple example which is very often seen in codes and that is **logging**. We do logging in almost every method. This means you are repeating codes everyday violating the **DRY** (**Don't Repeat Yourself**) principle.

The following diagram shows you the repeated codes of logging commonly found in your codes:

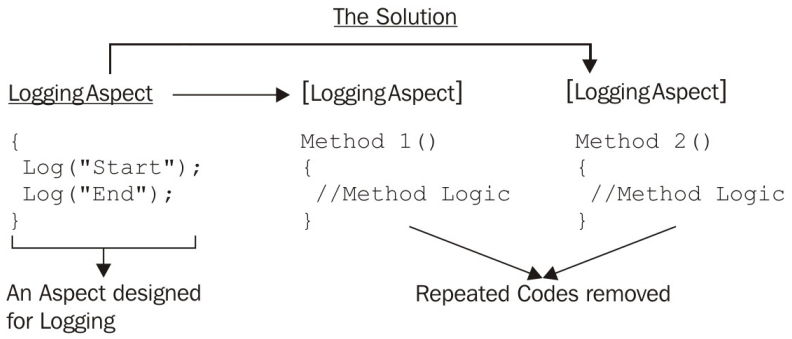


If I tell you that we can package those logging lines into one modular piece of code and just ask all the methods that need logging, to just import that while running, what would be your reaction? Yes, that is what an aspect is all about. So, we can write an aspect to manage logging. The pattern we follow in order to implement an aspect in our app is known as **Aspect-oriented programming** (**AOP**). It aims to separate or modularise these repeated codes from the core functionality.

The following is a sneak peak of the solution with an aspect called `LoggingAspect`, which packages all the logging codes inside it and is attached to the methods as an attribute above the method (written like `[LoggingAspect]`). Don't worry about the rules for how the logging lines are applied to the methods at the particular location such

WOW! eBook
www.wowebook.org

as *start* or *end* for now. We will learn all these soon, which are the concepts behind AOP:



When we follow AOP in our application, we can easily attach these modules automatically to our methods. Thus, developers have all the freedom to only focus on the business logic or core concerns inside the methods, instead of worrying about all the repetitive code.

WOW! eBook
www.wowebook.org

Cross-cutting concerns

A **concern** can be defined as a part of a system providing some specific functionality. If you consider a typical example of saving a user in your system, it may have a business logic something like the following:

1. Validate all fields of user for proper datatype, `null` values, and so on.
2. Check for the user existence in the system by a unique value such as `Email` or `UserName`.
3. If a user does not exist, create the user, otherwise update the user.

All these steps are denoted as **Business Logic**, which might be different in different applications since it depends on your requirements and design. So, our `SaveUser` method's principal duty is to perform all these steps, which can be defined as a **Main Concern**.

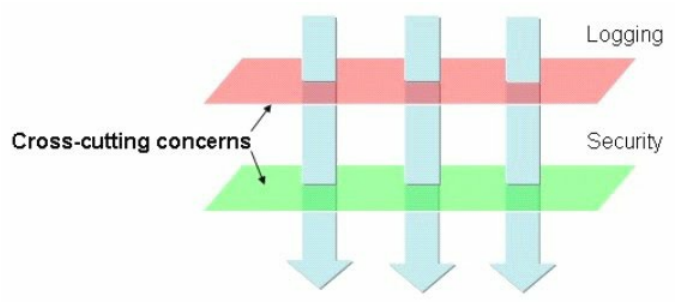
However, in a typical application, these are not the only things that happen inside a `SaveUser` method. You might have to do the following during saving a user:

1. Check whether the logged-in user is authorized to save the user.
2. Log in text/database that `SaveUser` started.
3. [Perform the *Business Logic* (all the steps previously defined)] - Main Concern.
4. Log in text/database that the `SaveUser` method succeeded.

Steps 1, 2, and 3 are the steps which are secondary and don't align with the Main Concern. But these steps can't be ignored. Moreover, these particular steps are performed in almost every method of the application.

While our method is trying to run the Main Concern, these guys try to interfere and cut the execution and do something else. That's why they are known as **Cross-Cutting Concerns**. These concerns are located throughout the application and affect the entire architecture. Have a look at the following diagram where the arrows indicate the Main Concern and the rectangular areas represent the Cross-Cutting Concerns:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

Aspect

In simple words, when we see some repetitive patterned codes throughout our app, that's the clue for us to consider them as Cross-Cutting Concerns. These are the code patches, which do not fit with the Main Concern inside the method in execution.

When I say *aspect* in the programming paradigm, it basically states the behavioral pattern independent of the actual task/concern at hand. So, when you package the Cross-Cutting Concerns into one modular piece that can be injected to the methods, you actually design an aspect.

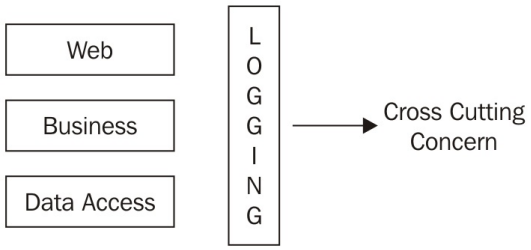
For instance, we want to log at certain lines inside many methods:

```
public void SaveUser(User user)
{
  Log.Debug("SaveUser started");
  try
  {
    // Service call to save the user.
  }
  catch(Exception ex)
  {
    Log.Error(ex, "Exception in SaveUser Method");
    throw;
  }
  finally
  {
      Log.Debug("SaveUser Method completed.");
  }
}
```

Clearly, `Log` class methods help us to log debug steps and exceptions inside a method. This particular pattern can be seen often at many places in the app. That is what a Cross-Cutting Concern is. The actual task at hand for the `SaveUser` method is to update the user details such as `UserName`, `Email`, `Password`, and so on, but this method also takes the responsibility to log some lines into files, which violates the SRP. Also, when these kinds of patterns are repeated across the project, it does not follow the DRY principle.

Look at the following diagram, which depicts logging as a common paradigm across layers of the application:

WOW! eBook
www.wowebook.org



So, here, AOP comes to our rescue which encapsulates this pattern to log the start, log the end, and log the exception, and then wrap it around the `SaveUser` method.

Another common scenario is wrapping our database operations with a `Transaction` block consisting of `Begin Transaction`, and `Commit/Rollback Transaction`. How about having someone else handling that part for us, while we concentrate on the core codes to operate on the database?

Cross-cutting concerns can be commonly found in the application where you log steps, handle exceptions, manage cache, access control, and so on. For example, while handling exceptions, you wrap your code using a `try..catch` *block so that you are able to operate on that exception found (log to text file/database or send email to admin, and so on). This particular pattern is needed inside every method. Now we need to find out a way to ask (more specifically wrap) every method that we need for exception handling, instead of adding the* `try..catch` *lines inside it. Thus, that wrapped modular piece can be termed as an aspect, which eventually wraps all methods instructed to it to perform the exception handling, instead of the methods themselves doing the same.*

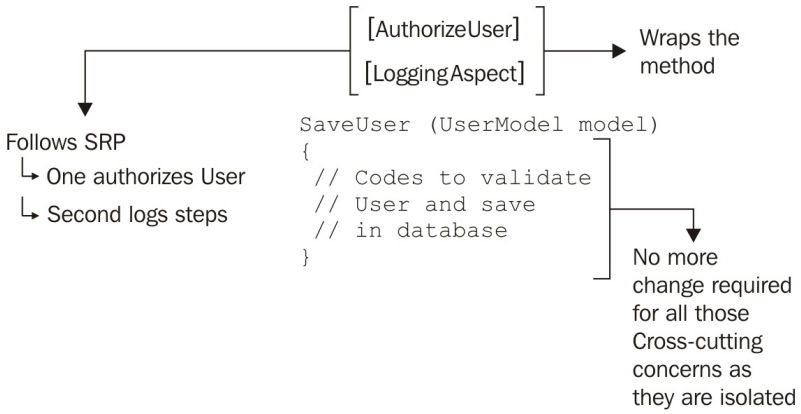
WOW! eBook
www.wowebook.org

Aspect characteristics

When we try to encapsulate these Cross-Cutting Concerns to create aspects, we actually make sure they follow some characteristics:

- **Wrapper:** All these aspects will be a wrapper around some business functionalities.
- **Single Responsibility Principle (SRP):** The wrapper is concerned only about one particular task. For example, logging will only do the task of logging and nothing else.
- **Decorator Pattern:** The aspect then attached with the existing functions with the Decorator Pattern.
    - **Open Close Principle:** When these repetitive codes were in business functions and we if needed any change in them in the future, then we would have to change the business function as it contains those pieces, which violates the Open Close Principle. Now, when we isolated those pieces to a module, the business functions became *Open* for extension, but *Close* to change.

Refer to the following diagram for a pictorial explanation:



AOP is not a competitor of OOP. Both are totally different. One is not a replacement for the other. With all these characteristics, AOP helps us to maintain a good structure for the project, hence resulting in a good OOP practice.

WOW! eBook
www.wowebook.org

Advantages

Let me summarise the important advantages of AOP:

- **Increases modularity:** Common functionality is centralized into independent modules and is attached those modules or aspects at many places in the app.
- **Manages Cross-Cutting Concerns:** The concerns which are scattered comes to one place which can be managed easily without making any code change to the actual business codes.
- **Better architecture:** Separation of these concerns into singly responsible aspects helps us to architect and organise business needs without unnecessarily repeating ourselves in the codes.

WOW! eBook
www.wowebook.org

Aspect attached locations

There are basically three locations where an Aspect can be attached to a method:

- **On Start:** When we want an aspect to execute immediately before the underlying function execution.
- **On Error:** Obviously, an aspect at this location will run only if an exception occurs in the method.
- **On Success:** Runs immediately after the method execution. However, it is limited to the functions which don't throw exceptions.

WOW! eBook
www.wowebook.org

Types of AOP

There are two techniques in AOP:

- **Interceptors:** Dynamic, which attaches the interceptors during runtime
- **IL Code Weaving:** Static, which runs after compilation and inserts code into the assembly

Static (post-compilation) ones include Fody, SheepAspect, Mono.Cecil, and PostSharp. Static might be faster, but we get more flexibility in a dynamic one, as we can change the code dynamically.

Applying the aspects to the code depends on the framework used. There are different techniques to attach the aspects such as writing attributes, XML configuration, and fluent interfaces.

WOW! eBook
www.wowebook.org

Investigating Interception

To intercept means to prevent something or someone from reaching an intended destination.

Interception (a programming paradigm), helps us to design aspects and inject cross-cutting concerns wherever demanded at the runtime. Using an Interceptor, we can easily intercept the calls to methods and properties in a class. To achieve this, we usually end up using an **Inversion Of Control** (**IoC**) container.

IoC provides the class functionality and then wraps it with our requested interceptor. Suppose somewhere in the code you ask for a class (say `Student`) to the IoC container of a specific type (say `IStudent`), which has an aspect for logging, then the IoC container would be able to provide the class (`Student`) with an interceptor decorated with it for the aspect demanded.

Interceptors are managed by external components that create dynamic decorators dealing with wrapping the aspects around your existing business components.

The main advantage of Interceptors is that they are not bound by the compilation process and don't modify our assemblies post-build. In other words, it's just the configurations for IoC containers, which you can easily take to another project instead of taking the DLLs and compiling them again.

However, Interceptors are configurable to work either at runtime or compile time.

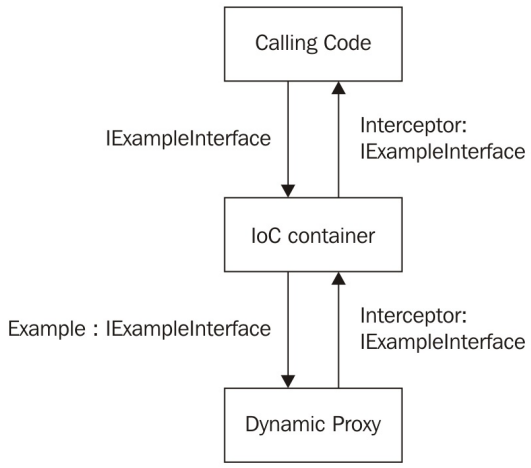
Unity, Ninject, Spring.NET, Castle Windsor, LinFu, Autofac, LOOM.NET, Seasar, and so on are some IoC containers that allow injecting Interception at compile time or runtime.

These IoC containers use a Dynamic Proxy to create in-memory decorators that wrap your existing code. Those Dynamic Proxies are responsible for code execution. Thus, it allows execution of the wrapper aspect as well as the underlying code being wrapped.

WOW! eBook
www.wowebook.org

The Interception process

The Interception process can be depicted as follows:



Here is what happens:

1. The calling code asks IoC container about the type it is looking for. For example, the `IExampleInterface` type.
2. IoC container now tries to match the concrete implementation of the type requested, which it can return back to the calling code that is `Example` class. While it is doing this, it recognizes that the type is already configured to use an interceptor.
3. Instead of directly returning to the calling code, the IoC container now sends the Interceptor and the implementation class for the requested type to the Dynamic Proxy.
4. The Dynamic Proxy now takes the concrete class and wraps it with the interceptor. Then it generates a class implementing the type initially requested with an instance of the concrete class given by the IoC container and the interceptor. After that, it returns back that implementation of the requested type to the IoC container. That is the `Interceptor` class.
5. The IoC container now sends back the `Interceptor` class generated by Dynamic Proxy to the calling code.
6. Calling code executes the returned class which in turn runs the interceptor and the underlying code of the concrete class `Example`.

WOW! eBook
www.wowebook.org

Castle Windsor

Castle Windsor is an Inversion of Control Container. This library is a part of the open source project known as the **Castle Project**. The Castle Project (http://www.castleproject.org/) provides reusable libraries for many purposes. Castle has many components and Windsor is the DI CONTAINER component of the Castle Project. However, it can be used independently of other Castle components.

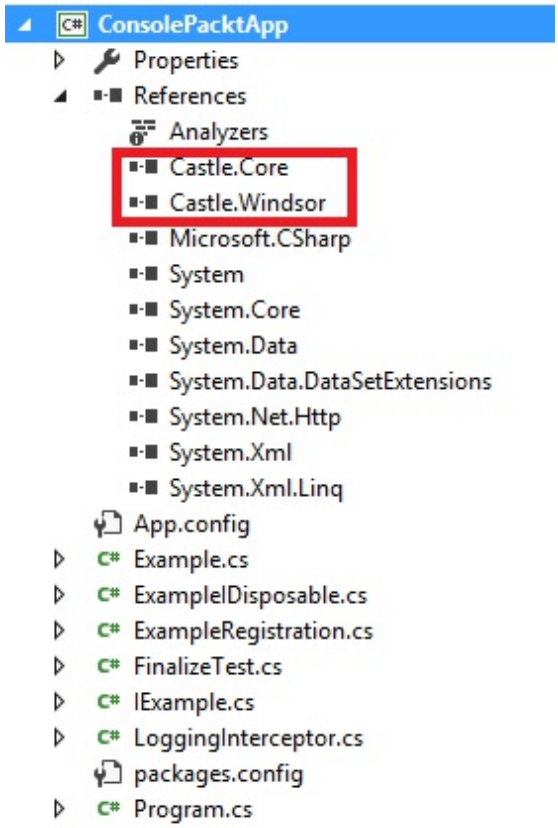
We have many other libraries available on the market such as Microsoft Unity, Autofac, Ninject, and so on. Each framework offers slightly different advantages and features. However, at the bottom, they implement the same for most of the core concepts, including Type Registration, Resolution, and Injection. You can, no doubt use any of these to apply Interception in your application.

There is no such logic as to why we are going to use Castle Windsor though. Using this tool, we can easily attach Interceptors to our codes. We will see that in a moment.

WOW! eBook
www.wowebook.org

Demonstration for using the Castle Windsor

Let's start to implement one Interceptor using the `Nuget` package Castle Windsor. First of all, create a Console App, go to Nuget Package Manager and install the Castle Windsor package. After successful installation, your Project Reference will look like the following:



So, it installs `Castle.Core` and `Castle.Windsor`. Now we are ready to create an Interceptor.

WOW! eBook
www.wowebook.org

Creating an Interceptor

We will do a simple Logging Interceptor which will log steps as well as exceptions for us. Let's name that `LoggingInterceptor.cs`. To qualify as an Interceptor, the class should implement the `Interceptor` interface. The only method inside this interface is `Intercept` which takes `IInvocation` as an argument.

The following code block illustrates what I said:

```
using Castle.DynamicProxy;
using System;
namespace ConsolePacktApp
{
  public class LoggingInterceptor : IInterceptor
  {
    public void Intercept(IInvocation invocation)
    {
      invocation.Proceed();
    }
  }
}
```

The call `invocation.Proceed()` just calls the underlying method which is supposed to be intercepted. That means, when any method which is registered to use this interceptor, it will come to this method, which in turn, calls the same method from here with the `Proceed()`.

To investigate the actual benefit of having an interceptor, we will add more codes here in the following block:

```
using Castle.DynamicProxy;
using System;
namespace ConsolePacktApp
{
  public class LoggingInterceptor : IInterceptor
  {
    public void Intercept(IInvocation invocation)
    {
        try
        {
            Console.WriteLine("Log Interceptor Starts");
            invocation.Proceed();
            Console.WriteLine("Log Interceptor Success");
        }
        catch (Exception e)
        {
            Console.WriteLine("Log Interceptor Exception");
            throw;
```

WOW! eBook
www.wowebook.org

```
            }
            finally
            {
                Console.WriteLine("Log Interceptor Exit");
            }
        }
    }
}
```

Next is to design an interface and a concrete class:

```
using System;
namespace ConsolePacktApp
{
  public interface IExample
  {
    void PrintName(string FirstName, string LastName);
  }

  public class Example : IExample
  {
    public void PrintName(string FirstName, string LastName)
    {
        Console.WriteLine($"Name is {FirstName} {LastName}");
    }
  }
}
```

So, the `PrintName()` method prints the name using `FirstName` and `LastName`. We will intercept this method and try to add log steps using an interceptor.

WOW! eBook
www.wowebook.org

Attaching an Interceptor

The most important part now is `ExampleRegistration`, which implements the interface `IRegistration` of `Castle.MicroKernel.Registration` namespace.

This is the place where we register `LoggingInterceptor` (refer to the first comment in the following code block) and then declare that `Example`, class which will be intercepted by `LoggingInterceptor` (refer to the second comment):

```
using Castle.MicroKernel.Registration;
using Castle.MicroKernel;
using Castle.Core;
namespace ConsolePacktApp
{
  public class ExampleRegistration : IRegistration
  {
    public void Register(IKernelInternal kernel)
    {
        // 1. Interceptor Registration
        kernel.Register(
            Component.For<LoggingInterceptor>()
                .ImplementedBy<LoggingInterceptor>());

        // 2. Interceptor attached with Example Class.
        kernel.Register(
            Component.For<IExample>()
            .ImplementedBy<Example>()
            .Interceptors(InterceptorReference.ForType<LoggingInterceptor>
               ()).Anywhere);
    }
  }
}
```

Last but not the least, the `main` method:

```
using Castle.Windsor;
using System;
namespace ConsolePacktApp
{
  class Program
  {
    private static IWindsorContainer _container;
    static void Main(string[] args)
    {
        _container = new WindsorContainer();
        _container.Register(new ExampleRegistration());

        var example = _container.Resolve<IExample>();

        try
        {
```

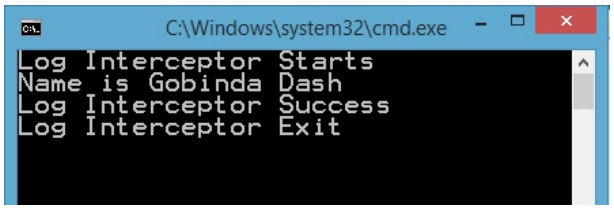
WOW! eBook
www.wowebook.org

```
                example.PrintName("Gobinda", "Dash");
            }
            catch (Exception ex)
            {
            }

            Console.ReadKey();
        }
    }
}
```

We take an `IWindsorContainer` instance. First, we register our `ExampleRegistration` class which contains our target method and interceptor configurations.
Then, `_container.Resolve()` helps us to get the required instance implementing `IExample`.

As we have already defined that `Example` is the class implementing `IExample` in the `Register()` method of the `ExampleRegistration` class, one instance of `Example` class is created. Then, we have a `PrintName()` call inside the `try` block.

Time to run the app and see the output:



Clearly, before printing the name, the first thing that executes is the log Starts message. Secondly, the actual method execution which prints the name. After that, as we expected, the Success and Exit messages. Exit means it exited from the interceptor.

Obviously, we did not get any exception, so that message got skipped.

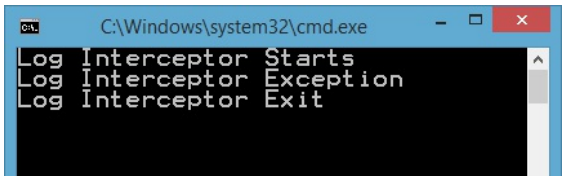
Now, let's try to see that. The modified `Example` class will be like the following with a `throw new Exception()` code line:

```
using System;
namespace ConsolePacktApp
{
  public class Example : IExample
  {
    public void PrintName(string FirstName, string LastName)
    {
        throw new Exception();
        Console.WriteLine($"Name is {FirstName} {LastName}");
    }
```

WOW! eBook
www.wowebook.org

```
        }
    }
```

Forcefully, we are throwing one exception before the `Console.WriteLine`. We will see the following output when we run this:



Interestingly, we don't see the Success log message, but the Exit message is printed and the actual execution of the method which was printing the name did not happen either. That is because the Exit message will definitely get printed no matter what happens to the calling code as that is inside the `finally` block of the interceptor.

We manually threw an exception before the actual code execution to print the name, which resulted in printing the exception log message along with Start and Exit.

Multiple Interceptors can be attached to a single class. There are two types of interceptor registrations, coded as follows. You can use either the following:

```
kernel.Register(
  Component.For()
 .ImplementedBy()
 .Interceptors<LoggingInterceptor,
     AnotherInterceptor>());
```

Or you can use this one:

```
kernel.Register(
 Component.For<IExample>()
 .ImplementedBy<Example>()
 .Interceptors(new InterceptorReference[] {
 InterceptorReference.ForType<LoggingInterceptor>(),
 InterceptorReference.ForType<AnotherInterceptor>() }).Anywhere);
```

WOW! eBook
www.wowebook.org

Intermediate Language (IL) Weaving

In this type of AOP, the aspects are attached to the underlying code after the application has been compiled. This post-compilation process alters the assemblies at the IL level so that the aspect hook points are called where they are configured.

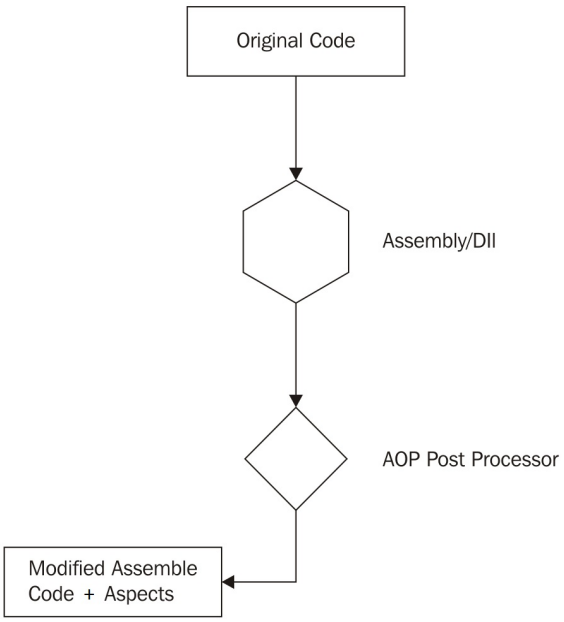
If I explain in simple words, it is a process by which the aspect codes are interjected into the original code after compilation, but that happens only to the IL code (original source code remains intact) and packaged into the assemblies. Your original code will be intact. However, along with your code, the aspect blocks will be included and the assembly gets the modified code. Unlike Interception, where the codes are interjected at the runtime, this process is static and codes are included beforehand.

The most widely used tool for weaving is PostSharp, which we will demo in a while. Others include LOOM.NET, Fody, SheepAspect, Mono.Cecil, and so on.

WOW! eBook
www.wowebook.org

IL Weaving process

Since I explained the process, let's see that in action in the following diagram:



Simple, isn't it! Now that we understand how the process goes along, it's time to get our hands dirty on some real code and try it out.

WOW! eBook
www.wowebook.org

Creating an aspect

Before starting up, we need to add the `Nuget` package named as PostSharp to the project.

To create an aspect, we need to design the class which would inherit from the `OnMethodBoundaryAspect` aspect class, which is an abstract class inside the `PostSharp.Aspects` namespace. This can be seen in the following code block:

```
using PostSharp.Aspects;
using System;
namespace ConsolePacktApp
{
  [Serializable]
  class LoggingWeaverAspect : OnMethodBoundaryAspect
  {
    public override void OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine("Inside OnEntry");
    }

    public override void OnExit(MethodExecutionArgs args)
    {
        Console.WriteLine("Inside OnExit");
    }

    public override void OnException(MethodExecutionArgs args)
    {
        Console.WriteLine("Inside OnException");
    }

    public override void OnSuccess(MethodExecutionArgs args)
    {
        Console.WriteLine("Inside OnSuccess");
    }
  }
}
```

`OnMethodBoundaryAspect` helps to write code which can execute before or after a method execution or in the case of an exception. As you can see from the previous class, we are overriding the `OnEntry`, `OnExit`, `OnException` and `OnSucess` methods. There are other aspect classes which can solve the different purposes.

Update the `main` method as follows:

```
static void Main(string[] args)
{
  try
  {
```

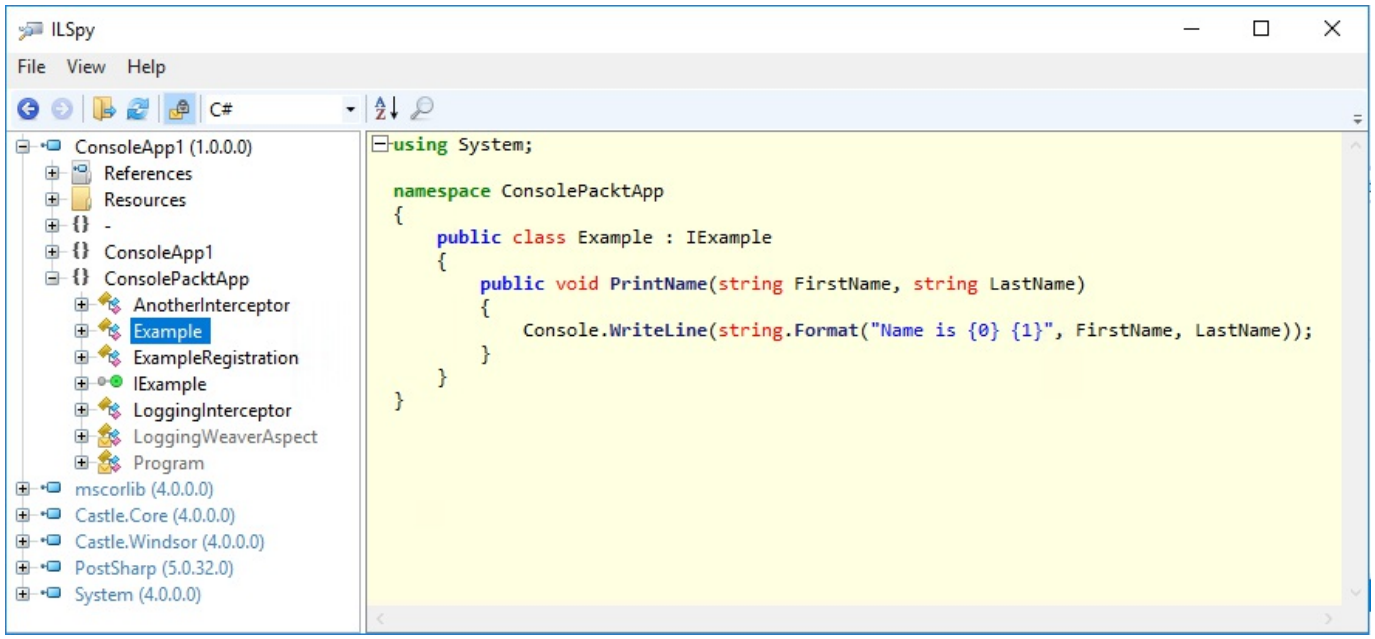
WOW! eBook
www.wowebook.org

```
        Example example = new Example();
        example.PrintName("Gobinda", "Dash");
      }
      catch
      {
      }
      Console.ReadKey();
    }
```

Now, when you run the app, nothing happens. We get a confirmation if we can see the generated assembly codes.

ILSpy is the open-source .NET assembly browser and decompiler. You can download it from `http://ilspy.net/`*. This will help us see the actual codes packaged inside the assemblies.*

When you run the ILSpy application and then select the assembly of our app, it will look something like the following:



You can see the `Example` class has code intact. No other code got inserted into the assembly. That's because we have not yet attached the Weaver with our `Example` class. Let's do that next.

If you encounter a PostSharp license error while building the app, that means you have to purchase one license. You can also use the free license, which has certain limitations. You can find the download page at `https://www.postsharp.net/essentials`*.*

WOW! eBook
www.wowebook.org

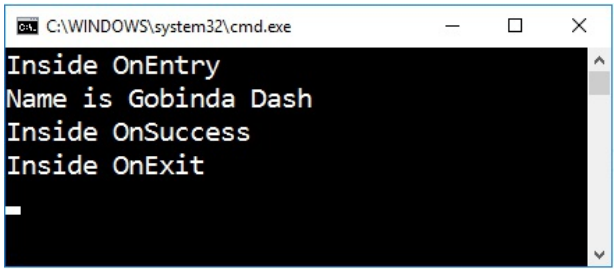
Attaching the aspect

Attaching the aspect is very easy. You can simply add the weaver as an attribute to the classes you want to wrap the aspect to.

Notice the bold line above the `PrintName` method in the following code snippet:

```
using System;
namespace ConsolePacktApp
{
  public class Example : IExample
  {
    [LoggingWeaverAspect]
    public void PrintName(string FirstName, string LastName)
    {
        Console.WriteLine($"Name is {FirstName} {LastName}");
    }
  }
}
```

We attributed the `[LoggingWeaverAspect]` around the `Example` class. Now, let's run it and see the output:



Have a look at the assembly code as follows:

WOW! eBook
www.wowebook.org

```
using PostSharp.ImplementationDetails_e6d251a4;
using System;

namespace ConsolePacktApp
{
    public class Example : IExample
    {
        public void PrintName(string FirstName, string LastName)
        {
            <>z__a_1.a0.OnEntry(null);
            try
            {
                Console.WriteLine(string.Format("Name is {0} {1}", FirstName, LastName));
                <>z__a_1.a0.OnSuccess(null);
            }
            catch (Exception CS$0$0__exception_2D)
            {
                <>z__a_1.a0.OnException(null);
                throw;
            }
            finally
            {
                <>z__a_1.a0.OnExit(null);
            }
        }
    }
}
```

The red boxes here indicate the code which was inserted into the assembly after you built the application. Since we had one `OnException` in the weaver aspect, the `PrintName()` method now contains a `try...catch` block. Finally, `OnEntry` is at the beginning and `OnExit` is inside. After the `Console.Writeline()`, we can see the `OnSuccess` call.

If we get an exception before the actual execution of the method (before `Console.WriteLine()` in this case), we will see the following output without the success message:

```
Inside OnEntry
Inside OnException
Inside OnExit
```

Now that we've had a sneak peek at the concepts, let's explore how Interception is implemented in .NET Core.

WOW! eBook
www.wowebook.org

Interception in ASP.NET Core

ASP.NET Core implements the concept of Interception to interrupt the calls to controller actions and request-response pipeline. We can do this by different techniques known as filters and middleware. We will discuss each one with examples next.

WOW! eBook
www.wowebook.org

Filters

Filters allow ASP.NET Core to intercept action methods. You can configure one global filter which would run every time one controller action is requested, or have individual unique filters for some action methods.

The filters would implement one of the filter interface residing in the `Microsoft.AspNet.Mvc.Filters` namespace. Let's see a simple filter skeleton as follows:

```
using Microsoft.AspNetCore.Mvc.Filters;
namespace FiltersAndMiddlewares.Filters
{
  public class SomeFilter : IActionFilter
  {
    public void OnActionExecuted(ActionExecutedContext context)
    {
      // Do something.
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
      // Do something.
    }
  }
}
```

As you can see in the preceding example, `SomeFilter` implements `IActionFilter`. `OnActionExecuting` and `OnActionExecuted`, as the names suggest, would run when an action is being executed and completes execution respectively.

If you recall the principle of Interception (which says that the *process by which we can run some code before or after the method which is being executed or totally replace that*), you will notice that these `IActionFilter` methods also follow the principle and are intended to intercept the calls to a controller action method.

WOW! eBook
www.wowebook.org

Global filter

Suppose you want to create an interceptor which would intercept all the action methods. Sounds like a common or globally declared code block that can be termed as a **Global filter**. Registering a filter to the MVC filters using the service collection inside `ConfigureServices` would make a filter globally executable or interceptable.

Refer to the following code, which is the only line to convert a normal filter to a Global filter except the definition of the filter, which of course you need to code:

```
services.AddMvc(mvc => mvc.Filters.AddService(
    typeof(SomeGlobalFilter)));
```

`SomeGlobalFilter` can also be injected with dependencies. The following code block can be considered as a Global filter, where `ISomeService` is injected using the most popular Constructor Injection Pattern:

```
using FiltersAndMiddlewares.Interfaces;
using Microsoft.AspNetCore.Mvc.Filters;
namespace FiltersAndMiddlewares.Filters
{
  public class SomeGlobalFilter : IActionFilter
  {
    public SomeGlobalFilter(ISomeService service)
    {
        // Do something with the service.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something.
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something.
    }
  }
}
```

Did you notice one thing? The parameterized constructor in `SomeGlobalFilter` is injected with the dependency of type `ISomeService`. Not optional though, but it can be done if we need any of the required dependency in filters. Imagine, if we need to add a log entry in the database whenever an action starts execution, something like a bookkeeping of what events occur in what sequence. To achieve this, we can inject a service and then use its methods to operate on the database.

WOW! eBook
www.wowebook.org

Attributes

Another way to attach filters is to write attributes for required actions. When you do global, it applies to all the actions. However, if we want to attach to certain actions only, then we can write attributes. Let's take a look at an action method which is attributed with `SomeFilter`:

```
[SomeFilter]
0 references
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

The compiler is complaining. If you hover, it will say SomeFilter is not an attribute class. It actually expects the `SomeFilter` like as follows:

```
public class SomeFilterAttribute : Attribute {}
```

Notice the name of the filter which was changed to `SomeFilterAttribute` and derived from `Attribute` abstract class. This is by syntax. Also, attributes require parameterless constructors. But to intercept, we need to implement some filter interface like `IActionFilter`.

To overcome this, we can take the help of the `TypeFilterAttribute` class which derives from the `Attribute` class and takes a type through its constructor. Thus, the following code shows the correct pattern to use a filter as an attribute:

```
[TypeFilter(typeof (SomeFilter))]
public IActionResult About()
{
  ViewData["Message"] = "Your application description page.";
  return View();
}
```

A couple of important things to note here are as follows:

- We have not registered `SomeFilter` in the entry point of the app; still, it works
- The `TypeFilter` attribute helps to create an instance of `SomeFilter`

WOW! eBook
www.wowebook.org

As we are playing with .NET Core, we should get the filter instance from the service registrations, instead of creating it dynamically through `TypeFilter`. That is where the `ServiceFilterAttribute` class comes onto the scene. Let's see how we can modify the code to use `ServiceFilter`:

```
[ServiceFilter(typeof (SomeFilter))]
public IActionResult About()
{
  ViewData["Message"] = "Your application description page.";
  return View();
}
```

Are you sure this change will work? No, it won't. When you run the app, you will see the error captured as follows:

```
InvalidOperationException: No service for type 'PacktDIExamples.Filters.SomeFilter' has been registered.
Microsoft.Extensions.DependencyInjection.ServiceProviderServiceExtensions.GetRequiredService(IServiceProvider provider, Type serviceType)
```

Ah, I know! Now you got what we missed. This is a very familiar exception when you are working with DI. As I said, `ServiceFilter` finds the filter instance from the service collection; if you don't register it, it will definitely throw an exception.

The following code is what you need to make it work. It is not mandatory to make it transient though, it depends on your scenario:

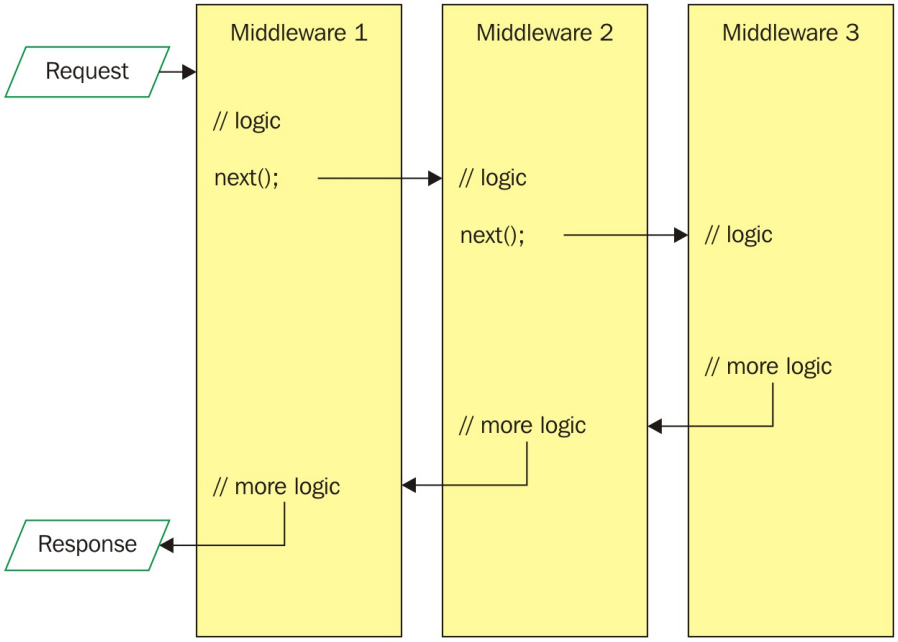
```
services.AddTransient<SomeFilter>();
```

WOW! eBook
www.wowebook.org

Middleware

Middleware, what it is known for, intercepts the pipeline of execution, and the developer can do anything before the response is served to the client. We will see how exactly we can implement Middlewares to intercept in the next section.

Middleware in ASP.NET core follows the Interception technique to interject aspects between request and response. It's basically some block of code which is registered with the application pipeline. The following diagram shows how Request goes through Middlewares one by one before Response comes out:



A large amount of Middleware can be inserted into the pipeline. Each Middleware decides whether to pass the execution to the next one and can perform some logic before and after the invocation of the next component. These components are designed to solve specific purposes like logging, exception handling, authorizing, and so on.

WOW! eBook
www.wowebook.org

Registration

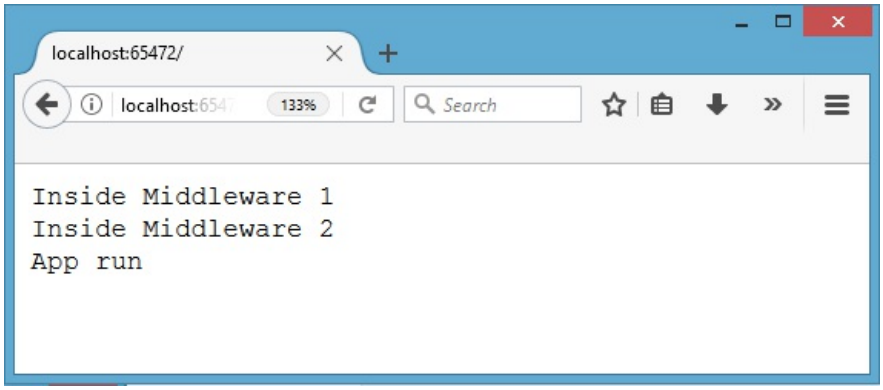
The `IApplicationBuilder` interface helps us register the Middleware using the `app.Use` inside the `Configure()`. Let's check a simple code block:

```
public void Configure(IApplicationBuilder app)
{
  var response = string.Empty;
  app.Use(async (context, next) =>
  {
    response += "Inside Middleware 1\n";
    await next.Invoke();
  });
   app.Use(async (context, next) =>
   {
    response += "Inside Middleware 2\n";
    await next.Invoke();
   });
   app.Run(async context =>
   {
     response += "App run\n";
     await context.Response.WriteAsync(response);
   });
}
```

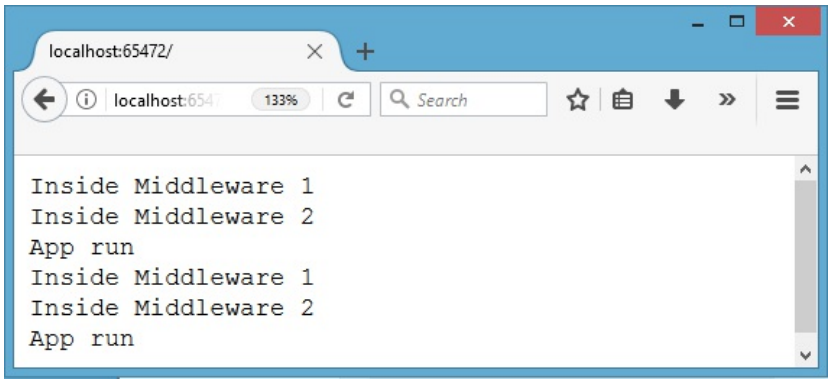
WOW! eBook
www.wowebook.org

Execution

We have two Middlewares and then `app.Run`. When we run this, we will see the following one:



There is a catch here. If you refresh the page, without building the code, the following is what you see next:



The reason is I have concatenated the string and when you refresh the page, it directly comes to the Middleware 1 followed by Middleware 2 and then `app.Run`. As the variable got initialised inside the `Configure`, it has an application scope. It will be appended henceforth. If you refresh the page again, you will see another set of the same messages getting appended to what you see in the preceding screenshot.

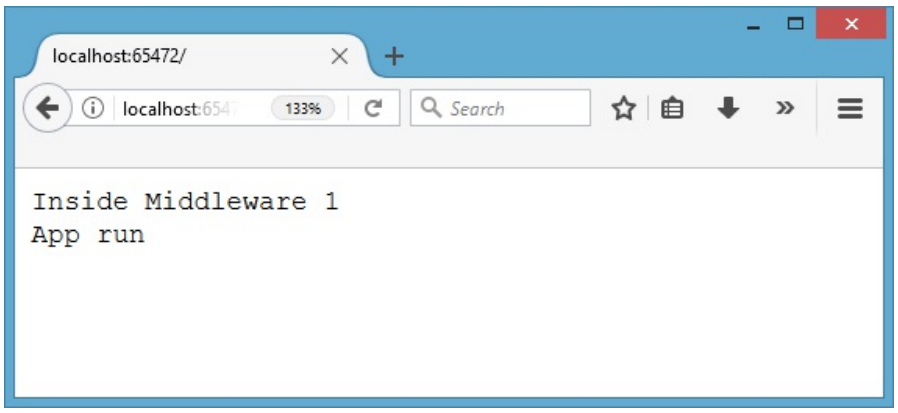
This is the way Middlewares hijack the application pipeline and execute before the response is delivered to the client. Let's consider another code block:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
   ILoggerFactory loggerFactory)
{
 var response = string.Empty;
```

WOW! eBook
www.wowebook.org

```
app.Use(async (context, next) =>
{
   response += "Inside Middleware 1\n";
   await next.Invoke();
});
app.Run(async context =>
{
   response += "App run\n";
   await context.Response.WriteAsync(response);
});
app.Use(async (context, next) =>
{
   response += "Inside Middleware 2\n";
   await next.Invoke();
});
}
```

What do you think the output will be? Check it out:



Middleware 2 got missed. That is because we wrote `app.Run` before Middleware 2. `app.Run` is the exit point which terminates the pipeline. Anything after that won't be considered by the runtime.

You can, definitely, code before and after invoke is called, as follows:

```
app.Use(async (context, next) =>
 {
 // Do something before the next middleware is
     invoked.
      await next.Invoke();
 // Do something after next middleware invocation.
});
```

WOW! eBook
www.wowebook.org

Ordering

There is a simple rule that the order in which they are registered is the order in which they will be executed. The ordering plays an important role while one component depends on another. The exception is something which may occur anytime, anywhere, within the app. So, we need to register it first before everything else, so that we can easily capture the exceptions.

WOW! eBook
www.wowebook.org

Summary

We explored what aspects are and how to determine Cross-Cutting Concerns in a typical programming scenario. There are specific locations where aspects can be tagged, which we illustrated next.

Aspect-oriented programming helps us to run custom logic before, after, or even replacing the method completely, where it is registered. We saw static as well as dynamic techniques to attach aspects with IL Weaving and Interceptors respectively. Interception is the technique which is more preferred over IL Weaving as it dynamically runs the registered block of code (otherwise called an aspect). On the other hand, IL Weaving deals with assembly update after the build process is over to insert aspect codes into the methods registered.

We then got into a thorough discussion on how Interception is adopted in ASP.NET Core in the form of filters and Middlewares.

Filters can intercept the path of action method execution, which can be either be attributed to the method as required, or can be registered with service collection to label it as a Global filter. One line of registration of a filter can make the filter intercept all action methods easily.

Middleware is a technique in ASP.NET Core which allows us to insert built-in/custom code chunks into the request-response pipeline.

Now that we have learned all the fundamental concepts of DI, in `Chapter 8`, *Patterns - Dependency Injection*, we are going to discover different techniques/patterns to inject dependencies into the clients.

WOW! eBook
www.wowebook.org

Patterns - Dependency Injection

In Chapter 7, *Interception*, we completed our journey in exploring the pillars of **Dependency Injection** (**DI**). Now, it's time to learn the principles behind Dependency Injection and explore how to apply different techniques to implement those principles to acquire a loosely coupled architecture. There are different techniques to implement DI, but if you are not choosing an appropriate one while you code, you will become a troublemaker for the project.

In this chapter, we will discuss different techniques, also known as **Patterns of Dependency Injection** with proper illustrations. I will shed light on use cases, and pros and the cons so that it will be easy for you to visualize the problems which you currently see in your application. That will eventually persuade you to adapt to these patterns in your app resulting in a better architecture.

Constructor Injection is the most important one among all these patterns. However, as developers, we should understand every available pattern. It's always recommended to use the best fit pattern for a particular scenario by analyzing the frequency and cases of using the dependency.

While most of these patterns address some specific context, always remember Constructor Injection is the simplest and can be opted without any confusion.

The following topics would be covered in this chapter:

- Dependency Inversion Principle
- Inversion of Control
- DI patterns
- The proper way to implement patterns
- The pros and cons of each pattern
- Adoption and examples in .NET Core 2.0

WOW! eBook
www.wowebook.org

Dependency Inversion Principle

The SOLID principles have a D in it, which is called the **Dependency Inversion Principle** (**DIP**). The following statement is by Robert C. Martin on DIP:

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

You can consider a High-level Module as a Business Module of one application comprising the complex logic of the application, and a Low-level Module as the classes that actually perform the basic or primary functions such as writing to disks, interaction with the database, and so on, with the commands received from the Business Module.

When objects of a High-level Module interact with objects of a Low-level Module, they generate a coupling. This happens because you have to refer to the classes of Low-level Modules in order to access them for instantiation. However, the DIP does not recommend this. The principle encourages us to reduce the coupling so that the modules live isolated. It also explains how this can be done through abstractions. Both should commit themselves to abstractions instead of directly depending on each other. Let's understand this point with an example.

Consider one class called `FeedbackService` that is saving one `Feedback` details in the database:

```
public class FeedbackService : IFeedbackService
{
  private INotifier notifier;
  public void SaveFeedback(Feedback feedback, NotifyType notify)
  {
    SaveFeedbackToDb(feedback);
    SendNotification(feedback, notify);
  }
  private void SendNotification(Feedback feedback, NotifyType notify)
  {
    if (notify == NotifyType.Email)
    {
        notifier = new EmailNotifier();
    }
    else if (notify == NotifyType.Sms)
    {
        notifier = new SmsNotifier();
    }
    else if (notify == NotifyType.Voice)
```

WOW! eBook
www.wowebook.org

```
        {
            notifier = new VoiceNotifier();
        }
        else
        {
            throw new ArgumentException("No matched notify type
              found.", notify.ToString());
        }
        notifier.SendNotification(feedback);
      }
      private int SaveFeedbackToDb(Feedback feedback)
      {
        // Save details in db.
        Console.WriteLine("Db Saving Started.");
        return 1;
      }
    }
```

The principal method here is `SaveFeedback`, which not only saves the feedback but also sends a notification. It takes the `Feedback` object and notifies type as parameters. When feedback is saved, usually one notification is sent to the customer and admin.

`FeedbackService` implements an interface which can be coded like the following:

```
public interface IFeedbackService
{
  void SaveFeedback(Feedback feedback, NotifyType notify);
}
```

`INotifier` is an interface which is implemented by all the notifier types. Have a look at the following code:

```
public interface INotifier
{
  void SendNotification(Feedback feedback);
}
public class EmailNotifier : INotifier
{
  public void SendNotification(Feedback feedback)
  {
    Console.WriteLine("Email Notification starts!");
  }
}
public class SmsNotifier : INotifier
{
  public void SendNotification(Feedback feedback)
  {
    Console.WriteLine("Sms Notification starts!");
  }
}
public class VoiceNotifier : INotifier
{
  public void SendNotification(Feedback feedback)
  {
    Console.WriteLine("Voice Notification starts!");
  }
```

WOW! eBook
www.wowebook.org

```
}
```

Note that I have not defined the body of the methods since they are here in the book for illustration. You can write logic and implement yours.

WOW! eBook
www.wowebook.org

Problems

The approach seems correct, but it is not recommended and does not respect software design principles which are described as follows:

Clearly, we are violating the Single Responsibility Principle with this approach. `FeedbackService` is taking the responsibility to save a feedback as well as sending notifications after success.

Tight coupling exists with the concrete classes `Feedback` and other `Notifier` classes such as `EmailNotifier`, `SmsNotifier`, and `VoiceNotifier`.

Our most popular, the `new` keyword is used inside the `SendNotification` of the `FeedbackService` class to decide which `Notifier` instance to create according to the input `notify` of type `enum`, which can be defined as follows:

```
public enum NotifyType
{
  Email = 1,
  Sms = 2,
  Voice = 3
}
```

WOW! eBook
www.wowebook.org

Solution

The solution should address all the issues with the preceding approach, so that the resulting architecture becomes loosely coupled. Let's get started.

The first thing is to separate the responsibilities from the `FeedbackService` class. This can be easily be done by transferring the `SendNotification` responsibility to the notifier instead of having it inside the class.

So, we can inject the `INotifier` type dependency inside the constructor. Thus, we can do `_notifier.SendNotification()` to send an injected type notification. Also, we changed the signature of `SaveFeedback`, which now accepts the `IFeedback` type parameter.

`FeedbackService` can be rewritten as follows:

```
public class FeedbackService : IFeedbackService
{
  private readonly INotifier _notifier;
  public FeedbackService(INotifier notifier)
  {
    _notifier = notifier;
  }

  public void SaveFeedback(IFeedback feedback)
  {
    SaveFeedbackToDb(feedback);

    _notifier.SendNotification(feedback);
  }
  private int SaveFeedbackToDb(IFeedback feedback)
  {
    throw new NotImplementedException();
  }
}
```

Now the `Service` does not depend upon the concrete implementations, and we eliminated the `if...else...if` clumsy codes from the class to decide which notification to send. The decision of notification type is now with the composer of the `Service` class instance. That can be done inside the Composition Root, something like the following:

```
var feedbackService = new FeedbackService(new SmsNotifier());
feedbackService.SaveFeedback(new Feedback());
```

WOW! eBook
www.wowebook.org

Inversion of Control (IoC)

Inversion of Control (**IoC**) is a Software Design Pattern which supports the **Dependency Inversion Principle** (**DIP**). There is a lot of confusion between these two terms, but if you closely observe, the names will clarify the doubts themselves.

The DIP is a principle or set of rules kind of theory. On the other hand, Inversion of Control is a pattern of well-defined steps we can follow to make our code DIP friendly. You have already experienced the steps of IoC, haven't you?

Notice the steps we followed to take away the control of object creations from the `FeedbackService` class to an external location, which can be considered as a Composition Root. We did the following to be precise:

1. Introduce interfaces for the concrete dependency: `INotifier` and `IFeedback`.
2. Update concrete dependencies to implement from interface: `EmailNotifier`, `SmsNotifier`, and `VoiceNotifier`.
3. Provide dependency to the constructor and store in a `private readonly` member:

```
public FeedbackService(INotifier notifier)
{
  _notifier = notifier;
}
```

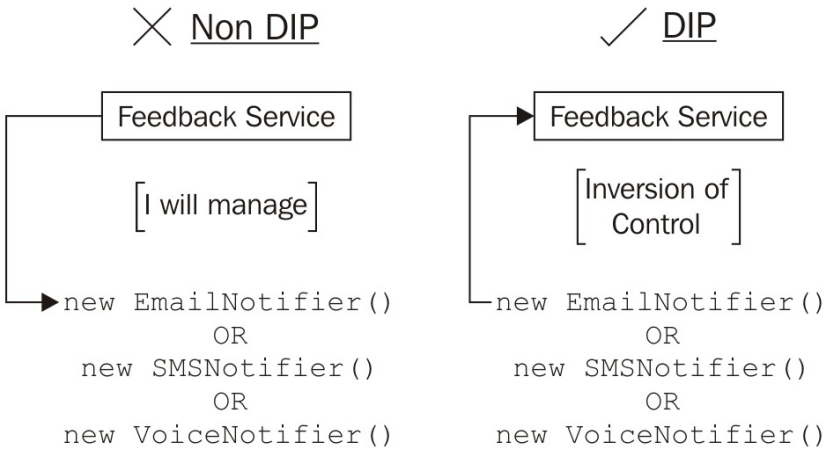
4. With the private member, we can call the method of the dependency class directly: `_notifier.SendNotification(feedback);`
5. Instantiate `Service` with dependency in the Composition Root: `var feedbackService = new FeedbackService(new SmsNotifier());.`

So, someone else is now trying to manage the object creation and lifetime. Before we executed those steps, the class itself was instantiating all its dependencies inside it using the `new` keyword. Therefore, we inverted the controls, and flow of management of objects is done in reverse through someone else, isn't it! That's it, this is what IoC is trying to express.

The last step involves injecting the dependency object into the constructor of `Service`, and the pattern is known as the **Constructor Injection Pattern**. Let's explore this pattern with other important ones in the next section.

WOW! eBook
www.wowebook.org

The following is the diagram representing the DIP and non-DIP implementations. Notice the arrows. In non-DIP, dependencies are managed by the class itself so, arrows pointing to concrete class objects, whereas, in DIP, an arrow is coming into the class, as the dependency objects are sent by someone else:



When a class manages all its dependencies itself, it is regarded as a ***Control Freak****, which is an anti-pattern. We will discuss more about this anti-pattern in the next chapter.*

WOW! eBook
www.wowebook.org

Patterns

The following are the four important patterns by which we can implement Dependency Injection:

- Constructor Injection Pattern
- Property Injection Pattern
- Method Injection Pattern
- Ambient Context

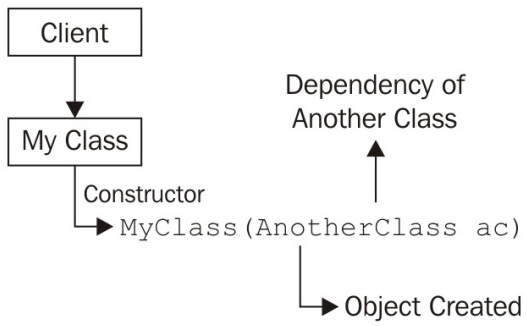
Let's explore one by one with illustrations, the advantages, and disadvantages.

WOW! eBook
www.wowebook.org

Constructor Injection pattern

As the name suggests, Injection will take the help of the constructor. In other words, the constructor is the target of injection. Injection is performed by supplying the dependency to the constructor as an argument. This is the most common technique to inject dependencies into classes.

The client calls the constructor to create the object and during the instantiation, the dependency gets injected, which is depicted as follows:



For example, a company, initially, was set up with only one department called Engineering. Let's design the `Employee` class for this company as follows:

```
public class Employee
{
    public int EmployeeId;
    public string EmployeeName;
    public Department EmployeeDept;

    public Employee(int id, string name)
    {
       EmployeeId = id;
       EmployeeName = name;
       EmployeeDept = new Engineering();
    }
}
```

We have a constructor which takes `id` and `name` as parameters and initialises an `Employee` object. As I told you, the company started only with Engineering as a department, so it's obvious that we assign `EmployeeDept` with a new object of the `Engineering` class. `Engineering` and `Department` classes would look something as follows:

```
public class Department
```

WOW! eBook
www.wowebook.org

```
{
  public int DeptId { get; set; }
  public string DeptName { get; set; }
}
public class Engineering : Department {  }
```

WOW! eBook
www.wowebook.org

The problem

Can you foresee a complex issue in this design? What if the company grows and builds another department like marketing? How would we incorporate this new department into the `Employee` class? How to say a particular employee belongs to the marketing department?

Unfortunately, there is no way to do this here, as the `Employee` class is tightly coupled with the `Engineering` class. This not only leads to a bad design, but developers also face hurdles for unit testing.

WOW! eBook
www.wowebook.org

The solution

So, what is the solution? We just need a `Department` object when we instantiate with `Employee`. How about providing the required type of `Department` (engineering/marketing) reference directly to the `Employee` class?

That's super easy. Let's have a look at the modification.

1. First, design an interface `IDepartment` which can be implemented by the `Department` class:

```
public interface IDepartment
{
  int DeptId { get; set; }
  string DeptName { get; set; }
}
public class Department : IDepartment
{
  public int DeptId { get; set; }
  public string DeptName { get; set; }
}
```

2. Instead of having a property of `Department` type, now we will have that of type `IDepartment`. Basically, we can now allow different types of departments with this technique. We will see this in action in a while. Also, see how this property is initialized inside the constructor using the param `dept` of type `IDepartment`.

```
public class Employee
{
  public int EmployeeId;
  public string EmployeeName;
  public IDepartment EmployeeDept;

  public Employee(int id, string name, IDepartment dept)
  {
    EmployeeId = id;
    EmployeeName = name;
    EmployeeDept = dept;
  }
}
```

3. Now have a look at the classes for different departments. They inherit from `Department`, thereby implementing `IDepartment`:

```
public class Engineering : Department
{
  public Engineering()
```

WOW! eBook
www.wowebook.org

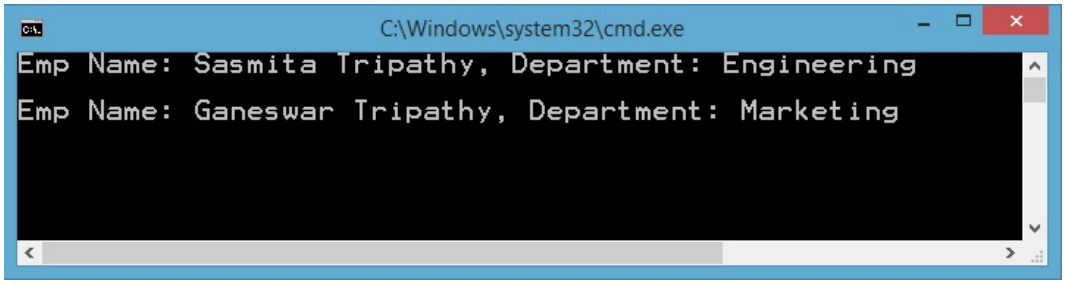
```
    {
      DeptName = "Engineering";
    }
  }
  public class Marketing: Department
  {
    public Marketing()
    {
       DeptName = "Marketing";
    }
  }
```

4. Creating employees with a different department is easy now. If we create such objects inside the `main` method, we can do the following:

```
static void Main(string[] args)
{
  var engineering = new Engineering();
  Employee emp = new Employee(1, "Sasmita Tripathy",
    new Engineering());
  var marketing = new Marketing();
  Employee emp1 = new Employee(2, "Ganeswar Tripathy",
    new Marketing());
  Console.WriteLine("Emp Name: " + emp.EmployeeName + ",
    Department: " + emp.EmployeeDept.DeptName);
  Console.WriteLine();
  Console.WriteLine("Emp Name: " + emp1.EmployeeName + ",
    Department: " + emp1.EmployeeDept.DeptName);
  Console.ReadLine();
}
```

You can see how a particular `Department` object is passed to the `Employee` constructor to create different employees who belong to different departments. That is the injection we did. We injected objects of type `IDepartment` (`Engineering` and `Marketing`) into the `Employee` constructor.

The following is the output:



WOW! eBook
www.wowebook.org

Curveball

You must be thinking everything in the preceding was perfect. But it is not. Let's analyze it:

```
var engineering = new Engineering();
Employee emp = new Employee(1, "Sasmita Tripathy",
  new Engineering());
emp.EmployeeDept = new Marketing();
```

You can see I created an object with employee department as `Engineering`. Then in the next line, we can easily change the department by assigning a `Marketing` object to it. This should be completely avoided.

To do that, the field must be marked as `private` and `readonly`.

```
private readonly IDepartment EmployeeDept;
```

It's not strictly recommended, but we should definitely follow this so that developers can't unnecessary hijack the actual values.

Secondly, the constructor does not handle the case where the dependent instance is received as `null`. The constructor can be altered as follows:

```
public Employee(int id, string name, IDepartment dept)
{
    EmployeeDept = dept ?? throw new ArgumentNullException();
    EmployeeId = id;
    EmployeeName = name;
}
```

Now here is the important thing, the constructor not only does the job of pushing the dependent instance to a `private` property, but also inserts values to other member variables such as `EmployeeId` and `EmployeeName`. This is against the Single Responsibility Principle. Therefore, we should have the constructor only handling the dependency thing and nothing else:

```
public Employee(IDepartment dept)
{
    EmployeeDept = dept ?? throw new ArgumentNullException();
}
```

Also, writing like this would guarantee that when an object is created, it will contain

WOW! eBook
www.wowebook.org

the dependency within it. At the same time, this will be valued only if the class is fully dependent on the dependency henceforth.

Let me summarise the points quickly as follows:

- The class which is dependent must expose a `public` constructor where the dependency can be injected
- If more than one dependency is required, then we can add more arguments to the constructor
- The dependent class should mark the member variable as `private readonly` where dependency instance will be assigned
- The dependent class constructor should only do the job of managing the dependency and nothing else

WOW! eBook
www.wowebook.org

Injecting with .NET Core 2.0

We saw every example previously, where we initialised using the `new` keyword. However, we should use the .NET Core 2.0 way of resolving the dependency:

```
    static void Main(string[] args)
    {
       var serviceProvider = new ServiceCollection()
       .AddTransient<IDepartment, Engineering>()
       .AddTransient<Employee>()
       .BuildServiceProvider();

       var emp = serviceProvider.GetService<Employee>();
       emp.EmployeeId = 1;
       emp.EmployeeName = "Sasmita Tripathy";

       var emp1 = serviceProvider.GetService<Employee>();
       emp1.EmployeeId = 2;
       emp1.EmployeeName = "Ganeswar Tripathy";

       Console.WriteLine("Emp Name: " + emp.EmployeeName + ",
          Department: " + emp.GetEmployeeDepartment());
       Console.WriteLine();
       Console.WriteLine("Emp Name: " + emp1.EmployeeName + ",
          Department: " + emp1.GetEmployeeDepartment());

        Console.ReadLine();
}
```

We use the `ServiceCollection` object to register the interfaces with the type we expect when it is resolved. When we do `serviceProvider.GetService();`, it returns an employee object with the department as `Engineering` since we registered `IDepartment` as `Engineering` in the collection.

WOW! eBook
www.wowebook.org

Implementation inside ASP.NET Core 2.0

Let's have a closer look at ASP.NET Core 2.0 internal codes for `ControllerActivatorProvider.cs` inside `Microsoft.AspNetCore.Mvc.Core Controllers` to understand how Constructor Injection is implemented.

WOW! eBook
www.wowebook.org

ControllerActivatorProvider.cs

I will show a screenshot of the class where the constructor is written:

```
/// <summary>
/// Provides methods to create an MVC controller.
/// </summary>
public class ControllerActivatorProvider : IControllerActivatorProvider
{
    private static readonly Action<ControllerContext, object> _dispose = Dispose;
    private readonly Func<ControllerContext, object> _controllerActivatorCreate;
    private readonly Action<ControllerContext, object> _controllerActivatorRelease;

    public ControllerActivatorProvider(IControllerActivator controllerActivator)
    {
        if (controllerActivator == null)
        {
            throw new ArgumentNullException(nameof(controllerActivator));
        }

        // Compat: Delegate to controllerActivator if it's not the default implementation.
        if (controllerActivator.GetType() != typeof(DefaultControllerActivator))
        {
            _controllerActivatorCreate = controllerActivator.Create;
            _controllerActivatorRelease = controllerActivator.Release;
        }
    }
```

`IControllerActivator` is the dependency which is injected into the constructor of this class. Then, it is using the instance to do some operation inside the class. We can easily find more of such kind of examples in .NET Core 2.0.

To see how this dependency is resolved, we can check the method `AddMvcCoreServices` of the `MvcCoreServiceCollectionExtensions.cs` class inside the `Microsoft.AspNetCore.Mvc.Core DependencyInjection`:

WOW! eBook
www.wowebook.org

```
//
// Controller Factory
//
// This has a cache, so it needs to be a singleton
services.TryAddSingleton<IControllerFactory, DefaultControllerFactory>();

// Will be cached by the DefaultControllerFactory
services.TryAddTransient<IControllerActivator, DefaultControllerActivator>();

services.TryAddSingleton<IControllerFactoryProvider, ControllerFactoryProvider>();
services.TryAddSingleton<IControllerActivatorProvider, ControllerActivatorProvider>();
```

WOW! eBook
www.wowebook.org

Importance of the Constructor Injection pattern

As the injection is tied to the constructor, this means whenever you try to instantiate a particular class, we are 100% sure that the dependency is shipped or tied with the object created. The guarantee of having the dependency is ensured by the Guard Clause, which verifies the dependency as not `null` before assigning that to the `private readonly` data member. Otherwise, the construction of class throws an exception.

There is no possibility of having one object of such class without injecting the dependency. Thus, the state of the object is always valid with no discrepancies. Unit testing with mocks becomes easy with this approach. Let's have a look at the pros and cons too.

WOW! eBook
www.wowebook.org

Advantages of the Constructor Injection pattern

This is the most preferable way of respecting the concept of DI. Most DI containers target the constructors. It is also the most common and more widely used pattern for DI. The injected component can be used anywhere in the class because when you create the object, the injected dependency comes into the class automatically and stays in some variable.

Developers prefer this technique when classes require some external dependency to initiate all other operations. As a constructor is involved, an object becomes ready with the dependent object once it is instantiated.

WOW! eBook
www.wowebook.org

Disadvantages of the Constructor Injection pattern

This is not so easy to implement when the class contains many constructors. Moreover, when the number of dependencies increases for a class, updating the constructor each time is not a feasible task.

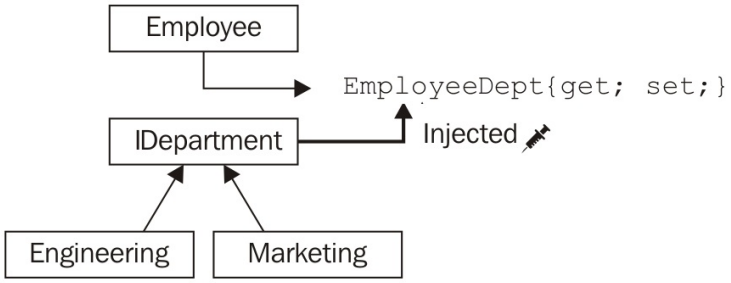
When you have a number of arguments for a constructor by adding the dependency ones with the existing ones, the class will look messy.

WOW! eBook
www.wowebook.org

Property Injection pattern

In this technique, instead of asking the constructor to support the injection, we will directly inject to the property of the class. Let's get started.

The following diagram depicts one property `EmployeeDept`, which can be injected directly with an `IDepartment` implementation:



The `Employee` class without the constructor argument for `IDepartment` and a new setter property will look something like the following:

```
public class Employee
{
  public int EmployeeId;
  public string EmployeeName;

  private IDepartment _employeeDept;
  public IDepartment EmployeeDept
  {
    set
    {
      this._employeeDept = value;
    }
  }

  public Employee(int id, string name)
  {
     EmployeeId = id;
     EmployeeName = name;
  }
}
```

We just removed the `IDepartment` argument from the constructor because we are going to directly assign an `IDepartment` type object to the property `EmployeeDept`. This can be done while creating the `Employee` object as follows:

```
static void Main(string[] args)
{
```

WOW! eBook
www.wowebook.org

```
    Employee emp = new Employee(1, "Sasmita Tripathy")
    {
      EmployeeDept = new Engineering()
    };

    Employee emp1 = new Employee(2, "Ganeswar Tripathy")
    {
       EmployeeDept = new Marketing()
    };
    Console.WriteLine("Emp Name: " + emp.EmployeeName + ",
       Department: " + emp.EmployeeDept.DeptName);
    Console.WriteLine();
    Console.WriteLine("Emp Name: " + emp1.EmployeeName + ",
       Department: " + emp1.EmployeeDept.DeptName);
    Console.ReadLine();
  }
```

Thus, by doing `emp.EmployeeDept = new Engineering();`, we are directly pushing the `Engineering` object into the property `EmployeeDept`. The same thing is done for `emp1` too.

However, this code won't compile. We are missing something important here. Consider the following screenshot:

```
Console.WriteLine("Emp Name: " + emp.EmployeeName + ", Department: " + emp.EmployeeDept.DeptName);
Console.WriteLine();
Console.WriteLine("Emp Name: " + emp1.EmployeeName + ", Department: " + emp1.EmployeeDept.DeptName);
```

```
IDepartment Employee.EmployeeDept { set; }
The property or indexer 'Employee.EmployeeDept' cannot be used in this context because it lacks the get accessor
```

The error is totally self-explanatory. We forgot to include the Getter block as we wanted to print the department name. Consider the following code snippet:

```
  private IDepartment _employeeDept;
  public IDepartment EmployeeDept
  {
    get {
          return this._employeeDept;
    }
    set
    {
          this._employeeDept = value;
    }
  }
```

Now, we are good. When you run it, you will see the same output as we got in the case of Constructor Injection.

Property Injection is preferred when there is an optional dependency required for the class. What that means is, if you don't supply the value, the class won't mind as it can run without this dependency. If the developer forgets to assign the dependency, it

WOW! eBook
www.wowebook.org

should not affect the flow of the app.

For instance, in our example, if you don't do anything with the property `EmployeeDept`, the class will behave as usual as long as you are not reading the property. In the next section, I will tell you how to beautify the code because the code is not yet perfect.

WOW! eBook
www.wowebook.org

Curveball

The problems with the approach which we have taken can be any one of the following:

- If you don't assign anything to the property (as we developers forget sometimes being lazy), the output won't return you anything when you print.
- You can assign null to the property as simply as `emp.EmployeeDept = null;`.
- Once you assign a dependency to a property, you can easily assign another dependency easily, which might be a problem. This might happen accidentally, however, it will create side effects for the whole system at some point in time. Imagine a situation where you initialized the property with engineering and then changed that to marketing, which is wrong.

To handle all these situations, we should introduce more protection to the property. Let's do this:

```
private IDepartment _employeeDept;
public IDepartment EmployeeDept
{
  get
  {
      if (this._employeeDept == null)
      this.EmployeeDept = new Engineering();
         return this._employeeDept;
  }
  set
  {
    if (value == null)
       throw new ArgumentNullException("value");
    if (this._employeeDept != null)
       throw new InvalidOperationException();

     this._employeeDept = value;
   }
 }
```

Inside the setter, we check for `null` and throw an exception. Then, we again investigate whether it is already there and throw an `InvalidOperationException`. Likewise, in the Getter, we are assigning a default value as engineering when passed `null`. Thus, we guarded our property for all such difficult situations, as previously discussed.

Property Injection is not supported by the .NET Core built-in DI or IoC

WOW! eBook
www.wowebook.org

Container out of the box. Also, there is no such plan to bring this into the default container as of now. You have to use an external container such as Autofac to support this feature.

WOW! eBook
www.wowebook.org

Advantages of the Property Injection Pattern

We can see the following advantages of having a Setter Injection Pattern:

- It does not require adding a new constructor or modifying an existing constructor.
- This pattern comes in handy if the class has multiple dependencies of different types. Suppose there is a class which will use different types of logging such as logging, to text or logging to the database at the same time in different conditions, then having two setter properties where we can simply inject the required logger to do our task.

WOW! eBook
www.wowebook.org

Disadvantages of the Property Injection Pattern

Setters are not easy to identify in a particular design to inject some external dependency. Still, they are preferred in some situations. However, we should be careful while using the property which is injected because that might be `null`. There should be an extra check for `null`, everywhere where you want to use it. As we already discussed, protection must be provided to the property to avoid the inconsistent behavior.

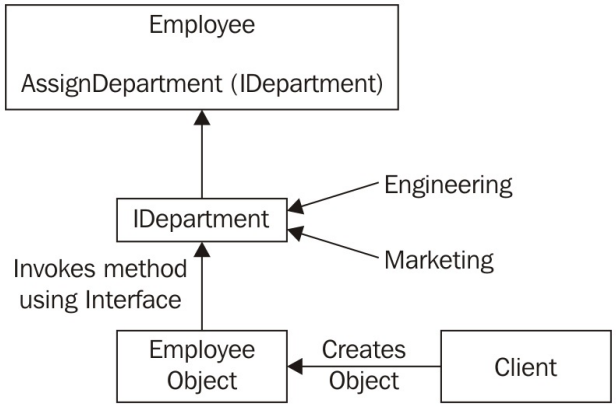
Property Injection is considered as bad practice in most of the cases because it hides the dependencies. When the class is instantiated, there is no guarantee that the dependency object will be available for the class to do any operation, unlike Constructor Injection, where the dependency makes its way through during instantiation itself.

WOW! eBook
www.wowebook.org

Method Injection Pattern

As the name suggests, we will inject the dependency into a method like we did in the case of the constructor. This comes in handy when we want different types of dependency in different methods of a class.

Look at the following diagram, which tells us about the `AssignDepartment(IDepartment)` method taking `IDepartment` as a dependency inside the `Employee` class:



>

The client can create an object of the `Employee` class and then fire `AssignDepartment` with the appropriate implementation of `IDepartment`, so that dependency is provided.

Let's see how the `Employee` class can be coded:

```
public class Employee
{
    public int EmployeeId;
    public string EmployeeName;
    public IDepartment EmployeeDept;

    // Default Constructor added for .NET Core 2.0 DI.
    // So that it can automatically create the instance.
    public Employee() { }

    public Employee(int id, string name)
    {
      EmployeeId = id;
      EmployeeName = name;
    }

    public void AssignDepartment(IDepartment dept)
    {
       EmployeeDept = dept;
       // Other business logic if required.
```

WOW! eBook
www.wowebook.org

```
        }
    }
```

In this particular scenario, we have a method `AssignDepartment`, which takes one `IDepartment` type argument and then assigns that to the property. An `Employee` object can be created accordingly. Consider the following code snippet:

```
Employee emp = new Employee(1, "Sasmita Tripathy");
emp.AssignDepartment(new Engineering());

Employee emp1 = new Employee(2, "Ganeswar Tripathy");
emp1.AssignDepartment(new Marketing());
```

It also produces the same output as what we have already seen.

This pattern is useful when we require some dependency, only for some particular operation in a method and not for the whole class. Sometimes, the situation demands a small change which might be an edge case. To accommodate a small change, the simplest way would be to create a method and do whatever you want inside that (using dependencies passed in as arguments) instead of touching the constructors or properties.

WOW! eBook
www.wowebook.org

Curveball

In the case of Method Injection, by convention, we should also take care of null check to ensure that the dependency is available before using it:

```
public void AssignDepartment(IDepartment dept)
{
  EmployeeDept = dept ?? throw new ArgumentNullException("value");
  // Other business logic if required.
}
```

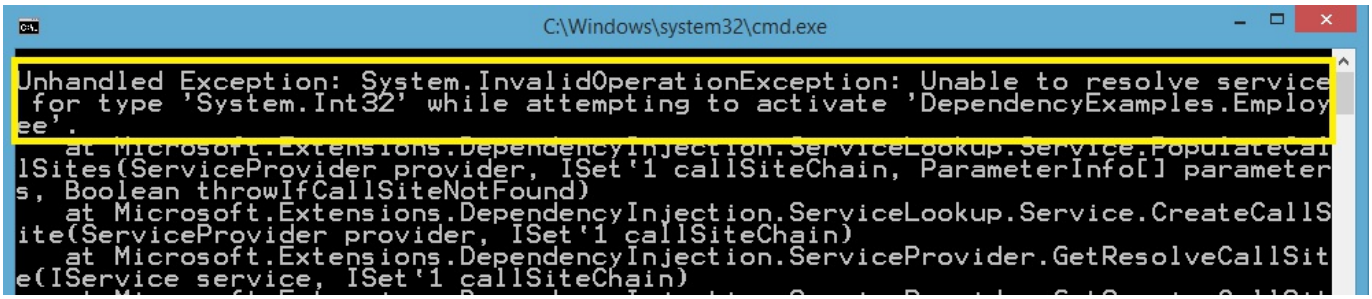
WOW! eBook
www.wowebook.org

Injecting with .NET Core 2.0

As we did in previous patterns, we will get the instance using the `GetService` method and then call the required method by injecting the dependency:

```
Employee emp = serviceProvider.GetService<Employee>();
emp.EmployeeId = 1;
emp.EmployeeName = "Sasmita Tripathy";
emp.AssignDepartment(serviceProvider.GetService<IDepartment>());

Employee emp1 = serviceProvider.GetService<Employee>();
emp1.AssignDepartment(serviceProvider.GetService<IDepartment>());
emp1.EmployeeId = 2;
emp1.EmployeeName = "Ganeswar Tripathy";
```

The next thing you will see is the following as the output:



This happened because we have a constructor which is taking `integer` and `string` params. The provider is not aware of the process to resolve them, therefore, it throws an exception. The solution is to provide a default constructor, as the service provider is looking for it.

```
public Employee() {  }
```

WOW! eBook
www.wowebook.org

Implementation inside .NET Core 2.0

We can get many instances of Method Injection Pattern usage inside the .NET Core 2.0. One of them is inside the simplest and frequently used class `MvcServiceCollectionExtensions.cs`.

WOW! eBook
www.wowebook.org

MvcServiceCollectionExtensions.cs

The `MvcServiceCollectionExtensions.cs` class contains the method `AddMvcCore`, which is an example of Method Injection. The following screenshot shows the method at a glance:

```
/// <summary>
/// Adds essential MVC services to the specified <see cref="IServiceCollection" />.
/// </summary>
/// <param name="services">The <see cref="IServiceCollection" /> to add services to.</param>
/// <returns>An <see cref="IMvcCoreBuilder"/> that can be used to further configure the MVC
public static IMvcCoreBuilder AddMvcCore(this IServiceCollection services)
{
    if (services == null)
    {
        throw new ArgumentNullException(nameof(services));
    }

    var partManager = GetApplicationPartManager(services);
    services.TryAddSingleton(partManager);

    ConfigureDefaultFeatureProviders(partManager);
    ConfigureDefaultServices(services);
    AddMvcCoreServices(services);

    var builder = new MvcCoreBuilder(services, partManager);

    return builder;
}
```

`IServiceCollection` is injected into the `AddMvcCore` method. After it has been verified as not `null`, the service collection is processed further. I am only showing a screenshot of the relevant codes to understand the concept. You can definitely find such examples in other classes.

*The **Interface Injection Pattern** is something which we have already discussed in other patterns, where we injected the implementations such as `IDepartment` which is an interface, into the constructor, property or a method. It's recommended to inject the implementation instead of the concrete class to avoid tight coupling. Interface injection allows us to achieve loose coupling along with abstraction.*

WOW! eBook
www.wowebook.org

Ambient context

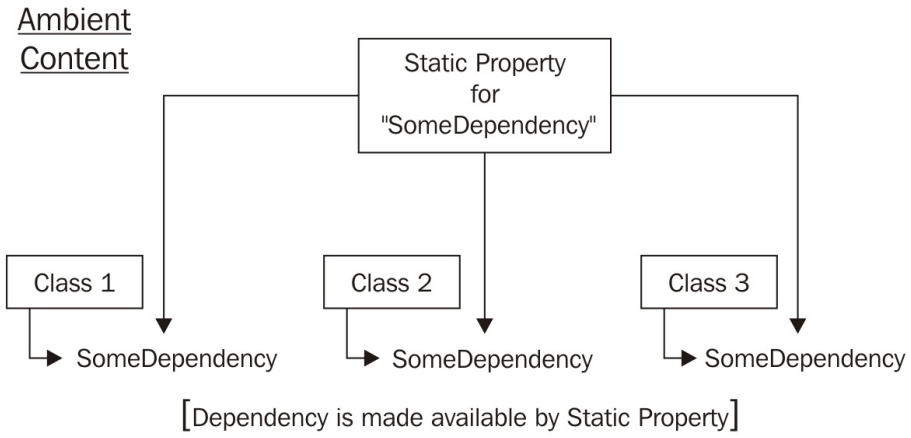
Ambient is an adjective meaning completely surrounding or encompassing. This means when we say Ambient context, it denotes some context, which would pertain its existence and behavior in the background or surrounding.

When many classes require the same dependency, we should follow some technique to make it available to every such client.

The patterns we discussed are not appropriate for such a scenario. If you try to make this possible using Constructor Injection, for example, you would end up adding a constructor for every class, having the same dependency injected to it.

Sounds more like a Cross-cutting Concern, doesn't it? To achieve this, we can simply add a static accessor for the dependency. This will make sure that the dependency is reachable to all the clients demanding it.

The concept can be visualized in the following diagram:



Let's consider an example to get a better understanding of the implementation. An abstract class `DepartmentProvider` can be designed to provide a static accessor called `Current`:

```
abstract class DepartmentProvider
{
  private static DepartmentProvider current;
  public static DepartmentProvider Current
  {
    get
```

WOW! eBook
www.wowebook.org

```
        {
           return current;
        }
        set
        {
           current = value;
        }
      }
      public virtual Department Department { get; }
    }
```

We marked the class as `abstract` and there is a `virtual` property called `Department`, which will be accessed by any class requiring that.

Let's see how we can use the `Current` property. The following code uses the class `MarketingProvider` instance which is a derived class of `DepartmentProvider`:

```
    static void Main(string[] args)
    {
      var serviceProvider = new ServiceCollection()
      .AddTransient<IDepartment, Engineering>()
      .AddTransient<Employee>()
      .AddTransient<MarketingProvider>()
      .BuildServiceProvider();

       // Set the Current value by resolving with
          MarketingProvider.
       DepartmentProvider.Current = serviceProvider.GetService
          <MarketingProvider>();

       Employee emp = serviceProvider.GetService<Employee>();
       emp.EmployeeId = 1;
       emp.EmployeeName = "Sasmita Tripathy";
       emp.EmployeeDept = DepartmentProvider.Current.Department;

       Employee emp1 = serviceProvider.GetService<Employee>();
       emp1.EmployeeId = 2;
       emp1.EmployeeName = "Ganeswar Tripathy";
       emp1.EmployeeDept = DepartmentProvider.Current.Department;

       Console.WriteLine("Emp Name: " + emp.EmployeeName + ",
         Department: " + emp.EmployeeDept.DeptName); // Marketing
       Console.WriteLine();
       Console.WriteLine("Emp Name: " + emp1.EmployeeName + ",
         Department: " + emp1.EmployeeDept.DeptName); // Marketing
       Console.ReadLine();
    }
```

See how we can register `MarketingProvider` using the built-in DI container. Then we assigned that to the `Current` property by which we are able to get the value as marketing when we read `DepartmentProvider.Current.Department`. `MarketingProvider` is a child class of `DepartmentProvider`, which returns a `Marketing` object. Refer to the following code:

WOW! eBook
www.wowebook.org

```
class MarketingProvider : DepartmentProvider
{
  public override Department Department
  {
     get { return new Marketing(); }
  }
}
```

So, we made a conclusion here. We can use the `static` property wherever we need inside the app and the value will be the same, unless you set it differently in-between.

WOW! eBook
www.wowebook.org

Curveball

A following are a few important points to remember while implementing an Ambient Context:

- This should be used only if it is really necessary. Constructor Injection or Property Injection should be the first choice before deciding to use Ambient Context. If you can't decide, just choose other DI patterns.
- The Service Locator anti-pattern is quite similar to this pattern in the way it provides the dependency. However, there is a difference and that is Ambient Context supplies a single dependency, whereas Service Locator is responsible for providing all requested dependencies. We will discuss more on Service Locator in `Chapter 9`, *Anti-Patterns and Misconceptions on Dependency Injection*.
- If you don't correctly implement Ambient Context, it will have side effects. Suppose you initiate using one context or provider and during the process, you change it due to some reason. Now, when you read the value, it will provide you with the changed value instead of the first one because of static. That means you need to be very careful while implementing this.
- With the implementation we discussed, you can easily set `null` to the `Current` property `DepartmentProvider.Current = null;`. Therefore, we have to protect it by using a guard clause.

The following code represents a simple guard clause inside getter and setter blocks:

```
abstract class DepartmentProvider
{
  private static DepartmentProvider current;
  public static DepartmentProvider Current
  {
    get
    {
      if (current == null)
      current = new DefaultDepartmentProvider();
      return current;
    }
    set
    {
       current = value ?? new DefaultDepartmentProvider();
    }
  }
  public virtual Department Department { get; }
}
```

WOW! eBook
www.wowebook.org

We did a null check both in getter and setter and used a backup provider called `DefaultDepartmentProvider` to overcome the situation of mishandling the context.

WOW! eBook
www.wowebook.org

Implementation inside .NET Core 2.0

The `corefx` library has a `CurrentPrincipal static` property inside the partial class `Thread` under `System.Threading` namespace.

```
public static IPrincipal CurrentPrincipal
{
  get
  {
     return CurrentThread._principal;
  }
  set
  {
     CurrentThread._principal = value;
  }
}
```

The usage can be seen in a method `Demand()` of the class `PrincipalPermission` under the namespace `System.Security.Permissions`:

```
public void Demand()
{
    IPrincipal principal = Thread.CurrentPrincipal;
    if (principal == null)
        throw new SecurityException(SR.Security_PrincipalPermission);
    if (_idArray == null)
        return;
```

WOW! eBook
www.wowebook.org

The advantages of the Ambient Context

Ambient Context is definitely a savior when an app demands the same dependency multiple times during the app execution. It's very hard to realise the fact that while we write codes for injecting dependencies, there are many such instances of asking the same dependency repeatedly within the app. This is the time where we should take the help of the Ambient Context Pattern, and with just a simple `static` accessor, we will be able to get the dependency. This not only reduces code but also imposes the DRY principle.

WOW! eBook
www.wowebook.org

The disadvantages of the Ambient Context

Ambient Context is not easy to implement and needs careful attention while using it. By just looking at a class, we can't tell whether it implements the Ambient Context Pattern or not. If the context changes in between execution, the outcome will be different, resulting in side effects.

WOW! eBook
www.wowebook.org

Summary

While implementing DI, we should definitely learn the techniques associated with it. These techniques, otherwise called DI patterns, play a vital role in the application architecture. So, deciding which pattern to use when is the most important factor in the case of injecting dependencies.

Constructor Injection is the most widely used, is easy, and should always be your first choice. However, there are scenarios where you might opt for another pattern.

When you see there are multiple dependencies required for different operations of a class, choose the Method Injection Pattern as it allows you the flexibility to inject dependency as you require.

Another beautiful design case reveals itself while you land on a situation where you need certain dependency at many places in your app. It looks like a Cross-Cutting Concern, however, you expect a certain return type, which is the dependency. For that, you need to use Ambient Context and can't rely on interceptors. This is because you need the returned dependency to do some operation in your class.

Property Injection is something which allows you to inject optional dependency. This means it relies upon a Local Default for sure, otherwise, we might end up in an exception while requesting it.

In `Chapter 9`, *Anti-Patterns and Misconceptions on Dependency Injection*, we will explore bad practices while implementing DI, which are denoted as anti-patterns.

WOW! eBook
www.wowebook.org

Anti-Patterns and Misconceptions on Dependency Injection

Patterns show us the proper way to implement Dependency Injection. In the last chapter, we got an insight into every pattern available in the DI Ecosystem and explored their usage, their pros, their cons, and which one to select when. If these patterns are followed correctly, we achieve a loosely coupled architecture which will be easier to unit test.

However, while applying these patterns, we generally neglect some principles, which leads us to problems in the future. That might happen either to the ignorance about pattern' behavior or simply due to laziness.

At that point, a pattern becomes an anti-pattern, as it does not solve the problem; rather it gives birth to more bugs and maintenance becomes a headache.

After reading this chapter, don't worry if you find anti-patterns in your project because we will learn the steps to refactor each anti-pattern to a proper Pattern by selecting any technique from the patterns we discussed in the last chapter. If you directly came to this chapter, I suggest you complete the previous chapter (which covers everything you should know about DI Patterns in detail) first before moving ahead.

In this chapter, we will talk about such scenarios in projects with examples to understand how a pattern can behave as an anti-pattern.

We will cover the following topics in this chapter:

- When DI becomes an anti-pattern
- DI anti-patterns
- Identifying anti-patterns in the project
- Problems anti-patterns introduce to apps
- Solutions and refactoring steps for each type of anti-pattern

WOW! eBook
www.wowebook.org

When does Dependency Injection become an anti-pattern?

We are heading towards exploring anti-patterns followed by developers in projects with DI in place. However, have you envisaged the scenario where DI itself becomes an anti-pattern! Yes, it can be an anti-pattern sometimes. The cases are listed as follows:

Suppose we have a controller that is depending on a `Service` for some database operation. For instance, `UsersController` would need an `IUsersService` type as a dependency for `User` table-related operations. We configured `UsersService` for the dependency and it got registered with the container. Now, help yourself a little bit and ask the following question.

Are you going to change the dependency to any other implementation of `IUsersService`*?* If your answer is *No*, then hold back and think a bit. The reason is, if you won't change the implementation for the dependency either dynamically through code or configurations, then DI is not going to play an important role. For example, if you save data to a database/XML/text file, you might have different implementations and swapping is needed in certain conditions, then DI will definitely come in handy. But if you only save to the database and nothing else, there is no necessity to have extra codes for injecting dependencies. It does not make much sense.

Are you going to need a dependency as soon as you start the app? DI recommends that we register everything at one place called the Composition Root. However, imagine a `Service` called `CompanyService` whose instance is needed only when I want to add a `User` to a company. For example, see the following code:

```
public IActionResult AddUser(UserModel userModel)
{
  var user = _usersService.CreateUser(userModel);
  if (userModel.AddUserToCompany)
  {
    var companyService = new CompanyService();
    companyService.AssignUserToCompany(user);
  }
  return View();
}
```

WOW! eBook
www.wowebook.org

Normally we have admins in a company who manages user records. Imagine a scenario, where the admin is logged in and wants to assign a few users to a certain company. In this case, there is a `boolean` coming from the model named `AddUserToCompany`. If that is `true`, we need to assign the user to the company. This means a lazy instantiation and it is looking pretty good.

So, here, it's meant to get a `CompanyService` instance quickly and work with it. However, if you have chosen DI, then the `CompanyService` instance stays in the container (because you register it at the Composition Root) till you actually use it inside the code, which often does not happen. Sometimes, it doesn't happen at all if the user logged in as a normal user and not an admin.

This is just a small example. You can visualize a complex application with these types of scenarios that might create a nasty impact with DI in place. Sometimes it's better to have lazy instantiation instead of having them in a container which consumes memory space unnecessarily.

Like a pattern, if DI is selected unnecessarily or used incorrectly, that leads to an anti-pattern itself. If you are never going to inject different dependencies for one implementation, then no need to use DI. Let's now explore anti-patterns, which we might encounter while using DI in the project.

WOW! eBook
www.wowebook.org

Anti-patterns

There are four major types of anti-pattern found in projects where DI is implemented and those are listed as follows:

- Control Freak
- Bastard Injection
- Constrained Construction
- Service Locator

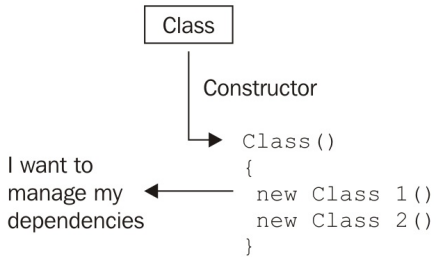
Let's analyze each type and see how to avoid them.

WOW! eBook
www.wowebook.org

Control Freak

Control Freak is opposite to Inversion of Control. When a class holds its dependencies and tries to manage them itself without any interference from others, it will be tagged as a **Control Freak**.

The following diagram shows Control Freak at a glance:



When we said that *it tries to manage the dependencies itself*, what do we mean by that? Remember in `Chapter 6`, *Object Lifetime*, if a class wants to manage a dependency, that means it wants to instantiate it, then manage its lifetime and kill or dispose of it as well.

Instantiation, yes, that is done by our well-known keyword `new`. The class instantiates every dependency with the `new` keyword inside it, then works with the objects and disposes of them through a `destructor` or `finalizer`. As a result, it is creating a tightly coupled system as soon as it does so. Moreover, it makes unit testing difficult.

The class either creates the object itself or asks some other class to do so on its behalf. Let's understand the scenario with an example:

```
public class EmployeeService
{
  private readonly EmployeeRepository repository;
  public EmployeeService()
  {
     string connectionString = "Read String from config";
     this.repository = new SqlEmployeeRepository(connectionString);
  }
}
```

The first thing that instantly comes into play is `new SqlEmplyeeRepository()`. Now, this

WOW! eBook
www.wowebook.org

service is tightly coupled with `SqlEmployeeRepository`. Suppose you want to use another repository class instead of this, then we have to change the service code and recompile. There is no such plugin point to say *I am providing you with this repository, please use it*.

WOW! eBook
www.wowebook.org

Problem

To fix this problem, the developer might think of different patterns, which eventually complicates the scene. We can see the following types of factory used commonly:

- Concrete Factory
- Abstract Factory
- Static Factory

WOW! eBook
www.wowebook.org

Concrete Factory

Creating `EmployeeRepositoryFactory` is another excuse (and a lazy approach) to have a `Create()` method that will create a `SqlEmployeeRepository` instance by the `new` keyword:

```
public class EmployeeRepositoryFactory
{
  public EmployeeRepository Create()
  {
    string connectionString = "Read String from config";
    return new SqlEmployeeRepository(connectionString);
  }
}
```

We removed this block from `EmployeeService` but added another class which is working very similar to the previous one. We can then use the factory as follows:

```
public EmployeeService()
{
  var employeeRepofactory = new EmployeeRepositoryFactory();
  this.repository = employeeRepofactory.Create();
}
```

Inside the `EmployeeService` constructor, we use the `new` keyword to get a factory instance and then call `Create()` to get the `SqlEmployeeRepository` instance and assign it to the `repository` variable.

Did we achieve anything useful? Not at all. We just added another bunch of code to the `Service`, doing the same thing (using the `new` keyword to instantiate) in an indirect way through the factory. Basically, the factory uses the same `new` keyword to instantiate the `SqlEmployeeRepository` class. That is what we wanted to avoid, but we could not achieve this.

WOW! eBook
www.wowebook.org

Abstract Factory

Abstract Factory behaves as an encapsulating component that encapsulates complex logic including dependencies associated with it. It can transfer the control from the consumer because it does not totally allow the consumer to control the lifetime of objects.

A Derived Factory is responsible for the creation and management of the required repository instead of the initially designed factory. Consider the following code snippet:

```
public abstract class EmployeeRepositoryFactory
{
    public abstract EmployeeRepository Create();
}
```

That means we are trying to hide which repository will be served. We are trying to achieve loose coupling by hiding the actual supplied type.

In order to assign the `repository` variable, we have to inherit this class and create one child of it returning `SqlEmployeeRepository`:

```
public class SqlEmployeeService : EmployeeRepositoryFactory
{
  public override EmployeeRepository Create()
  {
    string connectionString = "Read String from config";
    return new SqlEmployeeRepository(connectionString);
  }
}
```

Basically, we decoupled the instantiation of the `repository` from the main service. The same question again arises. Did we achieve anything useful? I don't think so. That is because this new architecture again achieved at the cost of one instantiation inside the `EmployeeService` (with the `new` keyword):

```
public EmployeeService()
{
  var sqlEmployeeService = new SqlEmployeeService();
  this.repository = sqlEmployeeService.Create();
}
```

Although you were able to hide the `SqlEmplyeeRepository` from the factory by making

WOW! eBook
www.wowebook.org

it abstract, you made no change to the way things are handled inside the `EmployeeService` constructor. You are now instantiating `SqlEmployeeService`.

WOW! eBook
www.wowebook.org

Static Factory

The next approach is to avoid factory instantiation by introducing the `static` pattern. Consider the following code snippet:

```
public static class EmployeeRepositoryFactory
{
  public static EmployeeRepository Create()
  {
    string connectionstring = "read string from config";
    return new SqlEmployeeRepository(connectionstring);
  }
}
```

This would prevent us from creating an object and using it directly. Let's see how:

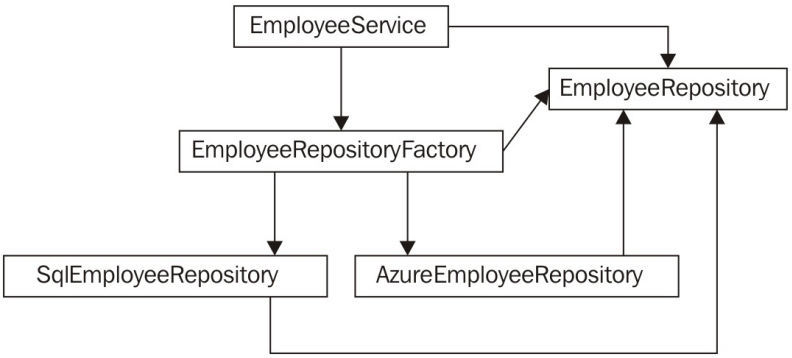
```
public EmployeeService()
{
   repository = EmployeeRepositoryFactory.Create();
}
```

Horray! We finally removed the `new` keyword. Alright, looks like we are done. Oh, wait! We still have `SqlEmployeeRepository` instantiated using `new` inside `Create()`. But there is a simple solution to reading this type of repository from the `config` or something:

```
public static EmployeeRepository Create()
{
  var repository = "read from config";
  switch (repository)
  {
    case "sql":
        return EmployeeRepositoryFactory.CreateSql();
    case "azure":
        return EmployeeRepositoryFactory.CreateAzure();
    default:
        throw new InvalidOperationException("Invalid operation");
  }
}
```

It looks promising at first glance, but it is actually something else. All classes become tightly coupled. This is depicted in the following diagram:

WOW! eBook
www.wowebook.org



`EmployeeService` depends upon the `EmployeeRepositoryFactory` for the `EmployeeRepository` instance, which means that clients of the service need to refer to the factory, repository, and then the types of repository such as `SqlEmployeeRepository` and `AzureEmployeeRepository` returned by `CreateSql()` and `CreateAzure()`.

These concrete classes got coupled, which can't produce a flexible design resulting in bad days for upcoming programmers of the application.

WOW! eBook
www.wowebook.org

Solution

We have already explored patterns in the last chapter. The best fit for Control Freak issues is Constructor Injection. Consider the following code snippet:

```
public class EmployeeService
{
  private readonly IEmployeeRepository repository;
  public EmployeeService(IEmployeeRepository repository)
  {
    this.repository = repository ?? throw new
      ArgumentNullException("repository");
  }
}
```

In this way, you abstracted out the concrete repository and also inserted the dependency through the constructor. More refactoring can be applied by introducing a factory class, which will be responsible for generating the repository.

Now using the DI Containers, you can register either the factory interface or the repository interface and resolve as required so that the dependency will be available for the service.

Control Freak is the most commonly found anti-pattern implemented in projects. While developers consider using DI in their project, they find it difficult sometimes and they are attracted to controlling the object creation instead of any other component doing it for them. The outcome will be awesome if they just ignore being a control freak and go with the DI flow.

The next anti-pattern is Bastard Injection. However, before going to that, we need to understand a manual approach to dependency management called **Poor Man's DI**.

WOW! eBook
www.wowebook.org

Poor Man's DI

Poor Man's DI; the name sounds very interesting, doesn't it! When you yourself try to take care of the registrations of dependencies at commonplace (which otherwise can be denoted as the Composition Root) instead of using a library (specifically a DI Container), that technique can be defined as Poor Man's DI.

WOW! eBook
www.wowebook.org

The approach

Let's find out how this can be done with a quick code example.
Suppose `EmployeeService` has a dependency of the `EmployeeRepository` type, which we can directly provide into the constructor, shown as follows:

```
static void Main(string[] args)
{
  EmployeeService empService = new
     EmployeeService(new EmployeeRepository());
  Console.ReadKey();
}
```

Consider this console app example, where `EmployeeService` is instantiated inside the `Main` method. It looks simple and beautiful. But it is not that effective if the dependency nesting runs deep.

What if `EmployeeRepository` again needs some other dependency and then another one and so on. You might end up doing something like the following:

```
EmployeeService empService = new
  EmployeeService(new EmployeeRepository(new
  Cass1(new Class2(new Class3()))));
```

Now it becomes ugly and not so maintainable. Then you might think of simplifying this structure to some extent by introducing a default constructor for each class. So, the following is what you will do for `Service` and `Repository`:

```
public class EmployeeService : IEmployeeService
{
  private readonly IEmployeeRepository repository;
  // Default Constructor calls the parameterized one
  public EmployeeService() : this(new EmployeeRepository())
  {
  }
  public EmployeeService(IEmployeeRepository employeeRepository)
  {
     repository = employeeRepository;
  }
}
public class EmployeeRepository : IEmployeeRepository
{
  private readonly ISomeClass class1;
  // Default constructor calls the parameterised one.
  public EmployeeRepository() : this(new Class1())
  {
  }
  public EmployeeRepository(ISomeClass someClass)
```

WOW! eBook
www.wowebook.org

```
    {
      class1 = someClass;
    }
  }
```

You can do the same thing to all the nested classes. We added default constructors for all the classes; these a call internally to the parameterized constructors with a default dependency instance. It will definitely, reduce the code. Check the following reduced code:

```
static void Main(string[] args)
{
  EmployeeService empService = new
    EmployeeService(); // No Dependency passed here.
  Console.ReadKey();
}
```

Now there is no need to pass any dependency to these constructors. Moreover, if you pass any dependency of the expected type, that will also work because a parameterized constructor also exists. That means we got a very flexible structure for the class and we also reduced the instantiation.

WOW! eBook
www.wowebook.org

Problem

While we tried to make the instantiation flexible, testable, and simple, we did not realize the following disadvantages with this approach:

- A concrete reference was created in the default constructor when we did `new EmployeeRepository()`.
- The default dependency instance is tied with all the classes using the `new` operator. But the whole point of the dependency injection technique is to reduce the `new` keywords in the app.
- We also violated another principle in registering all dependencies in one place known as the Composition Root. Now, we have Composition Roots everywhere across the app in all classes. This is bad.
- Imagine the situation of having 10 classes using the same dependency; now it would be very problematic to instantiate everything with the same `new` keyword again and again.
- You don't have any facility to manage the lifetime of the dependencies created like this using the `new` keyword. You have to manually dispose of everything which can become a headache in a fully fledged application. Moreover, if you want to reuse one instance, you need to carefully deal with that. This might lead to inconsistent behavior and wrong data.

So, how do we deal with this? Let's find out.

WOW! eBook
www.wowebook.org

Solution

Obviously, using a DI Container to register the dependencies solves the problem. We can remove the default constructor that is actually creating a tight coupling with the required types:

- The registration of dependencies can be restricted to one place unlike Poor Man's DI, where you instantiate concrete classes here and there.
- With the registrations inside the Composition Root using a DI Container, we can make use of the facility to configure the app to use different types of dependency using different configuration options.
- We get cleaner code for deeply nested dependencies.
- For a dependency to be used by many classes, it's just a matter of registering it to the container and sitting back and relaxing.
- You can manage the lifetime of dependencies as you like. The instance can behave as a `Singleton` , `Transient` , or `Scoped` as you configure them.

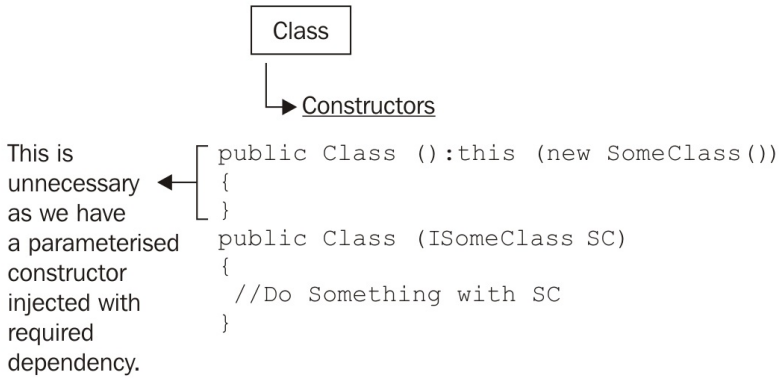
It's recommended to use a DI Container to register and resolve dependencies instead of manually managing them in the Composition Root. So, the default constructor approach is denoted as Bastard Injection. In the next section, we will explore this a bit more.

WOW! eBook
www.wowebook.org

Bastard Injection

Usually, classes have more than one constructor. You might have a situation where you have referred to some class from another assembly in your default constructor.

The following diagram shows how one class has two constructors: one default and one parameterized. The parameterized constructor handles the injection and makes the `ISomeClass` implementation available for the class to operate on. Thus, the default constructor that is creating an instance of `SomeClass` becomes unnecessary:



WOW! eBook
www.wowebook.org

Problem

For simplicity, let's consider the same example: `EmployeeService`, which needs a repository to work with:

```
public class EmployeeService
{
  private readonly IEmployeeRepository repository;
  // Default Constructor.
  public EmployeeService()
  {
     repository = CreateDefaultRepository();
  }

  // Constructor Injection can happen here.
  public EmployeeService(IEmployeeRepository repository)
  {
    if (repository == null)
    {
      throw new ArgumentNullException("repository");
    }
    this.repository = repository;
  }
  // Method creating a default repository.
  private static EmployeeRepository CreateDefaultRepository()
  {
    string connectionString = "Read String from config";
    return new SqlEmployeeRepository(connectionString);
  }
}
```

There is a default constructor which ensures repository availability by creating an instance of `SqlEmployeeRepository`. The default repository is referred from another assembly, obviously, as the service and repository don't stay in one assembly together. That's why the default can be denoted as a **Foreign Default**.

While we thought of having a default repository just because we wanted to make the service usable as soon as it is instantiated, unknowingly we designed a tightly coupled system between the service and repository.

WOW! eBook
www.wowebook.org

Solution

Bastard Injection is bad just because of this Foreign Default. Also, it might be depending on something we don't even need inside the class. From the preceding example, it's quite clear that, if we use a DI Container, it will automatically wire up the resolved dependency to the other parameterized constructor. However, if we have these kinds of default constructor, then DI Container might get confused while selecting which one to target. Having just one constructor for injection ensures a smooth operation with the container.

When you encounter a default constructor generating coupling with a Foreign Default, you can consider eliminating that, as you decide to apply Constructor Injection. Having one constructor for DI is enough because it is the responsibility of the DI Container to make sure that Foreign Default is in place whenever the service is requested.

While you refactor the code and remove the default constructor, the compiler might complain. In that case, you need to move instantiation codes to the Composition Root. If the referred dependency is a *Local Default* (it resides inside the same assembly), then we still need to remove that constructor, because constructor ambiguity leads to an increase in the complexity of *Auto-Wiring*.

If you remember, we have already discussed local defaults in `Chapter 8`*, Patterns - Dependency Injection. The easiest way to handle local defaults is to introduce Property Injection. The default constructor can be converted to a writable property.*

WOW! eBook
www.wowebook.org

Constrained Construction

Here comes yet another type of *Anti-Pattern,* that tries to hijack the constructor. So, the developer forces the dependency's constructor to have a particular signature, which creates the problem. The reason behind this might be the intention of defining the dependencies in external configuration files to achieve late binding.

Late binding can be referred to as reading the type of repository (the derived class of the repository) and connection string from the configuration file to instantiate some repository.

Having a late binding technique not only helps to isolate code from the dependency but also makes sure that the code is not recompiled if the configuration is updated. Using a commonplace or application root to define all dependencies doesn't expose any problem, but recompiling is a must when we want to update dependencies:

EmployeeRepository

```
EmployeeRepository()
{
 Read connectionString from config
 Create (connectionString)
}
```

StaffRepository

```
StaffRepository()
{
 Read connectionString from config
 Create (connectionString)
}
```

WOW! eBook
www.wowebook.org

Problem

Imagine two repositories `EmployeeRepository` and `StaffRepository` used in an application. Both have constructors and you will pass the type of repository and connection string to both of them in order to create the repositories using these params. This is bad as you will now fetch the repository type and connection string from the config which might create issues if the `config`, does not have the required keys.

Now after you fetch the repository type name from the `config`, you have to create a `System.Type` instance using that name:

```
var employeeRepositoryTypeName = "Read from config";
  //SqlEmployeeRepository
var connectionString = "Read from config";

var employeeRepositoryType = Type.GetType(employeeRepositoryTypeName,
     true);
var employeeRepository = Activator.CreateInstance(employeeRepositoryType,
   connectionString);
```

`Activator.CreateInstance` is used to call the constructor of the given type. The second param in this method is passed to the constructor of the type provided in the first param.

Suppose the type in the `config` is `SqlEmployeeRepository`. Basically, it assumes the following:

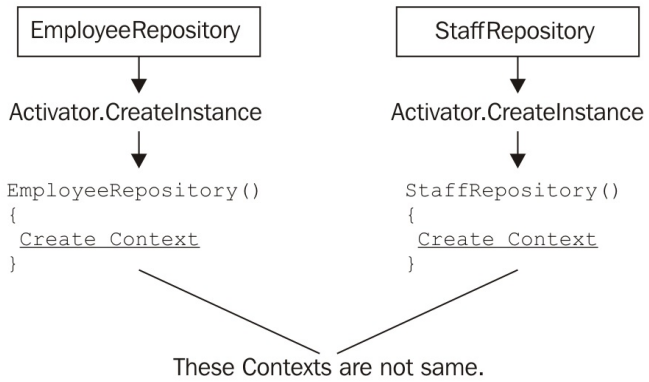
- `SqlEmployeeRepository` derives from `EmployeeRepository`
- The implementation should contain a constructor that can take a connection string as a parameter

These are the constraints, actually.

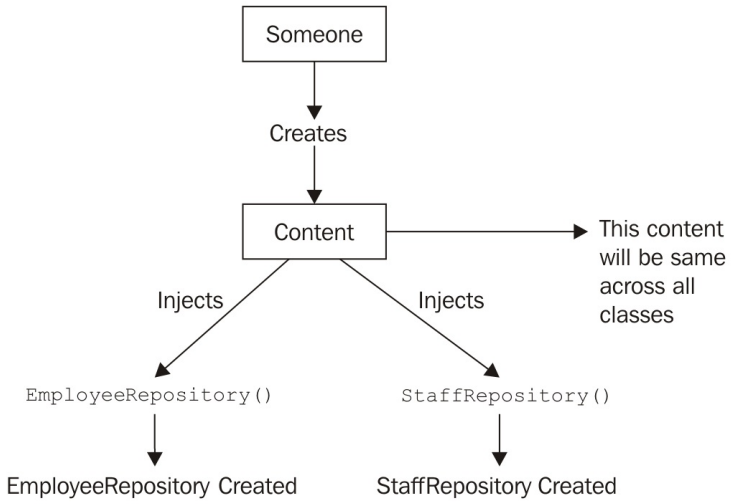
From the object construction perspective, it might look perfect, but there are disadvantages. For instance, think of constructing the repositories with the same dependency such as a single context. With the `Activator.CreateInstance`, we can no doubt create instances of the type specified to it, which in our case will create instances of `EmployeeRepository` and `StaffRepository`. But we can never assign a single context to both of them, as everyone will create their individual context based on

WOW! eBook
www.wowebook.org

their constructors. The following diagram depicts what I want to convey:



Sharing of context can only happen if someone from the outside provides that to each of them instead of their creating the context themselves:



Sharing a single dependency across the classes becomes difficult in this situation. More than one instance of the same dependency will be created, which is unnecessary. This would consume memory and resources as well.

A single instance of a particular dependency should be carefully chosen while designing. If not handled with care, there might be an adverse effect on the application running in different threads. If you remember, we have already discussed the Singleton Lifetime in `Chapter 6`*, Object Lifetime, where we talked about the pattern's benefits and usage.*

The Someone can be considered to be a Factory which we are going to explore next.

WOW! eBook
www.wowebook.org

Solution

A DI container comes to the rescue at the composition root to overcome these difficulties and resolve all the dependencies in one go. Thus, the injection can take place and everything works smoothly. There is no need to have individual constructors for the dependencies with connection strings.

Even without a DI Container, we can formulate our solution differently. Abstract Factory comes into the play when we think of centralizing the dependency construction.

First of all, the Factory, Service, and Repository should be in different assemblies. There is a strong reason behind that. We will explore it shortly.

Instead of creating every dependency object inside the *Global asax* for ASP.NET app using `Activator.CreateInstance`, we will design in a different way so that there will be no need to recompile the app every time you want to use different types of repositories.

We will design a Factory named `EmployeeServiceFactory` implementing `IEmployeeServiceFactory` using a particular repository. This factory will be responsible for creating the service. `Service` would look something like the following:

```
public class EmployeeService : IEmployeeService
{
  public EmployeeService(IEmployeeRepository repository)
  {
  }
}
```

`EmployeeServiceFactory` contains a `CreateService()` method to create the service it is responsible for (`EmployeeService` in this case).

It's important to isolate the Factory from the app and `DataAccess` to remove coupling. So, the Factory should be in a different assembly.

You can store the assembly-qualified type name for a factory in a `.config` file. `Activator.CreateInstance` can then be used to create an instance of the `IEmployeeServiceFactory` implementation (that you will define in the config) because

WOW! eBook
www.wowebook.org

that has a default constructor:

```
var employeeFactoryTypeName = "Read from config";
var employeeFactoryType = Type.GetType(employeeFactoryTypeName, true);
var employeeFactory = Activator.CreateInstance(employeeFactoryType);
```

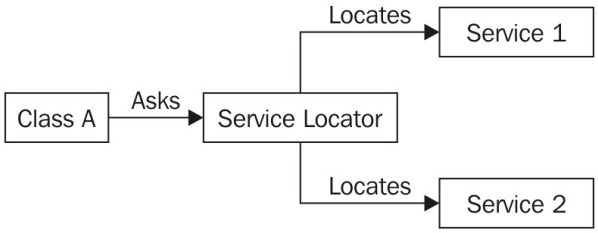
With `employeeFactory` now you can fire `CreateService()`, which will return the `EmployeeService` instance by using the related repository `EmployeeRepository`.

If you decide to use another type of dependency for the service instead of `EmployeeRepository`, then you can update the related key in the `config`. But before that, you need to add another factory in the factory assembly to implement `IEmployeeServiceFactory` and operate on the new configured repository. In this way, you recompile the factory assembly without compiling the application again and everything works as expected.

WOW! eBook
www.wowebook.org

Service Locator

If you remember, in the *Control Freak* section, we discussed Static Factories. With certain modifications, a Static Factory can behave as a Service Locator. As the name suggests, it will locate or find your required service:



Service Locator is very useful in many cases, which is why developers consider it to be a pattern. But it has many disadvantages. That's why we are listing this in this chapter instead of the previous one.

I am not going to discourage you from using this technique, but I would like to shed some light on its pros and cons. This will definitely help you to make better design decisions according to your application.

Some noticeable properties of Locator include the following:

- The important logic behind *Locator* is that it allows dependencies or services to be injected into it easily
- A Static Factory, when configured with services or dependencies, is usually considered a *Service Locator*
- The configuration of a Service Locator can happen at the Composition Root
- Service Locator configuration can be managed through code or reading-related settings from config

A DI Container looks like a *S*ervice Locator. The principal duty of a Locator or Container in the DI Context is to resolve the dependency graph before every other operation starts. Ideally, resolving the graph should only happen at the Composition Root for proper implementation. The problem starts when you ask for dependencies or services in the application directly using the Locator or Container, instead of injecting them to the consumer. In this case, Locator will be tagged as an anti-pattern.

WOW! eBook
www.wowebook.org

The Locator Process can be defined as follows:

1. **While registering a service, Locator stores one instance of it in a Dictionary**: The registration is normally done using an interface. You basically inform the Locator about an interface and its concrete implementation. For example, you would say *Serve* `EmployeeService` *if* `IEmployeeService` *is requested, Serve* `StudentService` *if* `IStudentService` *is demanded*.
2. **Locator receives the request to serve one instance of a service registered by its interface**: So, when some code in your application wants to work with a `Student` entity, it will ask the Locator for an `IStudentInterface` implementation.
3. **Locator searches for the instance from the stored instances and then returns that to the client:** As you have already trained the Locator, it will just return the requested implementation of the interface by checking all the stored instances.

WOW! eBook
www.wowebook.org

Design

A simple Service Locator would resemble the following class:

```
public static class ServiceLocator
{
  static Dictionary<Type, object> servicesDictionary =
    new Dictionary<Type, object>();
  public static void Register<T>(T service)
  {
     servicesDictionary[typeof(T)] = service;
  }
   public static T GetService<T>()
   {
     T instance = default(T);
     if (servicesDictionary.ContainsKey(typeof(T)) == true)
     {
        instance = (T)servicesDictionary[typeof(T)];
     }
     return instance;
   }
}
```

The `Register` method stores the service in the dictionary and `GetService` returns. We can use the locator to get the instance of a particular type:

```
public class EmployeeService : IEmployeeService
{
  private readonly IEmployeeRepository repository;
  public EmployeeService()
  {
    this.repository = ServiceLocator.GetService<IEmployeeRepository>();
  }
 }
```

If you have registered the service beforehand using the `Register` method, then you can get it from the dictionary.

WOW! eBook
www.wowebook.org

Advantages

Of course, the pattern has some advantages:

- It supports late binding by altering the registration codes.
- It adopts program against interfaces so that we can develop code in parallel and we can replace modules as per our requirements.
- Separation of concerns can be achieved. We can write maintainable code, though it is not so easy.

Don't get fooled by the advantages. It might look perfectly fine to you, but there are many disadvantages with this approach.

WOW! eBook
www.wowebook.org

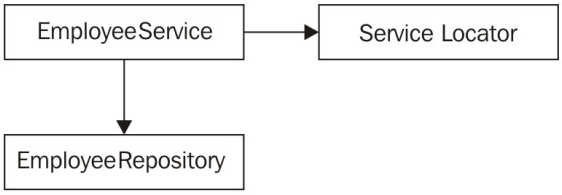
Problem

Service Locator acts as a proper pattern; however, you have to live with the following issues:

WOW! eBook
www.wowebook.org

Code reusability

It hampers the reusability of the class as dependencies are not centralized anymore with Locator. They might be scattered throughout the class by the Locator's `GetService` method:



`EmployeeService` now depends upon both `EmployeeRepository` and `ServiceLocator`. Ideally, it should just rely upon the repository to follow the DI principle.

With both these dependencies in place, if someone wants to reuse `EmployeeService`, then they have to refer to both of them. If `ServiceLocator` stays in a different assembly, then the assembly reference is also required, which draws a very inefficient design. You will definitely agree with me if I say that it is a tightly coupled architecture.

Moreover, the consumer of the service won't be able to identify the dependencies while instantiating the service. That is because Locator is used internally inside the constructor or the methods and not exposed, unlike the DI strategy:

```
var empService = new EmployeeService();
```

Now you might argue, Why would the developer would try to know what's inside because the dependencies are internally handled inside the default constructor? But what will happen if you forgot to register the dependency to Locator? Don't say that you won't forget. Most likely it will happen because, while you instantiate the class, the dependency's existence is not clear through the constructor. Therefore, if the class is not telling you what it depends on, you won't be so careful in finding and registering it, which would throw up unexpected exceptions.

I know what you are thinking now; apparently, the service is a type of Control Freak, isn't it? It is taking control of the dependencies using Locator.

The needy class means the class that depends on someone else. It no longer follows

WOW! eBook
www.wowebook.org

DI because the dependency is not injected; rather, it is fetched from Locator's `static` dictionary.

Another problem is clearly identifiable when the developer wants to add more dependencies to the class that has already adopted the Locator Pattern. Either you follow the same principle to introduce more dependencies or remove the Service Locator pattern and implement DI. Compilation errors are a certainty in both these cases.

For all the preceding reasons, Service Locator is considered an anti-pattern. Let's talk about a robust solution to address this anti-pattern.

WOW! eBook
www.wowebook.org

Solution

As usual, Constructor Injection is the best fit and comes to mind first when we decide to refactor Service Locator codes. Complete removal of Service Locator can be achieved through Constructor Injection.

WOW! eBook
www.wowebook.org

Refactoring steps

In most cases, Locator is used everywhere in the code base to get an instance of the dependency. Follow these steps to refactor it towards DI:

1. Identify all the Locator calls in the code base.
2. Introduce a member variable to the class to hold the dependency, if that is not present.
3. Mark the field `readonly` so that it can't be modified outside the constructor.
4. Assign the field inside the constructor using the Locator. Now the Locator call is in one place.
5. Add a constructor parameter for the dependency.
6. Remove Locator from the constructor block and directly assign the constructor parameter to the `readonly` field.
7. Identify all the instantiation made to the class and move the wiring to a Composition Root.

After all these steps, the same `EmployeeService` with the Constructor Injection Pattern can be designed, shown as follows:

```
public class EmployeeService : IEmployeeService
{
  private readonly IEmployeeRepository repository;
  public EmployeeService(IEmployeeRepository repository)
  {
    this.repository = repository;
  }
}
```

The service now asks its consumers to provide a dependency of type `IEmployeeRepository` implementation, which was not happening before.

WOW! eBook
www.wowebook.org

Summary

The last chapter illustrated ways to implement DI. When we don't implement patterns correctly, we end up with a bad design for our app. We learned about mistakes we regularly make while implementing DI.

Before moving on to DI anti-patterns, we discussed why and when we can consider DI as an anti-pattern itself!

Then we moved ahead and touched upon all those common anti-patterns caused by misconceptions regarding Dependency Injection. We explored Control Freak, Bastard Injection, Constrained Construction, and (the most important) Service Locator.

Control Freak is the easiest one to discover. Whenever you see any class using the `new` keyword to instantiate its dependencies, that means it is trying to manage them without any external module controlling it. This is bad and should be avoided in the DI ecosystem.

This is the most dangerous one, which we should address first while refactoring. Other patterns are less damaging than this because it has a direct impact on loose coupling. Composition Root should be the place to instantiate everything needed for the application and then every possible dependency will be available by injection.

Bastard Injection can be seen with Foreign Defaults which is unnecessary from the DI point of view. We can easily move to Constructor Injection by removing codes related to Foreign Defaults.

On the other hand, Constrained Construction imposes restrictions on constructors to support late binding of services by taking types from configuration files, which in turn creates tight coupling. Adopting a DI Container or one Abstract Factory would help to eliminate constraints.

Last but not the least, we discussed Service Locator, which is arguably a proper Pattern. However, we explored its pros and cons and came to the conclusion that it is an anti-pattern.

WOW! eBook
www.wowebook.org

If you have reached this point, that means you have now have grasped the ways to implement DI and what to avoid. In the next chapter, we will be discussing more realistic problems in projects and how to deal with them.

WOW! eBook
www.wowebook.org

Dependency Injection in Other JavaScript Frameworks

In Chapter 9, *Anti-Patterns and Misconceptions on Dependency Injection*, we covered the most important anti-patterns when using Dependency Injection, together with some typical misconceptions when it comes to how to use it. In this chapter, we're going to deal with Dependency Injection in other framework, specifically in TypeScript 2.3 and Angular 4+.

The reason for reviewing the very basics of TypeScript is that Angular 2+ uses this language, so we need some understanding of how class creation and module management is undertaken to really grasp the main concepts behind this architecture.

In this chapter, we will cover:

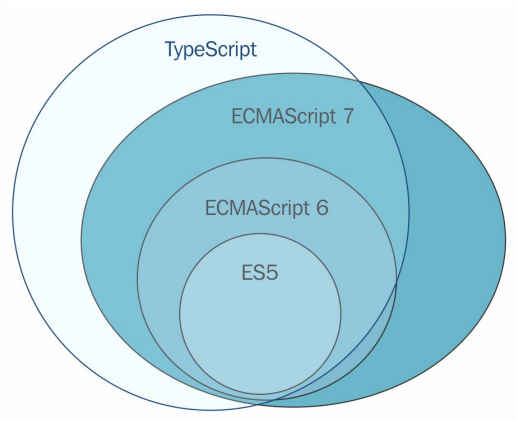
- Class creation and module management in TypeScript 2.3+
- Native implementation and usage of DI techniques in AngularJS (version 1.x)
- Native implementation and customizable options to use DI in Angular 4+

WOW! eBook
www.wowebook.org

TypeScript

You probably already know the role that TypeScript plays in the development of modern JavaScript frameworks, and even by itself as an improved replacement for JavaScript.

TypeScript has many times been defined as a strongly typed superset of JavaScript. It is a superset because it includes everything JavaScript has (and that extends to versions 3, 5, and 6, also called ES2015), plus a number of features to allow programmers to code in an object-oriented paradigm. At the time of writing, the latest version available is 2.3:



(Image courtesy: http://blog.soat.fr/2016/08/feedback-typescript/)

Consequently, we find many syntactical benefits while it still lets you write normal JavaScript if you want to (an important point is that any valid JavaScript is valid TypeScript).

In this way, TypeScript fosters a more declarative style of programming by using interfaces and static typing, it offers the concepts of modules and classes, and it's capable of integrating well with most existing JavaScript libraries and code. We can see it as a strongly static layer over JavaScript with a bunch of features to make the programmer's work (especially debugging) much more manageable.

If you want to get a nice introduction to the TypeScript language, you can read Introducing Object-Oriented Programming with TypeScript, by Remo H. Jansen, (see https://www.packtpub.com/books/content/introduci

WOW! eBook
www.wowebook.org

ng-object-oriented-programmng-typescript) and, if you prefer to delve deeply into the language and its possibilities, take a look at the excellent Mastering TypeScript by Nathan Rozentals, available at https://www.packtpub.com/web-development/mastering-typescript.

The main goals of Anders Hejlsberg (the TypeScript Chief Architect) when creating the language were:

- Creating a totally object-oriented language that would convert into JavaScript at compilation time (it's called **transpilation** since it just produces another language and not a compiled module), allowing the final JavaScript generated to be executed in any browser (or Web platform).
- Make the language statically typed, so the tools could offer modern development technologies in any editor: Intellisense, Code Completion, Intelligent Refactoring, and so on.
- Engage the community in the project by making the language completely open source. You can see the current state of the project and collaborate on its website at http://www.typescriptlang.org/.

Actually, TypeScript has been so successful that the Angular development team adopted it for the creation of Angular 2 and keeps on working with the language for coming versions (the most recent being Angular 4.1).

WOW! eBook
www.wowebook.org

Architectural changes

Understanding some architectural changes is fundamental to realize how you can implement DI in a language that, once transpiled, is not object-oriented, except if you are doing that transpilation into ES2015.

One of the biggest changes in ES2015 (or ES6, for short) is the presence of modules. A module is, basically, a piece of JavaScript code stored into a file. Once written, you can say that you have one file per module and one module per file.

WOW! eBook
www.wowebook.org

Modules in TypeScript

TypeScript defines two different types of modules--internal and external. Moreover, we could still differentiate internal ones into two more categories: those that have a name and those that don't (you could call them implicit). The difference, in this case, is in the way you define them and use them.

Say you have a TypeScript file including the following:

```
// Implicit module (goes to the global namespace)
// It appears as part of the window object
class ClassA {
  private x = "The String";
  public y = 4;
};
var ca = new ClassA();
```

This code by itself is an implicit module. It becomes a part of the global namespace and you can find it inside the `window` object at runtime. You could also consider the global namespace as the implicit (default) module.

Of course, we have better solutions than polluting the global namespace. The most obvious one is the `module` keyword which allows defining a private code section.

By definition, everything declared inside a module is private to that module. Therefore, the new reserved words `import` and `export` are used to allow access to a piece of code inside a named module.

If we wrap the previous class inside a module definition, any attempt to reference a module's member outside the module is not recognized (see the following screenshot, inside the Visual Studio 2017 editor):

WOW! eBook
www.wowebook.org

```
module MyClasses {
    export class ClassA {
        private x = "The String";
        public y = 4;
    };
    var ic = new ClassA();
}
ic.y = 9;  // Not recognized
```

any

Cannot find name 'ic'.

```
va          asses.ClassA();
co          ; // Logs 4 in the cons
```

To make a module's members available outside the module you should use the `export` keyword. Change the previous declaration into this:

```
export class ClassA
```

Then, you can use the module's name to access its public members:

```
var c = new MyClasses.ClassA();
console.log(c.y); // Logs 4 in the console window.
```

Notice that the `ClassA.x` member is not accessible either due to its private declaration. In this manner, we have a convenient way to keep things tied to the namespace they should belong to.

WOW! eBook
www.wowebook.org

External modules

However, the most useful approach when you're dealing with large applications is the use of external modules. Actually, as John Papa suggests (https://johnpapa.net/typescriptpost4/), it may be useful to organize the functionality of an application in terms of the modules you'll need.

Let's say we have a module definition inside the `ES6Code.ts` file. To export the `foo()` function and `Timer` class, you declare them as follows:

```
// File ES6Code.ts
module ES6Code {
  export function foo() { console.log("Foo executed"); }
  export class Timer {
    localTime: string;
    currentDate: string;
    constructor( todaysDate:Date ) {
        this.localTime = todaysDate.toTimeString();
        this.currentDate = todaysDate.toLocaleDateString();
    }
  }
}
```

Now, in a distinct module or script section we can access that functionality using a slightly different reference:

```
// File: app.ts ---------
// Simple Function Import
import foo = ES6Code.foo;
foo();  // OK.
```

The same goes for the `Timer` class, except that we can also adopt another approach to reference the class:

```
// File: app.ts ---------
// Reference Class Import (alternative syntax)
var timer = new ES6Code.Timer(new Date());
console.log(timer.currentDate);
console.log(timer.localTime);
```

In this case, the module's name is used as a prefix to the class instantiation, in a similar way to what you find in C# code when dealing with namespaces.

To test this code, we just have to include references to the corresponding transpiled files (`.js` extension), inside an HTML page, or test it inside the Node console. For

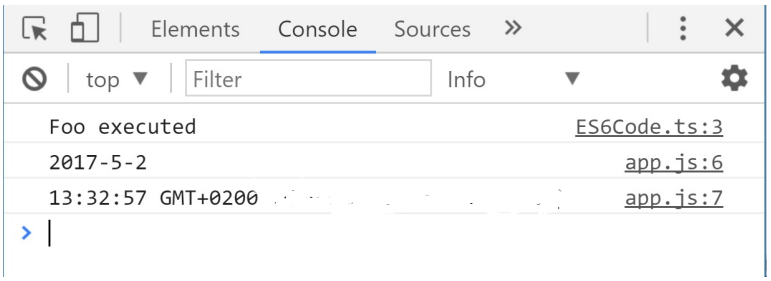
WOW! eBook
www.wowebook.org

instance, if you use a blank HTML page, you could include the files in this manner:

```
  <head>
    <meta charset="utf-8" />
    <title></title>
    <link rel="icon" href="data:; base64,iVBORw0KGgo=">
    <script src="ES6Code.js"></script>
    <script src="app.js"></script>
</head>
```

Note: The `<link>` tag is a fake favicon to prevent Chrome from presenting a File Not Found error in the console, next to the output.

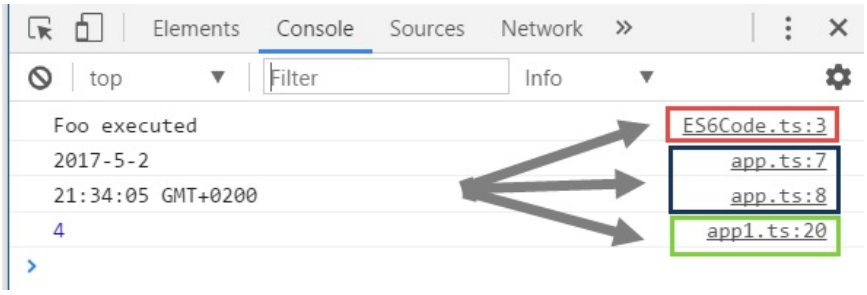
In both cases, the reference is correct, and the code is executed as expected, as we can see in the following screenshot (remember: `F12/Console` inside the developer's tools of any modern browser. I'm using Chrome here):



This is possible because, once the files are referenced inside a page, their members become accessible as long as they are marked as `export`.

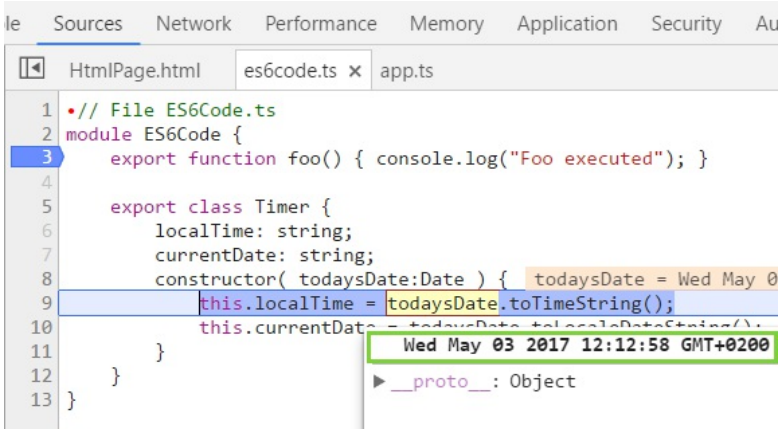
Actually, we can reference several files, and they'll be loaded and executed in sequence. Notice also that the console window indicates which code is responsible for the execution of which entry in the console.

In the initial example, `MyClasses` was stored inside the `app1.ts` file. So, now, we could also reference it, before the `</head>` tag, and check how a third source origin shows up in the console, which outputs the number 4, in the last position (check it in the following screenshot):



WOW! eBook
www.wowebook.org

Also, you'll notice that Chrome is smart enough to reference the original `.ts` files and not the transpiled ones, allowing you to debug any of these files, setting breakpoints, and so on. In the following screenshot, I'm showing the debugging window inside Chrome after setting a breakpoint before printing the date and time values:



Besides these options, now it's possible to debug within Visual Studio 2017 using Chrome as default browser.

This module separation is important because, as you'll see later, that implies file separation, which becomes crucial in the way TypeScript and, especially, Angular organize the distinct components of an application.

WOW! eBook
www.wowebook.org

Dependency Injection in TypeScript

All that said, TypeScript itself doesn't have a DI container. However, you could use some of the third-party options available. Some of the most popular are `Infuse.js` (available at `https://github.com/soundstep/infuse.js`) and TypeScript IoC, which you can find on the NPMJS site (`https://www.npmjs.com/package/typescript-ioc`), both of which work in a very similar way to what we've already seen in .NET Core.

That is, you have to make definitions that map an interface to a class or even a descriptor to a class and then register those options. Later on, you can make a reference to the type required and expect the DI container to provide you with the corresponding type.

Let's review the way TypeScript IoC works to define a simple injection scenario, the way it's shown on its official page.

First, assuming you have TypeScript installed, you should also install TypeScript IoC, with a typical `npm` command:

```
npm install typescript-ioc
```

Besides that, some modern options are also required inside the configuration file (the `tsconfig.json` file):

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

Once that is configured, you use one of the `import` syntax variants to put it to work:

```
import {AutoWired, Inject} from "typescript-ioc";
class PersonDAO {
  @Inject restProxy: PersonRestProxy;
}
```

As you can see, the `restProxy` property, which is of another type (`PersonRestProxy`), is marked with an `@Inject` decorator (a new feature in the latest versions of JavaScript and available in TypeScript), to indicate that it can be injected later on inside other

WOW! eBook
www.wowebook.org

code).

Somewhere else in the code, you'll be able to use these definitions with a very simple syntax:

```
let personDAO: PersonDAO = new PersonDAO();
```

The `restProxy` property will be provided by the container, resolving the dependencies. Parameter injection is also provided, using a class with a constructor, as we can see in the following code:

```
class PersonService {
 private personDAO: PersonDAO;
 constructor( @Inject personDAO: PersonDAO ) {
   this.personDAO = personDAO;
 }
}
```

If, later on, you have another class that uses `PersonService` as a property, you mark that property with `@Inject`, like in this code:

```
class PersonController {
 @Inject private personService: PersonService;
}
```

You can rely on the chain of dependencies being managed by the container, which will go through all references previously marked as `@Inject`.

However, in practice, it's rare to see applications in TypeScript itself and the most common use of this language is to provide other frameworks, such as Angular or Ionic, with a consistent development language.

WOW! eBook
www.wowebook.org

Angular

As mentioned in Chapter 2, *Dependency Injection and IoC Containers*, Angular is a development framework created by a team at Google (led by MiskoHevery), that has become very popular now (you can access the official information at http://angularjs.org).

Nowadays, Angular is available in two flavors that follow different release paths or branches:

- **Version 1.x**: Also called **AngularJS**. It's recommended for small/medium applications and uses the MVC model to implement a suitable separation of concerns, from the beginning:
  - It's composed of a set of JavaScript libraries, each one supplying a part of the functionality required. However, all libraries depend on the basic AngularJS library.
  - At the time of writing, the latest version is 1.6.4, and the team guarantees support and future updates. It can also be downloaded via CDN at https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js.
- **Version 2+**: It's just named **Angular** and it follows a semantic versioning path, which means no breaking changes are allowed in minor revisions, only in new versions. There was a version number jump that avoided 3.0, so the latest version is Angular 4 (4.1, to be precise). It has a dedicated website apart from AngularJS at https://angular.io/:
  - Perhaps the biggest change is that Angular is not backward-compatible with AngularJS since it embraces a number of changes that appeared in ES6 and directly affects the way modules are managed.
  - Its main features are cross-compatibility, improved speed and performance, excellent tooling, and an increasing adoption by the community all over the world.
  - Angular is built in TypeScript thanks to a joint venture with Microsoft's TypeScript team, which actively collaborates in the project. Recently, one of the lead members of the team announced: *TypeScript has become allowed for unrestricted client development as of March 2017. TypeScript and Angular on TypeScript are used in Google Analytics, Firebase, and Google Cloud Platform and critical internal tools such as bug tracking, employee reviews, and product approval and launch tools*.

WOW! eBook
www.wowebook.org

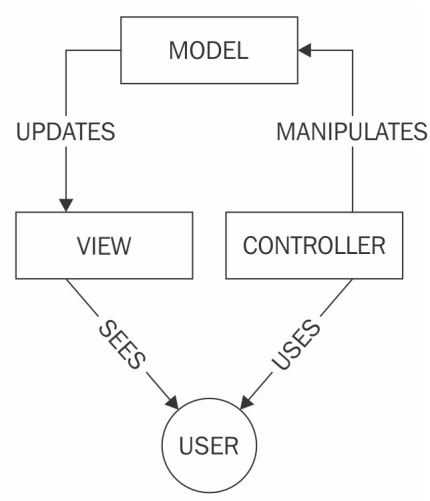
Both versions have some points in common from an architectural point of view. They foster SOLID principles, and especially SRP and DIP, so they both rely on Dependency Injection containers. Furthermore, the adoption of the MVC model helps to structure the separation of concerns.

Let's see how these two popular architectures implement Dependency Injection (although in different ways).

WOW! eBook
www.wowebook.org

AngularJS

Creating a very basic Angular application is easy, once the basic principles of the MVC are clear. The MVC model proposes a foundational separation of application components into three parts (see the following figure):



(Image courtesy: Wikipedia: https://en.wikipedia.org/wiki/Model-view-controller)

The user's interaction generates a circuit in which the three pillars of MVC are present:

- The **MODEL** is created automatically in the loading process of the AngularJS library.
- The **VIEW** corresponds to the HTML side, which is marked with custom attributes (all starting with ***ng-***) to indicate the distinct functionality required. Views also use a syntax called **mustache** to indicate which parts are subject to data-binding.
- The **CONTROLLERS** are JavaScript fragments, coded to reflect any changes requested by the user, and manipulate the Model if required.

When the user interacts with a UI element in the page, a controller takes care of the corresponding action, modifies the model if it is necessary, and Angular updates the view reflecting the changes. Finally, the view is received by the user and this cycle continues uninterruptedly.

WOW! eBook
www.wowebook.org

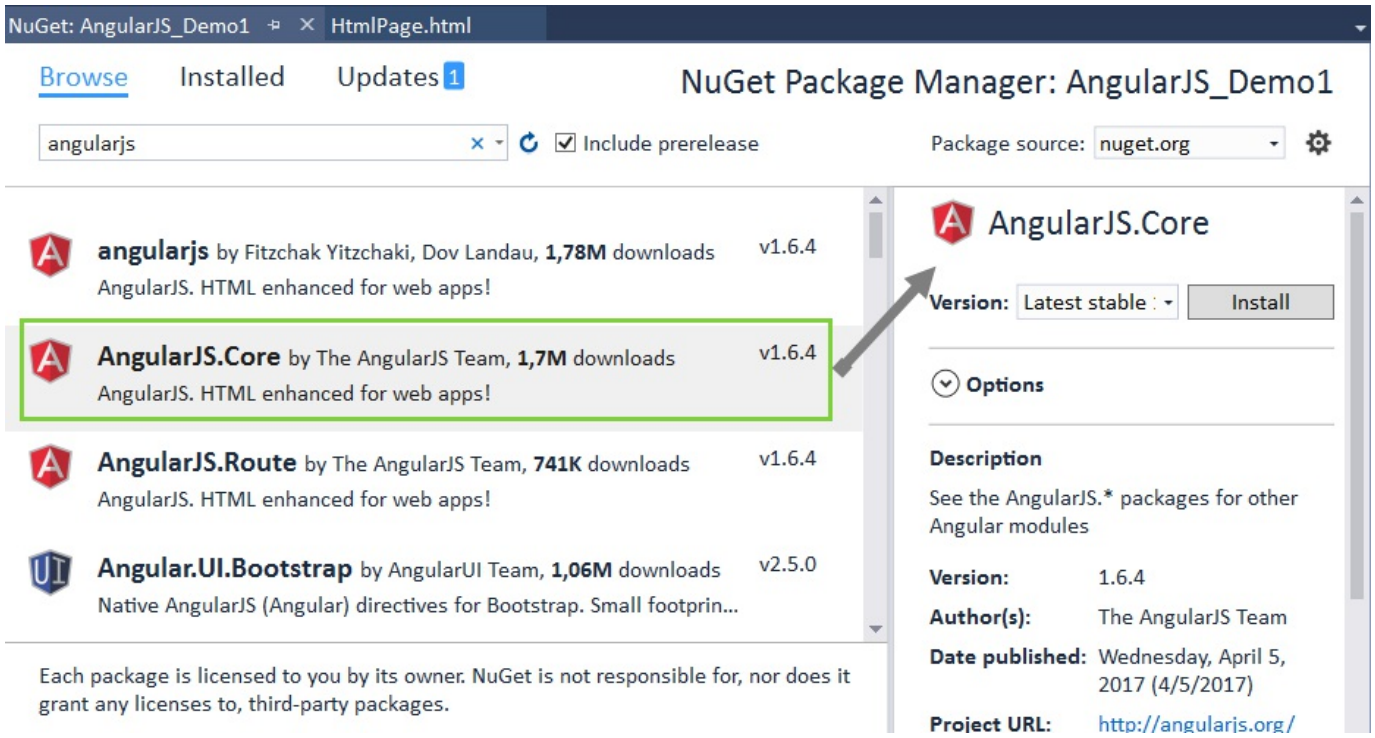
Examples using Visual Studio 2017

Once you have an HTML page that loads the AngularJS library, you should mark one DOM element with a special attribute (ng-app) to notify Angular of its working scope. This defines the area of action for AngularJS inside the page.

Let's see all this in a very simple example. We can use Visual Studio 2017 as well, for example creating a new website (notice that we don't need a project since we don't need any compiled code on the server: it all happens on the client side).

So, I suggest choosing an ASP.NET Empty Website for this initial demo. That will create a folder to hold the solution and will include a `Web.config` file just in case you want to instruct the server about some behavior or require some compiled code as well.

Next, we add an HTML page and save the solution to be able to use the Manage NuGet packages option and search for the AngularJS.Core library. Make sure you're selecting AngularJS.Core and not just angularjs if you don't want to be overloaded with all available libraries for this framework (see the following screenshot):



WOW! eBook
www.wowebook.org

Once installed, a new Scripts folder will show up in the Solution Explorer, including a few libraries. You just need to drag and drop the angular.js library inside the `<head>` tag, to let Visual Studio create a `<script>` tag pointing to the library and you're ready to go!

The next step is adding the `ng-app` attribute (for example, to the `<body>` tag) and giving it a valid name, such as `app`. Currently, we have a page, the angular library loaded, and a scope defined.

How can we see some AngularJS in action with this? We can create an HTML tag, such as `<h2>`, `<h3>`, `<div>`, `<article>`, and so on, and include a moustache link inside (they're called AngularJS binding expressions) that should be resolved at runtime, for example, a `<h2>` tag that will show the current time. In all, up to this point, we should have a page like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>AngularJS initial demo</title>
  <script src="Scripts/angular.js"></script>
</head>
<body ng-app="app">
  <h2>Current time: {{time}}</h2>
</body>
</html>
```

Observe the two identifiers (`app` and `time`) that are not defined yet. That will be our JavaScript section of this page. So, before the closing `</body>` tag, we will include an `<script>` tag with the following code:

```
<script>
  var app = angular.module("app", []);
  app.controller("TimeController", function ($scope) {
     $scope.time = new Date().toLocaleTimeString();
  });
</script>
```

This needs some explanation: first, we create a `module` named `app`. That is an object managed by AngularJS to establish the DOM area of interest, and it is created by calling the `method` module on the `angular` object that the library created at loading time. We cache the return value in a variable of the same name just to make the following code a bit clearer.

The next step is crucial. Here we create a controller called `TimeController` and assign a callback function to it. Well, that function is using Dependency Injection by default!

WOW! eBook
www.wowebook.org

If you notice the function's definition, there's a `$scope` variable defined. Where does it come from?

The explanation is that, in several angular's constructions, when you define a callback function and declare a recognizable service as a parameter (such as `$scope`), a singleton instance of that service is provided by the `$injector` object to you with no intervention on our own.

And it is precisely that `$injector` object that provides DI container services in AngularJS. Every object managed by a module has its own instance of that service and takes care of resolving all dependencies.

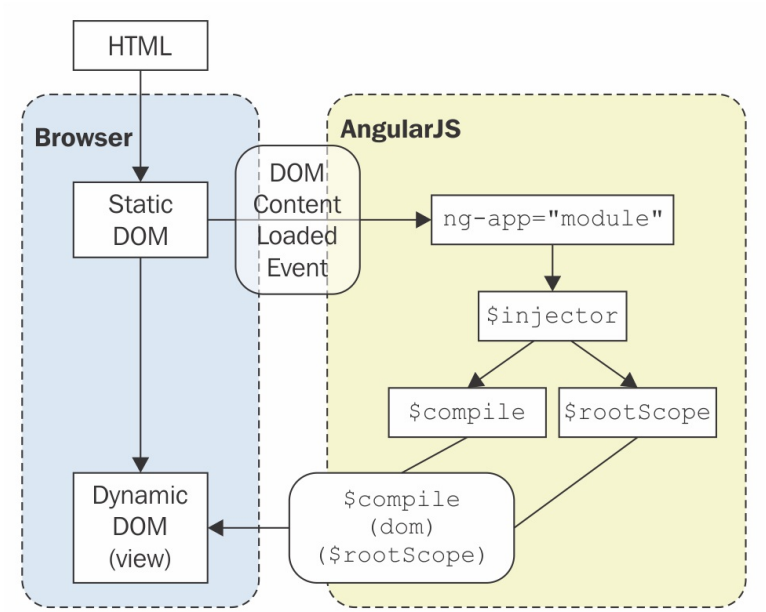
The official AngularJS documentation defines its implementation in this way:

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

WOW! eBook
www.wowebook.org

Understanding the structure of AngularJS bootstrap

The following diagram shows the structure of the bootstrap process that takes place when loading an Angular app:



The detailed explanation is this: HTML elements marked with attributes starting with `ng-` denote a part of the DOM called Dynamic DOM (where no `ng-` attributes are present, considered to be static DOM).

Let's briefly reproduce the steps used:

- When the DOM is loaded, AngularJS searches for a `ng-app`-marked element and defines a `$injector` linked to it.
- In turn, that injector defines a `$compile` service that teaches the HTML interpreter some new syntax. To be precise, the documentation explains this point by stating that:

> *The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior. AngularJS calls these behavior extensions directives.*

WOW! eBook
www.wowebook.org

- With all that in memory, AngularJS also creates a special service called `$rootScope`, which serves as the root model for the module. Of course, you can use it as well in your own code.
- Now, every controller that you create has a child of that `$rootScope`, called simply `$scope`: that's the section of the model that such a controller manages.
- As the last step, the `$compile` object traverses the module, searching for elements that have `ng-*` attributes, here called directives, or AngularJS expressions (`{{moustache}}` annotations), and substitutes those elements with the required data or code.

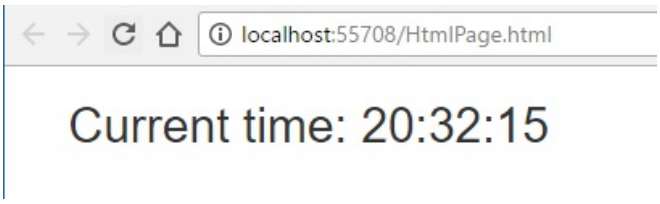
Consequently, the final aspect of our HTML code will be (I'm including only the contents of the `<body>` element):

```
 <body ng-app="app">
  <h2 ng-controller="TimeController">Current time: {{time}}</h2>
<script>
   var app = angular.module("app", []);
   app.controller("TimeController", function ($scope) {
      $scope.time = new Date().toLocaleTimeString();
   });
  </script>
</body>
```

So, what I'm doing here is creating a model's variable (`time`) assigned to the controller named `TimeController`. Its value is equal to a string representing the current's system time.

Finally, we need to indicate which AngularJS element is managed by which controller: in this case, it is the `<h2>` element containing the `time` variable defined in the model. At runtime, AngularJS substitutes the variable's value for the `{{time}}` expression.

The output in the browser changes every time you refresh the page (see the following screenshot):



This is a very simple demo, but it illustrates the basics of AngularJS and how DI is pervasive throughout the framework since you'll find it all over.

WOW! eBook
www.wowebook.org

Data access and Dependency Injection

Actually, we can see a better implementation of a controller and how to use the `$injector` object if we access some real data using another AngularJS service, called `$http`.

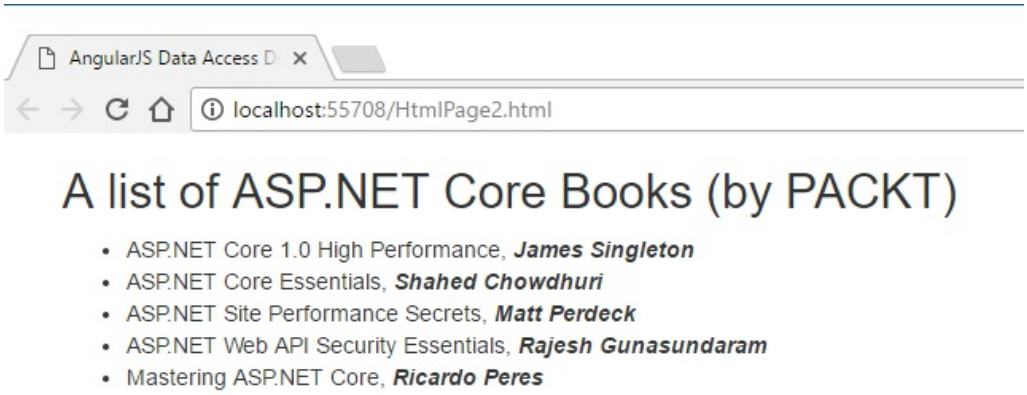
The following code reads all data from the `BookStore2.json` file that we used in Chapter 9, *Anti-Patterns and Misconceptions on Dependency Injection*, demos and creates a list of its elements. I'll show you the code first, and then we'll move to the explanation:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>AngularJS Data Access Demo</title>
  <link rel="icon" href="data:; base64,iVBORw0KGgo=">
  <link href="Content/bootstrap.css" rel="stylesheet" />
  <script src="Scripts/angular.js"></script>
</head>
<body ng-app="app" class="container">
  <h2>A list of ASP.NET Core Books (by PACKT)</h2>
  <ul ng-controller="ListController">
    <li ng-repeat="book in Books">
        {{book.Title}}, <strong><em>{{book.Author}}</em></strong>
    </li>
  </ul>
  <script>
    angular.module("app", []);
    var ListController = function ($scope, $http) {
           $http.get("BookStore.json").
           then(function (response) {
                 $scope.Books = response.data;
           }).
           catch(function (error) {
                 alert(error.statusText);
           });
    };
    ListController.$inject = ["$scope", "$http"];
    angular.module("app").controller("ListController",
       ListController);
  </script>
</body>
</html>
```

Before commenting this out, let us note that I'm also using the BootStrap library, but that's only for presentation purposes; it has nothing to do with DI.

WOW! eBook
www.wowebook.org

If you look at the previous code in any browser, it will present an output very similar to the one that's shown in the following screenshot:



Let's review the most important changes in this code. On the JavaScript side:

- Module creation is not cached into a variable (this avoids unnecessary variables in the global space).
- The controller is created using a function expression assigned to a variable of the same name. It declares two variables to be injected later on:
  - `$scope`: The reference to the model linked to the controller
  - `$http`: A service that facilitates request/response actions with HTTP servers, either using the `XmlHttpRequest` object (AJAX) or JSONP inside AngularJS
- Inside the function, `$http` is used to get data by calling its `get` method and passing it the URL of the resource to recover. That call returns a promise, which is resolved asynchronously:
  - When the promise is resolved, it returns the expected information, and the callback functions linked to it will receive an object with the following properties, according to the official site's documentation:
    - `data{string|Object}`: The response body transformed with the transform functions
    - `status{number}`: HTTP status code of the response
    - `headers{function([headerName])}`: Header getter function
    - `config{Object)`: The configuration object that was used to generate the request
    - `statusText{string}`: HTTP status text of the response
  - If the status code is a number between 200 and 299, the process is successful, and a subsequent call to `.then()` receives the information inside its `[parameter].data` property
  - Otherwise (if you get a different status code) there's an error and you

WOW! eBook
www.wowebook.org

should catch it with an extra call to `.catch()`, just as we do in the preceding code

- The `$injector` object is able to resolve the names of the variables as singleton instances of the services they represent, but what happens when you package your application and the *minifiers* change the name of those variables? This is where the `$inject` array comes in:
  - It allows the *minifiers* to rename the function's parameters and still be able to inject the referred services. Notice that it is an array linked to the controller, and it can grow or shrink at will.
- The final step is to define the controller itself inside the module. That's why we use the `angular.module("app")` syntax, which accesses the module and invokes the required methods inside (notice this time we don't pass the second argument: that means to access, not creation).

Of course, there is much more to AngularJS using DI and the framework itself, but I hope now have some idea of how Dependency Injection is implemented here. Let's summarize and complete the most important points about this pattern inside AngularJS.

WOW! eBook
www.wowebook.org

Summing up Dependency Injection features inside AngularJS

Dependency creation in AngularJS is the responsibility of the `injector` object. This object, by the way, uses the constructor injection paradigm. Actually, the injector behaves as a service locator that takes care of construction and look up for dependencies.

This is achieved using the declarative notation in HTML templates. When the HTML is processed (parsed), it passes the responsibility for component creation to the injector, thus avoiding the need to pass the injector throughout the application. All that work is done "behind the scenes".

As the documentation states, working in this manner *the application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter*.

Let's now focus on the latest versions of Angular (versions 2.0+), that are, in several ways, based on many of these principles but change its implementation quite a bit since they embrace ES6 and related technologies.

WOW! eBook
www.wowebook.org

Angular 2+

Since the release of AngularJS (version 1.x) in 2009, the web has changed immensely. We now have a multitude of build systems, modular loading capabilities, and Web components available to use. JavaScript development has also come on in leaps and bounds.

Those changes were not reflected in AngularJS, so it was not as performant as the team wanted, mainly due to the digest cycles (related to the DOM parsing that we mentioned previously), and directly related to change detection.

Change detection in Angular can be imagined as a tree working in one direction from the root to its leaves so that it's both faster and more predictable.

The name of any version higher than 1.x is just Angular, without the `JS` *suffix*

Angular has observable and immutable objects that greatly speed up how many properties are checked.

Besides, observable objects are triggered only if an input that they rely on emits an event. Immutable objects are only checked if one of their input properties has changed. In most cases though, these two types of object don't have to be checked, meaning that your whole application will be speeded up.

WOW! eBook
www.wowebook.org

Microsoft's TypeScript is the preferred language in Angular 2+

Another big change is the use of TypeScript as the preferred language (the Angular team also uses TypeScript language services and Visual Studio Code, to detect failures and inconsistencies in code, as Brad Green declared recently).

TypeScript has great support in editors such as Visual Studio Code and WebStorm and it behaves as a permanent assistant when you are importing modules and autocompleting intelligent suggestions.

Moreover, since it is a typed language, hints provided are far more in-depth than JavaScript's Intellisense. Additionally, any JavaScript is valid TypeScript, so you can use as much or as little of it as you're comfortable with. Many areas help Angular, such as interfaces, constructors, public variables, classes, typed arguments, and so on.

But perhaps the biggest change in Angular is that its architecture is based on the concept of the component. And those components are defined using class annotations or decorators, a feature that allows adding metadata to a class.

Before delving into it, let's first remind ourselves of the required tooling to work with Angular.

WOW! eBook
www.wowebook.org

Angular tools

To work with the latest versions of Angular you can certainly choose from a variety of tools, but I'm going to use those that the official site recommends, plus those you need as a requirement.

First, you need a recent version of Node.js installed. At the time of writing, two versions are available on its website (https://nodejs.org/es/): 6.10.3 and 7.10.0. Either one will be fine, although I have installed 7.10.0. This installation provides two basic tools to work with Angular: Node and **NPM** (**Node Package Manager**).

Once you have completed the installation, make sure the versions are correct by typing the following in a console window:

```
node-v
```

And by typing the following too:

```
npm -v
```

With that in place, there are a number of editors suitable for Angular, but you could try the free cross-platform Visual Studio Code, which has debugging capabilities and works fine in Windows, Linux, and OSX.

WOW! eBook
www.wowebook.org

Working with Angular

Let's install Angular CLI, a command line interface to work with Angular, which helps a lot in the initial steps with this framework. We will be able to create the early application very easily and understand how the architecture changes.

Angular CLI has a dedicated website (https://cli.angular.io/), where you'll find the installation process, downloads, and the related documentation. However, the easiest way to install it is via NPM. You just have to type the following in a Command Prompt:

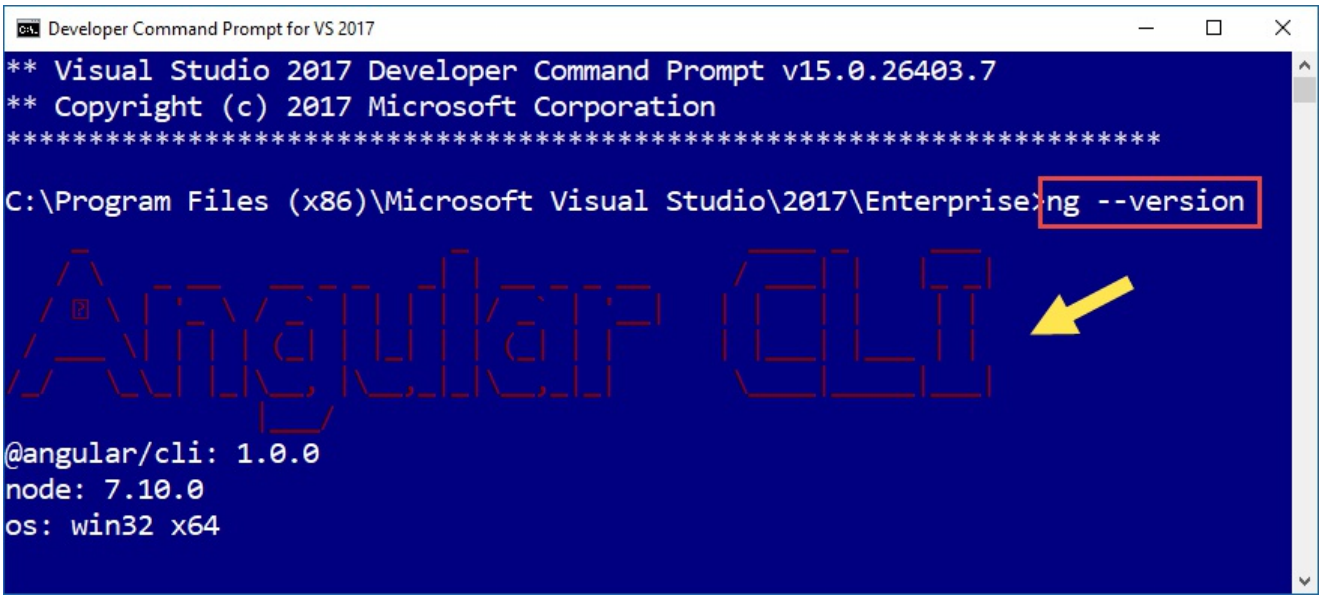
```
npm install -g @angular/cli
```

This installs the Angular-CLI tools globally, so it is available all over the filesystem.

Once again, it's a good practice to check the final state of the installation, which you can do by typing the following:

```
ng --version
```

The output should show the following information:



And we're all set! If you want to take a look at the list of commands that Angular CLI holds, just type `ng --help`. A long list will show up, so you can have an idea of how powerful this tool is in current versions.

WOW! eBook
www.wowebook.org

To create the first application, open a Command Prompt (you can use the Developer Command Prompt link that Visual Studio installs), go to the directory where you want your demos to be located, create a new directory, and type the following:

```
ng new [your-app-name]
```

In my case, I typed `ng new ng4-demo` and then you wait until the bunch of libraries from NPM is downloaded and installed in the directory of your choice.

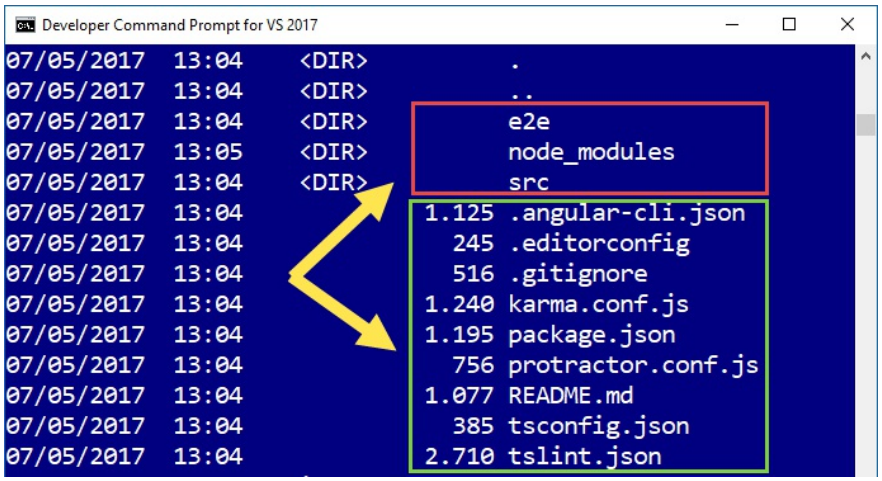
Inside the new directory, you can now see the list of files and directories created by the tool, and ready to launch. Notice that there are three new directories: `e2e`, `node_modules`, and `src`.

The first one contains end-to-end tests for the application. That's done by default and you should modify those definitions further on to suit your needs.

The second and largest one holds the large list of JavaScript libraries required for almost any Angular application. Don't get scared by its length: they're downloaded and installed locally to provide programmers with the tools they might need, but at deploying time only those required will be included in the bundling and minifying process prior to deployment (it is called **tree-shaking**).

Finally, in the `src` directory, you'll find all the files required for this initial demo. The rest are files used by Angular CLI and other tools (such as the editors) to manage the project (especially, `package.json`, `tsconfig.json`, and `angular-cli.json`).

You should end up with a list like the following:



To see it in action, just type `ng serve`. That will run the Webpack tool to get

WOW! eBook
www.wowebook.org

everything ready and launch a server on port 4200 (by default).

The final step will be to open any browser with the URL `http://localhost:4200` and see the very simple page saying `app works` (I omit the output; it is pretty obvious).

WOW! eBook
www.wowebook.org

Editing the initial project

Now that we know everything works, let's view the project inside Visual Studio Code (or the editor of your choice), and try to understand the architecture behind this.

So, I'll open the editor from the Command Prompt, just by typing `code .` on the main project's directory.

For more information about the VSCode, visit https://code.visualstudio.com/.

Perhaps, the most amazing thing about this initial demo (especially if you compare it to our previous demos in Angular 1.6) is the lack of references and directives inside the main HTML page of this app:

```
<!doctype html>
<html>
<head>
 <meta charset="utf-8">
 <title>Ng4demo</title>
 <base href="/">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
 <app-root>Loading...</app-root>
</body>
</html>
```

The only non-static HTML is marked in bold: `<app-root>`. Angular resolves this customized element into the real page you saw in the browser through a complex process of component-related architectures, in which every component defines its behavior, its visual elements (if it has any), and the way it communicates with the rest of the application.

If you look inside the `src` directory, you'll see several TypeScript files. The module responsible for instructing the browser what to do when `http://localhost:4200` is invoked appears inside the `angular-cli.json` file. This file contains a bunch of definitions about editor and server behaviors, and so on.

WOW! eBook
www.wowebook.org

It holds an app's entry containing that `main` property and also the entry point of the application, which is linked to an `index` field (see the following screenshot):

```
app.module.ts    .angular-cli.json ×
5     },
6     "apps": [
7       {
8         "root": "src",
9         "outDir": "dist",
10        "assets": [
11          "assets",
12          "favicon.ico"
13        ],
14        "index": "index.html",
15        "main": "main.ts",
16        "polyfills": "polyfills.ts",
17        "test": "test.ts",
18        "tsconfig": "tsconfig.app.json",
19        "testTsconfig": "tsconfig.spec.json",
```

With those definitions, the browser knows which page to launch, and the server which components should be resolved: whatever the `main` module indicates. And what it shows is a basic environment configuration:

```
app.module.ts    main.ts ×
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule);
```

So, this is how Angular manages the initialization of the app. Whatever is inside `AppModule` will be loaded and resolved.

WOW! eBook
www.wowebook.org

The structure of the main module

However, as it turns out that every angular application must consist of at least one module, the `app.module.ts` becomes the one that sets up the application. It's mandatory to understand how it works:

```
app.module.ts ×   app.component.ts
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4  import { HttpModule } from '@angular/http';
5
6  import { AppComponent } from './app.component';
7
8  @NgModule({
9    declarations: [ AppComponent ],
10   imports: [ BrowserModule, FormsModule, HttpModule ],
11   providers: [],
12   bootstrap: [AppComponent]
13 })
14 export class AppModule { }
```

First, we find `import` statements that load some previously exported components inside the Angular libraries, such as `platform-browser`, `core`, `forms`, and `http`. Some of these libraries are not needed in this basic demo, however.

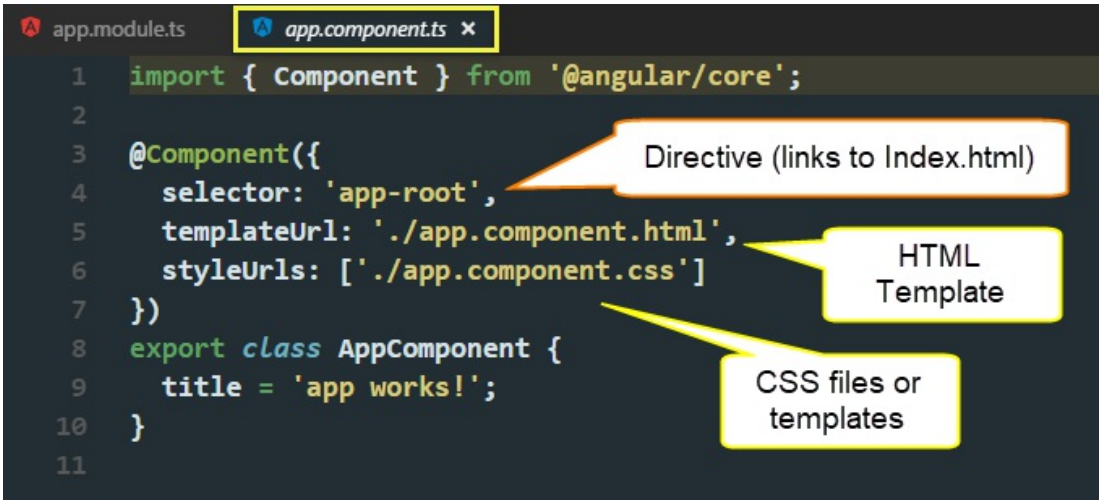
The last `import` statement is the one that links this module with the rest of the functionality, loading `app.component`.

The way you define a module is by means of a class (here named `AppModule`), marked with a `@NgModule` decorator. There are no functionalities or definitions inside. Only the decorator provides the correct link with the rest of the app's elements.

Actually, the `bootstrap` property indicates which component will be in charge of launching the application.

Let's see how that component is made up:

WOW! eBook
www.wowebook.org



Now we have the `AppComponent` class marked with the `@Component` decorator. Inside, we find the HTML part that we see in the browser. It's the title field of the class.

But, inside the decorator, we have some clues:

- `selector`: Indicates the customized piece of dynamic DOM that will be translated into a real fragment of HTML at runtime. Remember, the only foreign part of HTML inside `Index.html` is that reference to the `<app-root>` element.
- `templateUrl` (optionally just template, if indicated next): The file containing the HTML fragments that will replace the selector. They admit `{{moustache}}` syntax, as it happens here.
- `styleUrls`: An array of CSS files that will be loaded at runtime for presentation purposes. They will only affect the component that defines them, and not the whole page.

Finally, the `app works` sentence that we saw in the browser's initial page is just the `title` property of the `AppComponent` class. And, yes, it is linked via data-binding to the HTML inside the template, which is just a `<h1>` element:

```
<h1>
 {{title}}
</h1>
```

So, let's make a couple of changes, to see how it goes. Instead of `app works`, I'll change the `title` property to `First demo in Angular 4`, and the CSS file, which is empty, will also hold some formatting rules:

```
h1 {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  font-size: 2em;
  border: 3px solid navy;
```

WOW! eBook
www.wowebook.org

```
}
```

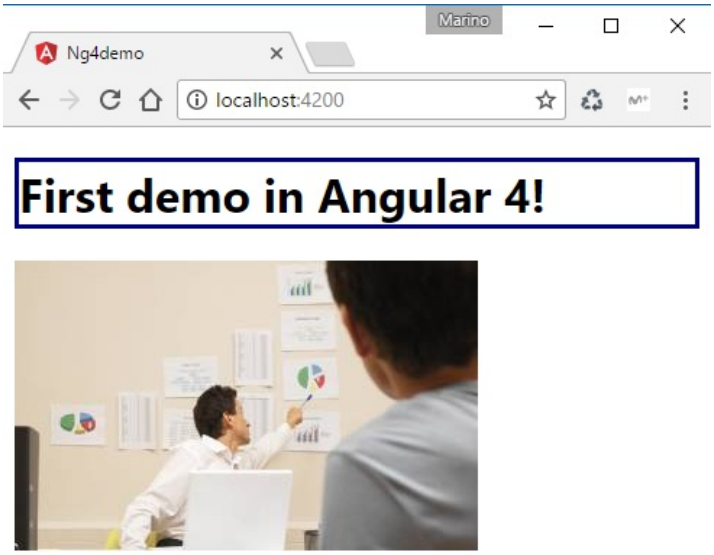
Of course, we can also add any static content to the **`index.html`** itself (an image, for example, next to the selector):

```
<img src="http://lorempixel.com/320/200/business" alt="">
```

The environment will take care of every change in the application and incrementally recompile those changes.

Notice that, at the end, the text we see is the class' `title` property, which is evaluated and managed by the component architecture of the Angular framework, and it is this component architecture that facilitates the implementation of Dependency Injection.

So, we should see those changes without any more intervention:



Some of the procedures implied in this model use types of dependencies that are solved at compile time and runtime. But how can we see the DI in action?

WOW! eBook
www.wowebook.org

DI in Angular 4

Well, DI in Angular follows the construction injection pattern; thus, since we are dealing with classes (components) here, most of the injection is required in the constructor's parameter definitions.

A dedicated page on the official Angular documentation site supplies more information about Dependency Injection at `https://angular.io/docs/ts/latest/guide/dependency-injection.html`.

As with AngularJS, Angular creates an injector object in the bootstrap process, so you don't have to do it by yourself (remember, the `platformBrowserDynamic().bootstrapModule(AppModule)` sentence inside `main.ts`).

WOW! eBook
www.wowebook.org

The concept of a provider

With that in mind, the programmer's responsibility is to register those classes that will be served via DI later on. You can do that, either in the `NgModule`, or inside any `Component`, but, in either case, the way to declare an injectable class is by adding its name to the `provider's` collection of a component or module.

The difference will be that, when declared inside a **`@NgModule`**, those services will be available alongside the entire application. In the remaining cases, it will be limited to the component's hierarchy.

Let's modify the previous example, so you can see this technique in action. Actually, it is quite simple.

The first step is to create a class that provides the injected information required (I'll call it `DIClass`). Let's imagine we want an extra text property and a picture. Since we only need the picture's URL, we'll define two string properties.

But to make the class injectable, we need to mark it as such. We do that by importing the "`Injectable`" definition from `@angular/core` and decorating the class with that annotation. The final aspect of this class' code will be:

```
1  import { Injectable } from "@angular/Core";
2
3  @Injectable()
4  export class DIClass {
5      public DIProperty: string = "This block comes from an injected Class";
6      public DIPicture: string = 'http://lorempixel.com/320/200/business';
7  }
```

Now, we can get rid of the `<img>` element we inserted in `index.html` and pass the task of inserting the image to the component.

Besides that, and only for presentation purposes, I've modified the `app.component.css` code to mark the injected information inside a border, so those fragments become clearly visible.

With the class ready, we now define the scope of this service. Since we only need it for our `AppComponent`, we'll add a new definition of `providers` in the `@Component`

WOW! eBook
www.wowebook.org

decorator's class; this will make our `DIClass` content available inside the component.

Finally, we can modify our class to hold two extra properties (`subtitle` and `fotosource`), whose values are received by injection. In practice, this injected class will often be data access services or any other type of resource that might be needed in more than one application's component.

Since the way those values are received is by declaring the injected class (or values) in the constructor, the final version will be:

```
import { Component } from '@angular/core';
import { DIClass } from "./DIComponent/DIClass";

@Component({
  selector: 'app-root',
  providers: [DIClass],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'First demo in Angular 4!';
  subtitle: string = "";
  fotosource : string = "";
  constructor(injectedValues: DIClass) {
    this.subtitle = injectedValues.DIProperty;
    this.fotosource = injectedValues.DIPicture;
  }
}
```

3 steps for Injection

Now, if you keep the server running and take a look at the new aspect of the page, next to the initial message, we should see those injected elements, formatted according to the CSS rules inside the **`app.component.css`** file:

And that's it. We can inject as many items as needed as long as we have previously declared those values with the **`@Injectable`** decorator; however, if you need more customizable services or data, I recommend reading the documentation's reference I mentioned previously.

WOW! eBook
www.wowebook.org

Wrapping it up

These techniques make your code more maintainable and scalable, and also less dependent on further changes. Since you only have to reference the required services, a change in the implementation of one doesn't necessarily mean a change in the high-level classes that use those services.

Of course, there is much more in DI in Angular 4 than the basic information included in this chapter. By now I hope you have a better understanding of how DI is present in these frameworks, and why it is becoming universally popular implementation design pattern.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we have covered DI in other frameworks, with a special focus on Angular in its two current branches.

First, we've seen the very basics of TypeScript on which Angular 2+ is based and how its capabilities for class definitions and module loading allow the construction of JavaScript-based applications with a more object-oriented, modular approach.

Next, we went through the implementation of DI inside AngularJS (the legacy branch of Angular), still in use in more than 70 percent of Angular projects worldwide.

Finally, we explored the basics of Angular 4 (the latest version available at thetime of writing this), and how the aforementioned object orientation and its component architecture (based on annotations) allow the implementation of Dependency Injection in a very easy way.

In `Chapter 11`, *Best Practices and Other Related Techniques*, we'll cover some of the most common best practices in DI and other related techniques.

WOW! eBook
www.wowebook.org

Best Practices and Other Related Techniques

The last chapter was a very interesting one as we explored Dependency Injection in different frameworks such as TypeScript, ES2015, and Angular 2. This is the last chapter and we are going to discuss the coding and architectural practices proved to be the best while implementing Dependency Injection in the project.

It's obvious that, when you learn something new and try to implement it in your application, excitement might lead you to create a bad architecture. That would eventually lead to a bad practice because, after you code it for a few times unknowingly, it becomes a habit. Suppose, you get a requirement to save user details to the database. You will definitely send the details to the `Service Layer` method which in turn would connect to the `Data Access Layer` method, to save the details. Quickly (if you are new to the application and don't know about the architecture implemented), the first thing you would do is to use the `new` keyword to instantiate the dependencies and directly call the methods. In that way, we can't achieve loosely coupled modules. However, we could have designed the app and the layers in such a way that the consumers won't be able to generate coupling by any means. We will explore these techniques in this chapter.

So, exploring how to deliver your best with the concept you just learned is important for you and your app. For that reason, there are best practices which you should be aware of. Dependency Injection deals with Object Composition, Object Lifetime, and Interception as a whole, which should be ideally managed by DI Containers. If we try to deal with these building blocks ourselves instead of using a container to manage the dependencies, we might end up with a bad design.

Although we can choose one DI Container, designing and refactoring our code accordingly to help the container to work with our classes might be an erroneous task.

In this chapter, we will touch upon these points in detail to understand the real-time problems faced during the induction of DI into the projects. We will learn techniques to avoid bad habits in DI.

WOW! eBook
www.wowebook.org

We will discuss the following topics:

- Best practices and recommendations in Dependency Injection
- Some recommended techniques to remove tight coupling
- Refactoring and implementation of DI in legacy layered applications

WOW! eBook
www.wowebook.org

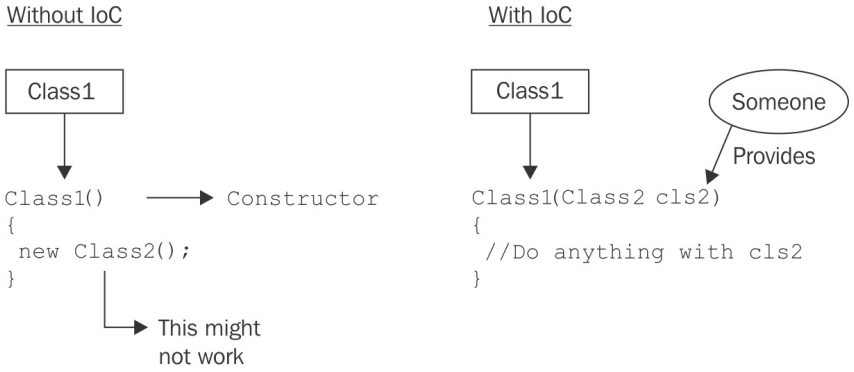
Tightly coupled systems

Till now, we have discussed *DI in .NET Core 2.0, Pillars of DI, DI Patterns, Anti-Patterns, DI in other frameworks*. Someone who is new to the DI concept might do make mistakes and it's normal. We will see more of such scenarios in upcoming sections. Before that, let's analyze something which you might find more often in ASP.NET MVC projects. If you don't, ask your colleagues and friends. You will definitely see such instances.

Let's see one simple controller without DI implemented and learn how to refactor that so that it will respect the principles of *Inversion of Control*.

Inversion of Control states *Do not call us we will call you*. When a class tries to instantiate another class inside it, it basically creates a coupling with the second class. Now the first class creation depends on the second class.

If the second class fails, for any reason, we won't be able to get the instance of the first class, as the constructor will break:

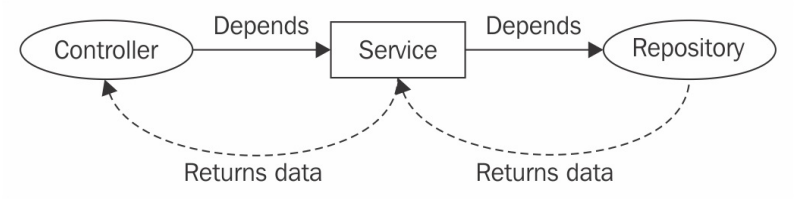


The **Someone** in the picture can be an IoC Container.
I would strongly recommend Mr. Shivprasad Koirala's article, if you want to grasp this concept in depth. Here is the link--https://www.codeproject.com/Articles/29271/Design-pattern-Inversion-of-control-and-Dependency.

WOW! eBook
www.wowebook.org

The problem

The following diagram depicts what we are going to do now. We will have a **Controller** depending on a **Service**, which in turn depends on a **Repository** for data:



First of all, let's think of a simple controller and how that is generally implemented in projects. The following is a simple `UsersController` which has an action method `Edit`:

```
public class UsersController : Controller
{
  public IActionResult Edit(int userId)
  {
    var usersService = new UsersService();
    return View(usersService.GetUser(userId));
  }
}
```

The following are the conclusions (more accurately the problems), which can be made from the preceding code block:

- `UsersController` instantiates `UsersService` using the `new` keyword (creates an instance of a concrete class)
- `UsersController` is tightly coupled with `UsersService`

As far as our knowledge from other chapters is concerned, we should definitely avoid both of the preceding points. This makes Unit Testing difficult. DI principles encourage us to utilize the area of the Composition Root to register all required classes/dependencies (such as `UsersService`) and then use a DI container to resolve the instances instead of using the `new` keyword whenever required.

Next is `UsersService`, which might look something like the following:

```
public class UsersService
{
  private readonly UsersRepository _usersRepository;
  public UsersService()
```

WOW! eBook
www.wowebook.org

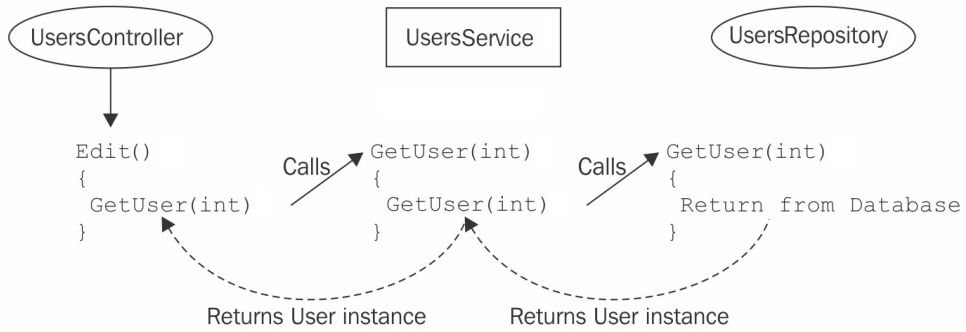
```
    {
      _usersRepository = new UsersRepository();
      // Concrete Class Dependency
    }
    public User GetUser(int userId) => _usersRepository.GetUser(userId);
  }
```

Yet another concrete dependency with `UsersRepository`. `UsersService` now does the same kind of stuff as we saw in the case of the controller. In this example, the service is trying to pass the data returned from the repository. But in reality, it might do a lot of things. The important aspect here is how we are managing our dependencies inside a class which might get uglier as we build a complex system in the future. The exercise we are doing now is just to point out the problem, so don't go by the simplicity of the structure.

The repository would look like the following:

```
public class UsersRepository
{
  public User GetUser(int userId) => new
     DataContext().Users.SingleOrDefault(u => u.UserId == userId);
  // You can just return a demo user like:
  // new User { UserId = 1, UserName = "Tadit" };
}
```

The following diagram explains what has been done so far:



Everything stops here. This is the last in the queue. It gets the data from the `Users` table and sends it back to the client. This architecture works, but there are shortcomings which might get worse as you progress:

- Unit testing is not easy as the classes are tightly coupled with each other. For instance, `UsersController` depends on `UsersService`, which in turn depends upon `UsersRepository`, and then, at last, it deals with the database, while unit testing might fail due to database interactions.
- If this architecture grows and becomes a fully fledged application, then

WOW! eBook
www.wowebook.org

maintenance becomes difficult as the classes are wound up with dependencies all over which will be difficult to identify and manage.

- There is no technique/method by which we can easily modify the dependencies. Suppose we want to implement another class for a service named `UsersNewService` and we want to use that instead of `UsersService`. It will be a tedious task as we have to identify all dependencies of `UsersService` and then change them everywhere.
- Moreover, the concrete service reference might fail in certain circumstances where the method signatures may vary across different classes. For example, the `GetUser` method in both of these service classes may vary in the signature, making the developer's life difficult as you have to modify the `GetUser` caller methods as well, including service reference changes.

These flaws not only describe a poorly architected project but also become a developer's nightmare to refactor and maintain.

WOW! eBook
www.wowebook.org

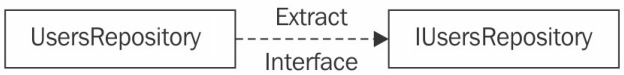
The solution - refactoring with DI

As we all know, the Dependency Inversion Principle states the following:

- High-level modules should not depend on low-level modules. Both should depend on **abstractions**.
- Abstractions should not depend on details. Details should depend on abstractions.

Therefore, let's design the interfaces to abstract out the concrete classes.

Start from the last one in the hierarchy, which is the repository:



The abstraction for the repository would look like the following:

```
public interface IUsersRepository
{
  User GetUser(int userId);
}
```

So, `UsersRepository` would implement `IUsersRepository` as follows:

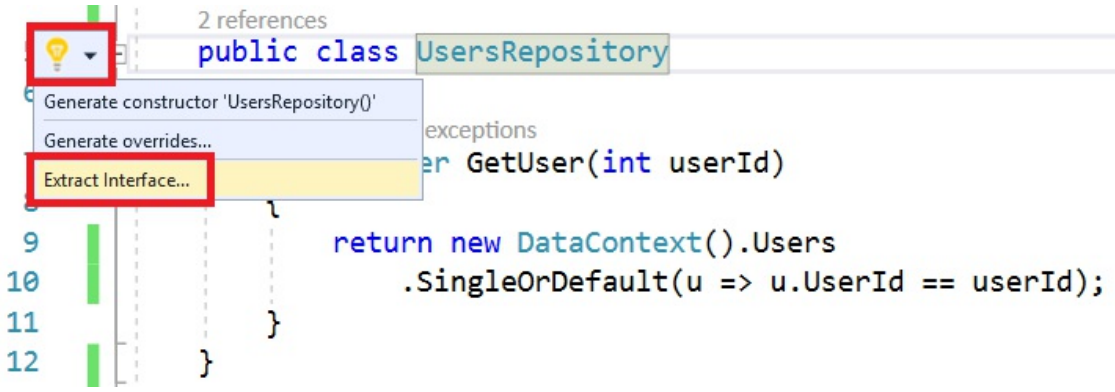
```
public class UsersRepository : IUsersRepository
{
  public User GetUser(int userId)  => new
    DataContext().Users.SingleOrDefault(u => u.UserId == userId);
  // You can just return a demo user like:
  // new User { UserId = 1, UserName = "Tadit" };
}
```

WOW! eBook
www.wowebook.org

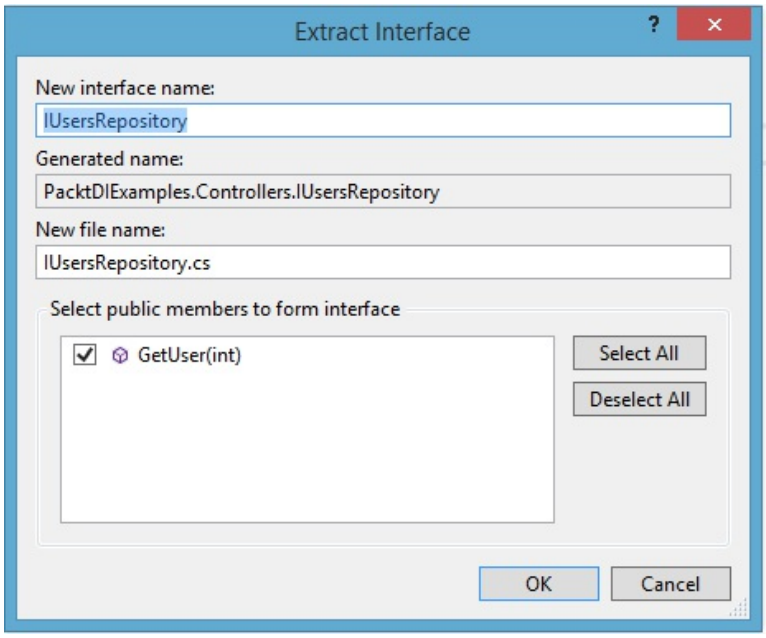
Interface extraction

Do you know how easy it is to extract an interface out of a class? It's very easy and effective with Visual Studio. Let's see the following steps:

1. Hover on the concrete class; you will see a bulb at the left side.
2. Click on that to open a drop-down menu. Refer to the following screenshot:



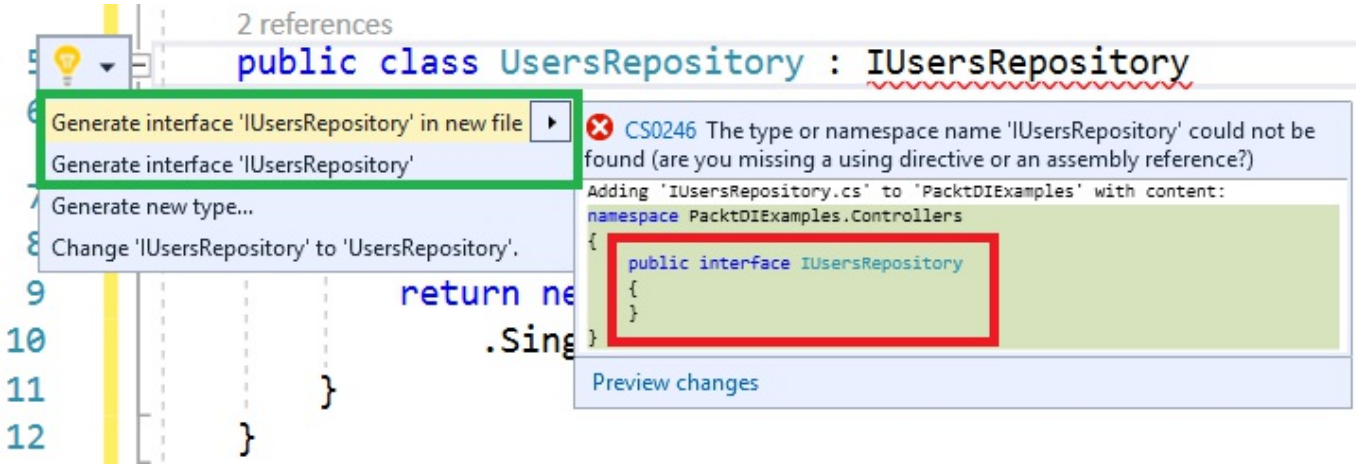
3. Select the `Extract` Interface.
4. A modal pops up which also lists the public members of the class. See the following screenshot:



5. Hit OK. It will create the interface for you.

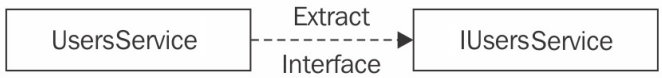
WOW! eBook
www.wowebook.org

There is another way to extract an interface. You can write the interface name and then do a *Ctrl* + . (dot) so that the option bubble appears. You need to select either of the top two options to generate the interface:



However, there is a difference in comparison to the first approach; its marked with the red block. If you select either of the `Generate` options shown in the screenshot, it will create a block for interface with a blank block without any code inside it (indicated inside the red block). However, in the first case, we had the opportunity to select the public members to include them in the interface. Imagine a huge class with lots of public members. Undoubtedly, the first approach wins for obvious reasons as previously described.

Alright. Now we can modify the `UsersService` to inject the repository as a dependency instead of instantiating using the `new` keyword inside the constructor.
The `IUsersService` interface can be designed as we did for the repository in order to use it inside the controller, which we will see in a while:



`IUsersService` is just a simple interface with `GetUser` as a public member:

```
public interface IUsersService
{
  User GetUser(int userId);
}
```

`UsersService` can be modified now to implement `IUsersService`. Injection can be done using the Constructor Injection pattern to inject the `IUserRepository` dependency:

```
public class UsersService : IUsersService // Abstraction
```

WOW! eBook
www.wowebook.org

```
{
  private readonly IUsersRepository _usersRepository;
  public UsersService(IUsersRepository
     usersRepository) // Constructor Injection
  {
    _usersRepository = usersRepository;
  }
   public User GetUser(int userId) => _usersRepository.GetUser(userId);
}
```

This change will trigger a modification to the controller; otherwise the compiler would complain. Let's make the compiler happy:

```
public IActionResult Edit(int userId)
{
var usersService = new UsersService(new UsersRepository());
return View(usersService.GetUser(userId));
}
```

We can now inject/insert the repository instance to the service inside the controller action `Edit`.

So far, we have eliminated one concrete reference from the architecture that is from the `UsersService` class. It's very important to understand our goal. We are actually trying to get every dependency in one place so that the design gets simplified. Thus, with minimal change, we can easily inject different types of dependency into the system.

For example, our `UsersService` is now taking responsibility for managing the `UsersRepository` dependency instead of the service managing it internally with a tight coupling.

However, we are still not done. We have another clearly visible dependency in the controller instantiated using the `new` keyword inside the action `Edit` and that is none other than the `UsersService`.

The refactored injectible `UsersController` would look like the following:

```
public class UsersController : Controller
{
  private readonly IUsersService _usersService;
  public UsersController(IUsersService usersService) //
    Constructor Injection
  {
    _usersService = usersService;
  }

  public IActionResult Edit(int userId)
  {
```

WOW! eBook
www.wowebook.org

```
        // We commented out the following line and used
        // private member _usersService instead.
        //var userService = new UsersService(new UsersRepository());
        return View(_usersService.GetUser(userId));
      }
    }
```

Simple, isn't it? Constructor Injection again comes to our rescue and allows us to inject the service so that we can use it for further operations.

Awesome, now we removed all dependencies. Go ahead and run the app:

> InvalidOperationException: Unable to resolve service for type 'PacktDIExamples.Controllers.UsersService' while attempting to activate 'PacktDIExamples.Controllers.UsersController'.

Unfortunately, it threw an exception. It isn't a surprise. The framework would expect one parameter less constructor inside the controller which we don't have anymore and then it found one parameterized constructor accepting the `IUsersService` implementation instance which was never provided anywhere in the app. Thus, the exception was formulated.

Okay, I know what you are thinking. Adding one parameter less or default constructor won't change the scene. Rather it will show the following in the browser:

> InvalidOperationException: Multiple constructors accepting all given argument types have been found in type 'PacktDIExamples.Controllers.UsersController'. There should only be one applicable constructor.

Moreover, DI does not recommend using multiple constructors, so that is not an option anyway.

Before we go ahead with the solution, let me summarise what we have done so far.

- We created abstractions for concrete classes
- We replaced new keywords with constructors injected with abstractions
- In the process, we also modified the default constructor of `UsersController` to inject the `IUsersService` dependency

However, we received exceptions. That is because we have neither instantiated any implementation of the `IUsersService` interface nor provided it to the constructor.

We need to do two more things to make the app run. That is to let the compiler know what the implementations of the following are:

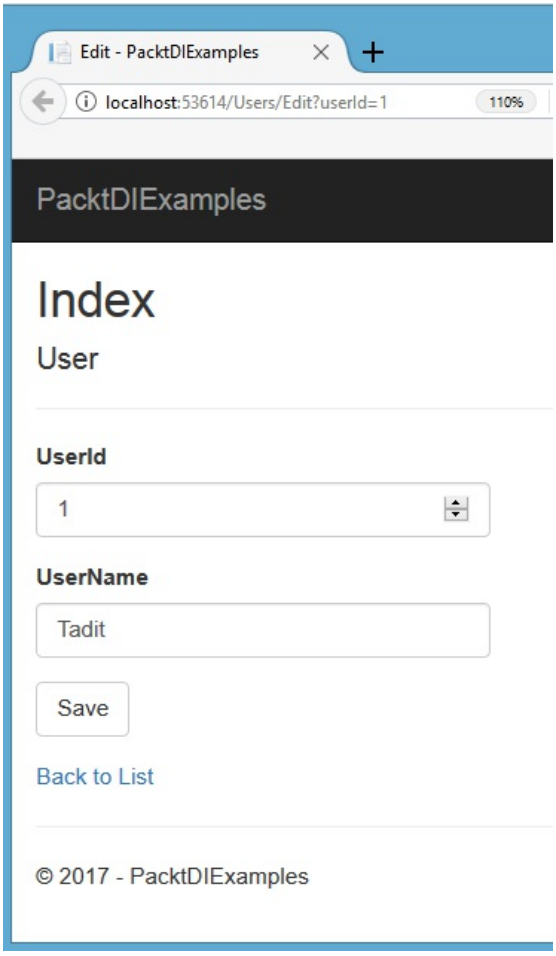
WOW! eBook
www.wowebook.org

- `IUsersService` interface
- `IUsersRepository` interface

That can be done with an IoC Container. As we are on .NET Core 2.0, let's use the built-in container and see what happens. The `ConfigureServices` method of the `Startup` class would look like the following:

```
public void ConfigureServices(IServiceCollection services)
{
services.AddTransient<IUsersService, UsersService>()
        .AddTransient<IUsersRepository, UsersRepository>();
services.AddMvc();
}
```

Let's run the app now:



Finally, it runs. Everything is perfect and we got the output in the browser, thanks to the DI engine of .NET Core and yes, of course, to the progress you have made so far with the book.

We did a fabulous job in refactoring the classes to inject dependencies instead of

WOW! eBook
www.wowebook.org

referring concrete classes which were inefficient and not maintainable. Still, the tight coupling is not totally eliminated. The service is still returning a database model class `User` instance to the controller.

Imagine a three-layered architecture, where the layers can be the `ASP.NET MVC Web App Layer`, `Service/Business Layer`, and `Data Access Layer`. If a developer uses the model classes directly in the presentation layer (`ASP.NET MVC Web App Layer`), that means it is tightly coupled and can't live without a reference to the `Data Access Layer` in the project. In the next section, we will see such a problem and try to refactor it to DI.

WOW! eBook
www.wowebook.org

Layered architecture

In a typical modern ASP.NET MVC web app, you will find a three-layered architecture with isolated components depending on each other as shown in the following diagram:

When we say some layer depends on another, that means the layer that depends, needs instances of the dependency objects to perform certain operations. As we saw in the previous section, to get a user from the database, the controller was asking the service to provide it from the repository. Thus, we can easily separate the classes and package them as components or a class library, speaking technically.

That means my controller remains in `UI Layer` or `Presentation Layer`, `Service Class` make another layer who talks to the database through `Data Access Layer` classes. Now, as we chose Dependency Injection Technique to manage the dependencies, it's the responsibility of the IoC Container to provide the dependent `Data Access Layer` objects to the `Service Layer`, and `Service Layer` objects to the UI Layer. Most importantly, it takes the control from each layer for object creation.

WOW! eBook
www.wowebook.org

The problem - The concrete class reference again

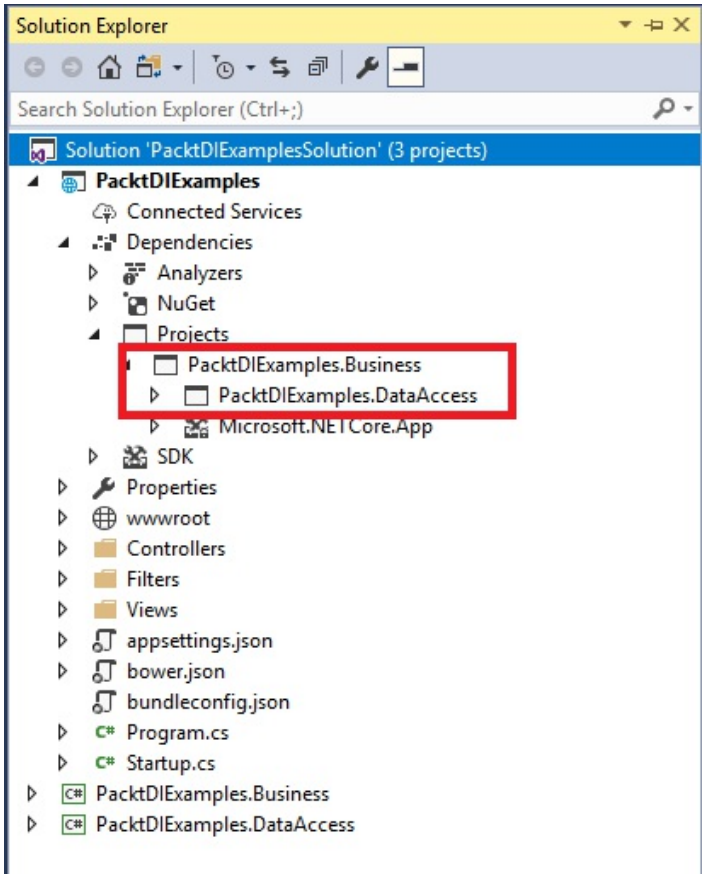
I think you may not have noticed one thing in the last section. We tried to decouple the system as far as we could. But we have not totally achieved a loosely coupled architecture. Let me explain in the next paragraph.

Composition Root is the place where we register all the dependencies to be used by the application. The `ConfigureServices` method in the `Startup` class was the location of this dependency registration as we saw:

```
services.AddTransient<IUsersService, UsersService>()
.AddTransient<IUsersRepository, UsersRepository>();
```

Notice that we are saying, take the `UsersService` instance when `IUserService` dependency is required and the `UsersRepository` instance when `IUsersRepository` is required. So, basically, we now have to get the assembly references of both `Service Layer` and `Data Access Layer` in order to run this code without compilation issues. Check out the following screenshot, which proves what I just said:

WOW! eBook
www.wowebook.org



Whatever classes we have designed so far are just separated out in these three layers. The repository is in the `Data Access Layer` and the service is inside the `Business Layer`.

Clearly, you can see there is a `Business Layer` assembly reference in the UI Layer. As the `Business Layer` has a reference to the `Data Access Layer`, so automatically it is carried over to the UI Layer.

Due to this, the `Presentation Layer` now has access to various implementations of the `Service Layer` and `Data Access Layer`. This is a very bad coding practice as we are trying to violate the layered architecture concept altogether. There is no separation existing between the layers. Moreover, developers can misuse the concrete class implementations, as they have direct access due to the assembly reference. This is not acceptable at all.

WOW! eBook
www.wowebook.org

Misuse of business and Data Access Layer assemblies

Here we will discuss how we, as developers, could misuse the layers in the code by direct reference to the layers. Let's try to understand a few scenarios where these assemblies can be misused in the `UI Layer`:

- **Direct Business Layer Service Access**: As we have a reference of `Business Layer`, so we can instantiate it and then call the `GetUser()` method. We needed a `Repository` instance for the `Service` constructor, which we can also easily provide.

```
public IActionResult Edit(int userId)
{
  UsersService service = new UsersService(new UsersRepository());
  return View(service.GetUser(userId));
}
```

  Notice how we can easily instantiate the `UsersService` class by passing the `Repository` instance to its constructor. That means we still love the `new` keyword.

- **Direct Data Access Repository Access**: Why go through `Service` when we have the `Data Access` reference? Why not directly deal with the Repository to get the data? Cool isn't it!

```
public IActionResult Edit(int userId)
{
  UsersRepository repo = new UsersRepository();
  return View(repo.GetUser(userId));
}
```

  Yet another `new` keyword usage.

- **Direct Data Layer Context Access**: Even better is the following approach where we don't even need these intermediate `Service` or `Repository` instances. We can directly get data from contexts:

```
    public IActionResult Edit(int userId)
    {
      return View(new DataContext()
       .Users
       .SingleOrDefault(u => u.UserId == userId)
```

WOW! eBook
www.wowebook.org

```
            );
        }
```

`DataContext` can provide the data directly inside the Controller Action as shown in the preceding code.

We destroyed the layered architecture and the modules are not independent anymore. Rather the `Business Layer` and `Data Access Layer` are now tightly coupled with the `UI Layer`. This happened just because we had to register concrete implementations for the interfaces with the Service Collections inside the `ConfigureServices` method. For DI injection to take place, we need this registration step so that we can ask the container for dependencies whenever required.

Now, we have to find a way. Let's discover a best practice for this kind of problem.

WOW! eBook
www.wowebook.org

Best practice for layered architecture

Remember, you can do a ton of things to improve the architecture and optimize the performance whilst using DI techniques; however, you can't remove the assemblies from inside the `bin` folder of the ASP.NET MVP web application. You need to understand that the problem lies with access to different concrete implementations of the `Business Layer` and `Data Access Layer` inside the `Presentation Layer`; as we saw how devastating it can be when developers try to play with the layers instead of properly using them.

Therefore, we need to figure out a way to overcome this situation and not allow the `Presentation Layer` to access the concrete classes of other layers while being benefited by Dependency Injection. The solution is MEF. Let's explore more in the next section.

WOW! eBook
www.wowebook.org

Managed Extensibility Framework (MEF)

Before exploring MEF, let me summarize the problem.

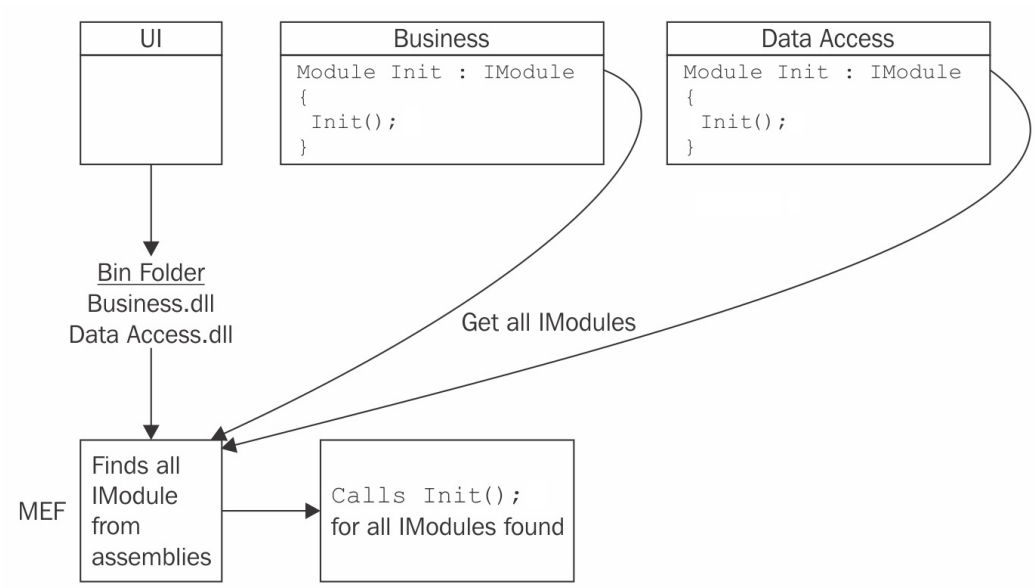
Tight coupling: Concrete classes are referenced inside the `UI Layer` by DI during the `Startup` inside `ConfigureServices` method.

So, basically, we need to do some automation which enables us not to depend upon concrete classes of other layers. That is where MEF comes into the picture.

MEF is a library for creating lightweight extensible applications. With the help of MEF, developers can easily build the an app to automatically bind extensions together without any configurations. There is more on MEF at https://docs.microsoft.com/en-us/dotnet/framework/mef/index.

We are not going deeply into the MEF; you can do that from the preceding link I shared. We just need to understand how we can solve our issue with MEF. For example, we have a `Service` class and a `Repository` class which we don't want to expose to UI for direct consumption as we saw in our preceding code snippets. MEF would do that for us by pulling the classes required to register the dependencies.

The following diagram depicts the role MEF is going to play in our app:



WOW! eBook
www.wowebook.org

You can see we have `ModuleInit` classes in every layer with one `Init()` method and these classes implement a common interface called `IModule`. It's obvious that the DLLs of all the layers will be there in the `bin` folder of the ASP.NET MVC Web Project.

Now the interesting scene. MEF gets all assemblies from the `bin` folder of the App, then it does the following:

1. Searches for the implementations of `IModule`.
2. Fires the `Init()` method of each found module.

In the next section, we will see the codes in different layers and how MEF can bring back loosely coupled modules to optimise the architecture.

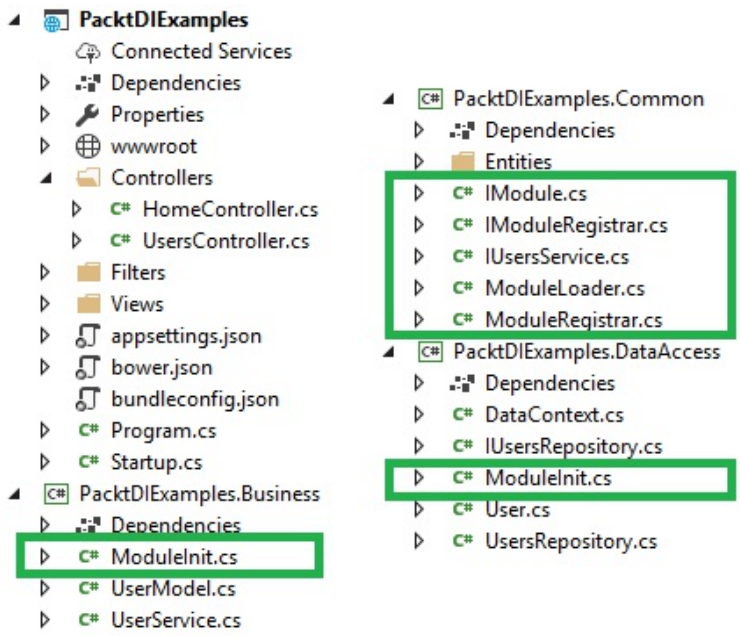
WOW! eBook
www.wowebook.org

Layers and MEF implementation

Before going into the code in some depth, let's first analyze the layers. So, there are three layers.

1. **Web App Layer:** Contains `Controller`, `View`, and so on. Most importantly, it has the entry point `Startup.cs`.
2. **Business/Service Layer:** Contains the `Services` where we can include our business logic.
3. **Data Access Layer:** Contains `Contexts`, `DbSets`, and `Repository` for data retrieval from the database.

Let's have a quick look at the `Visual Studio Solution` window:



We have organized our classes into different layers. Interestingly, we have introduced another layer named as `Common Layer`, where we have placed all common code related to `Business Layer` and `Data Access Layer`. A common project is referenced both by `Business Layer` and `Data Access Layer`.

This common code will help us fetch the dependencies from assemblies related to different layers and package them into one (called `ModuleLoader`) so that the web app can call just one method to initiate everything. Let's understand each part one by one.

WOW! eBook
www.wowebook.org

Introduction of IUser

The goal is to remove the concrete class access from the UI layer. We started to move all common items into one central layer called `Common`. As we moved `IUsersService` to `Common`, we identified a `GetUser` method, which is actually returning a concrete `User` instance. Moreover, `UserService` inside the `Business Layer` also has the same method as it implements `IUsersService`.

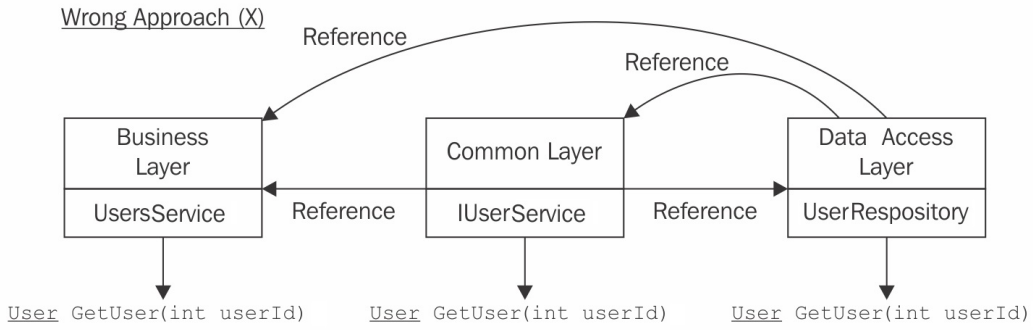
The following code shows how the return type of `GetUser` can be updated to `IUser` type instead of `User` type:

```
public IUser GetUser(int userId) =>
      _usersRepository.GetUser(userId);
// Return type is changed from User to IUser.
```

We need to get rid of this `User` class dependency, otherwise, `Common Layer` has to again refer to the `Data Access Layer` because the `User` resides in that, which is the same case for `Business Layer` to refer `Data Access`. Moreover, `Common` is designed to be referred by other layers and not to get other layers references inside itself. It should be free from project dependencies. This is important.

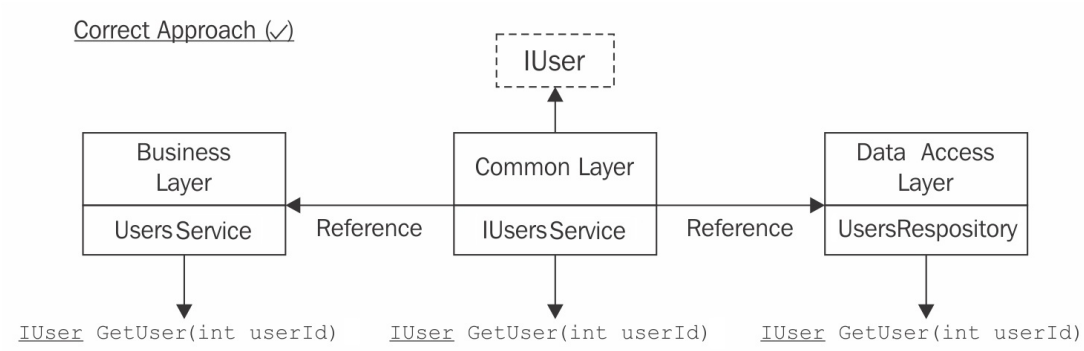
The solution is very easy. One `IUser` interface can be added inside the `Common/Entities/Users` directory, which will be used as a return type for the `GetUser` method. So, `GetUser` inside `UsersService` and `UsersRepository` can return the `IUser` type instead of the `User` instance. Now we can easily remove the coupling of `Data Access` with the `Common Layer`. Also `Data Access` coupling with the `Business Layer`.

The following diagram tells us what was wrong when we used `User` class. We had references to the `Common Layer` and `Data Access Layer` in the `Business Layer`:



WOW! eBook
www.wowebook.org

The following is the approach which is correct with an interface `IUser`, which allows us to reduce the coupling of layers:



Now there is no need for a reference to the `Data Layer` inside the `Common Layer`. On the contrary, the `Business Layer` and `Data Layer` are dependent on `Common`, which is our intention.

WOW! eBook
www.wowebook.org

The IModuleRegistrar interface

The `IModuleRegistrar` interface is responsible for adding the dependency into the service collection. Basically, the implementer of this interface will have an `Add()` method with the signature mentioned, which does the job of the `Add***` (`AddTransient`/`AddScoped`/`AddSingleton`) methods.

```
using Microsoft.Extensions.DependencyInjection;
using System;
namespace PacktDIExamples.Common
{
  public interface IModuleRegistrar
  {
    void Add(Type serviceType, Type implementationType,
       ServiceLifetime lifetime);
  }
}
```

This is present inside the `Common Layer`.

WOW! eBook
www.wowebook.org

The ModuleRegistrar class

`ModuleRegistrar`, which implements the previously mentioned interface, will look something like the following. This basically behaves as a wrapper for the built-in IoC Container to register dependencies:

```
using Microsoft.Extensions.DependencyInjection;
using System;

namespace PacktDIExamples.Common
{
  internal class ModuleRegistrar : IModuleRegistrar
  {
    private readonly IServiceCollection _serviceCollection;
    public ModuleRegistrar(IServiceCollection serviceCollection)
    {
        this._serviceCollection = serviceCollection;
    }

    public void Add(Type serviceType, Type implementationType,
        ServiceLifetime lifetime)
    {
        var descriptor = new ServiceDescriptor(serviceType,
          implementationType, lifetime);
        this._serviceCollection.Add(descriptor);
    }
  }
}
```

Notice the two important aspects of this class:

- `IServiceCollection` is injected into the constructor
- Injected `IServiceCollection` used to add dependencies into the container using `ServiceDescriptor` instance

This class is a part of the Common Library.

Service Descriptor: *`IServiceCollection` is the collection of the service descriptors. A Service Descriptor instance when created can provide full information about a service or dependency. We have different methods for different lifestyles such as `AddTransient`, `AddScoped`, and `AddSingleton` to register dependencies. However, we can't write all three methods to manage the dependencies at a commonplace. That is where a Service Descriptor can help takes lifetime as a parameter and can be added directly to the `ServiceCollection` using the `Add` method.*

WOW! eBook
www.wowebook.org

The IModule interface

`IModule` is the main hero of the architecture because this is the first thing which will be used by a Loader to recognize what modules are required to be fetched from all the assemblies that are targeted. This interface exposes one method, `Initialize()` which is invoked by the Loader to add dependencies into the container. Consider the following code snippet:

```
namespace PacktDIExamples.Common
{
  public interface IModule
  {
    void Initialize(IModuleRegistrar registrar);
  }
}
```

This interface lies inside Common Library as well.

WOW! eBook
www.wowebook.org

The ModuleInit class

Let's see how the preceding `IModule` interface can be implemented inside the `Business Layer`:

```
using Microsoft.Extensions.DependencyInjection;
using PacktDIExamples.Common;
using System.Composition;

namespace PacktDIExamples.Business
{
  [Export(typeof(IModule))] // Will be used by MEF to
     fetch the class.
  public class ModuleInit : IModule
  {
    public void Initialize(IModuleRegistrar registrar) // Registrar
        injected.
    {
       registrar.Add(typeof(IUsersService), typeof(UsersService),
          ServiceLifetime.Transient);
       // Adds the UserService instance
       // to the container with Transient Lifetime.
    }
  }
}
```

`ModuleInit` is present inside both the `Business Layer` and `Data Access Layer`. This class helps us to add different dependencies in each layer such as `UsersService` in `Business` and `UsersRepository` in `Data Access`. You can see how `UserService` is added to the registrar inside `Business Layer`. I am skipping `ModuleInit` of the `DataAccess Layer`. It's just a change of one line inside `Initialize`. The following line of code can be added for `UserRepository` dependency registration inside `ModuleInit` of `Data Access Layer`:

```
registrar.Add(typeof(IUsersRepository), typeof(UsersRepository),
  ServiceLifetime.Transient);
```

Just make a note of the bold part `([Export(typeof(IModule))])` in the last-but-one code block, which is very important. This is the attribute, which helps MEF to grab the `ModuleInit` classes in the specified assemblies, with the interface `IModule`. It can then easily call `Initialize` to start the process of registering dependencies. We will very soon see the code, that does this task for us.

Wait! We have a Constructor Injection here. And what is that used for? Actually, the code for registering the dependency lies inside the registrar, so we definitely need the `IModuleRegistrar` dependency in order to call `Add()` of `Registrar`, which registers the

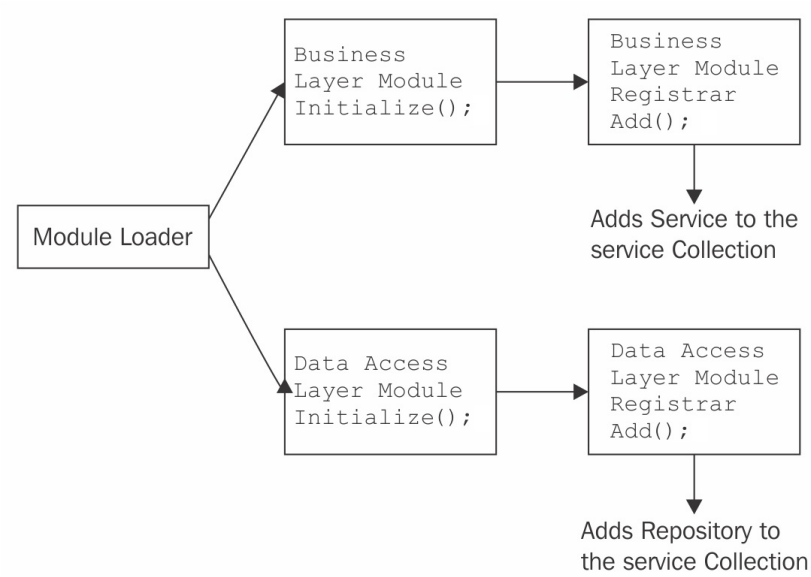
WOW! eBook
www.wowebook.org

required dependency, for example, `UsersService` here inside `Business Layer`.

Check out the flow of controls during the process of dependency registrations. Typically the flow is like the following:

`ModuleLoader` | `ModuleInit(IRegistrar)` | `ModuleRegistrar` | `Add()` (adds the dependency to collection):

The following diagram visualizes the work of **Module Loader** through the available layers as discussed:

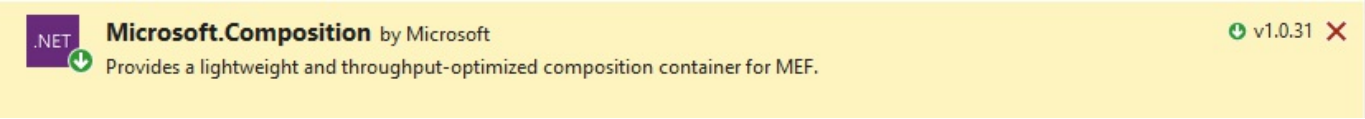


WOW! eBook
www.wowebook.org

The ModuleLoader class

To work with MEF, we need a Nuget package known as `Microsoft.Composition`. This will install `System.Composition` into the project. You need to add the package inside the `Common Layer`. `ContainerConfiguration` is the class present inside the `System.Composition.Hosting` namespace, which composes all assemblies and can provide us with a Container, from which we can easily extract the required concrete implementations.

The following screenshot is the `NuGet` Package preview shown in the NuGet Package Manager window when you search:



Finally the last, but not the least, `ModuleLoader`. All that we have learned till now is managed by the Loader which is a static class with one method `LoadContainer`. This method is invoked by our MVC Web App `Startup` to initialize the dependency registration process. Let's have a sneak peek at the codes:

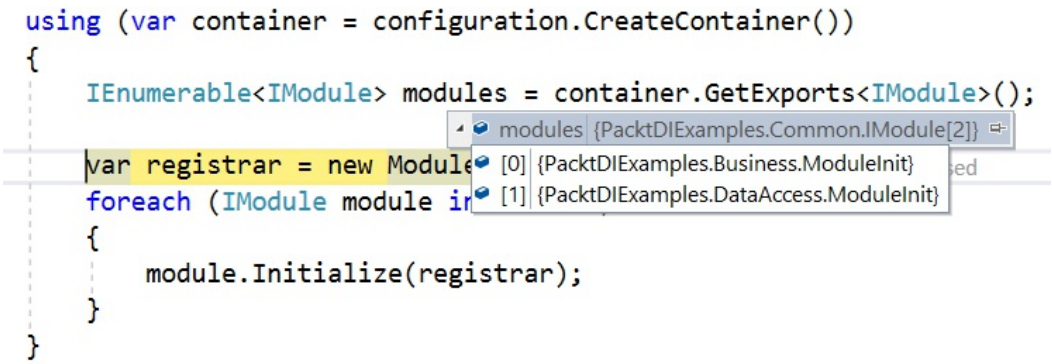
```
namespace PacktDIExamples.Common
{
 public static class ModuleLoader
 {
    public static void LoadContainer(IServiceCollection collection,
        string pattern)
    {
       // Gets the Assembly Location: The total path of the Web App
       assembly, which in our case is
       "C:\\Users\\taditd\\Desktop\\Packt\\Codes\\PacktDIExamples\\
        PacktDIExamples\\bin\\Debug\\netcoreapp2.0\\
        PacktDIExamples.dll".
        var executableLocation = Assembly.GetEntryAssembly().Location;

        // Get all assemblies inside the location with a pattern
          from "bin\Debug\netcoreapp2.0".
        var assemblies = Directory
            .GetFiles(Path.GetDirectoryName(executableLocation),
               pattern, SearchOption.AllDirectories)
            .Select(AssemblyLoadContext.Default.LoadFromAssemblyPath)
            .ToList();

        // Composes the assemblies.
        var configuration = new ContainerConfiguration()
           .WithAssemblies(assemblies);
        using (var container = configuration.CreateContainer())
```

WOW! eBook
www.wowebook.org

```
            {
              // This is important. The following line extracts
             all the IModule implementations from the
             assemblies we fetched.
             IEnumerable<IModule> modules = container.GetExports<IModule&gt;();
             var registrar = new ModuleRegistrar(collection);

             foreach (IModule module in modules)
             {
              // Invoke Initialize for each module with the registrar
               as a dependency.
              module.Initialize(registrar);
             }
           }
         }
       }
     }
```

I have added comments over each step so that it will be easy for you to analyze the steps. The steps can be defined as follows:

1. Get the `Web App Assembly Location`. As we are going to call this method from Web App and every other assembly is placed there, we need that location to find others.
2. Find out, what other assemblies present in that particular location. However, we need to have DLLs of the layers we have in our app. For that reason, a pattern is necessary to recognize the assemblies of the layers. According to our layers, we only need to find all assembly names starting with PacktDIExamples and ending with `.dll`, because the Layer names are like `PacktDIExamples.Business.dll` and `PacktDIExamples.DataAccess.dll`. So, the pattern will be `PacktDIExamples.*.dll`. We will see the code sends this pattern to the `LoadContainer` in a moment. You can see the extracted modules shown in the Visual Studio tooltip while debugging, shown as follows:



3. Create a container to hold all the assemblies using the `ContainerConfiguration` instance.
4. Now `container.GetExports<IModule>()` is used to extract the `IModule`

WOW! eBook
www.wowebook.org

implementation from these assemblies.

5. Execute the `Initialize` method for each `IModule` implementations with a registrar instance as a dependency because the `Registrar` has the actual `Add` method for dependency registration with the `ServiceCollection`.

> *I would advise you to put `try...catch` blocks in this method to handle the exceptions. Due to the space restrictions, I have not posted the `try...catch` block. I just want to show the core logic of a `ModuleLoader`. I have also removed the `usings`, which were taking up a lot of space. You can figure out the `usings`, anyway; if not, use Visual Studio. Hover on the red lines and then follow the steps which will include the required libraries.*

WOW! eBook
www.wowebook.org

Executing the ModuleLoader.LoaderContainer(method from Web App

This is the final step and the simplest one. See how we can call this Loader from the Web App inside the `Startup` as follows:

```
public void ConfigureServices(IServiceCollection services)
{
  // Commented out codes because we now load dependencies from
      another layer.
  // services.AddTransient<IUsersService, UsersService>()
  //         .AddTransient<IUsersRepository, UsersRepository>();

  ModuleLoader.LoadContainer(services, "PacktDIExamples.*.dll");
  services.AddMvc();
}
```

As I mentioned, we need to send a pattern to fetch the Assembly names from `bin/debug/netcoreapp2.0`; otherwise, it will fetch all assemblies, which is unnecessary.

Notice that we are sending the services to the `LoadContainer`. Now the previous code for registering the dependencies is commented, which got optimized by the `ModuleLoader.LoadContainer` call.

WOW! eBook
www.wowebook.org

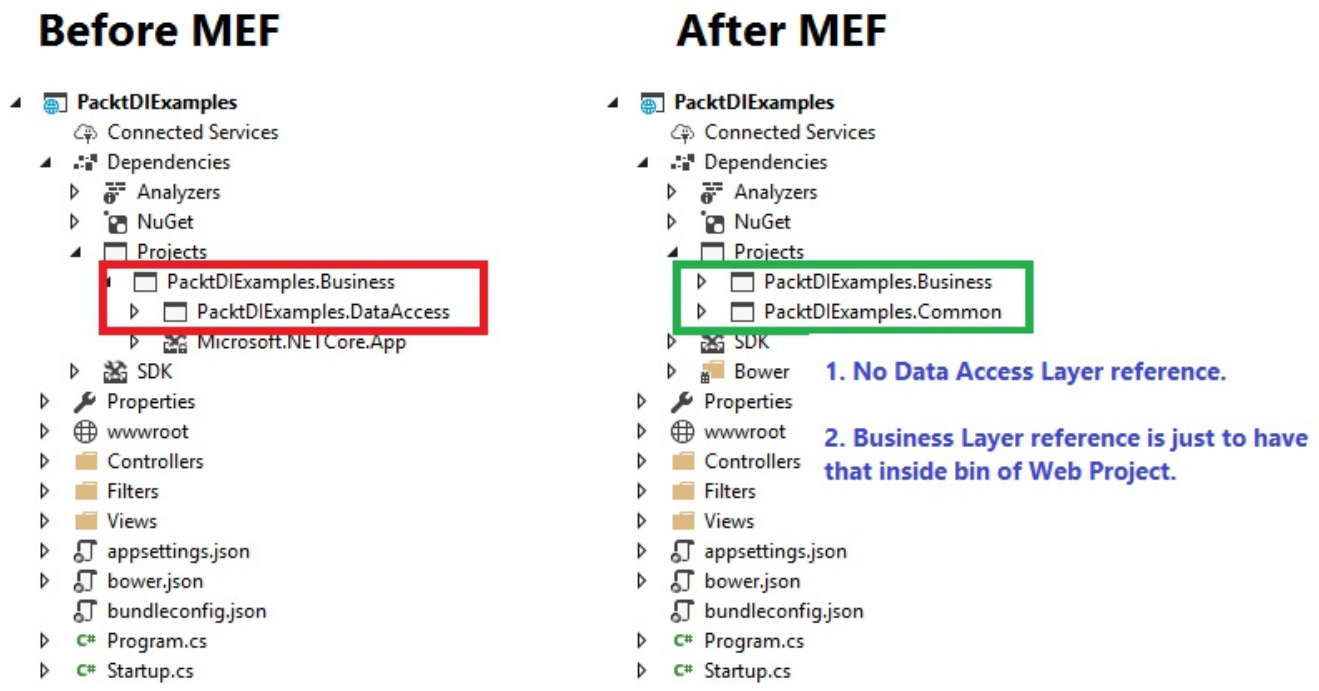
What have we achieved with MEF?

If you have not yet realized the benefits of refactoring the layered architecture with MEF, have a close look at the following screenshot with explanations.

WOW! eBook
www.wowebook.org

Layers are separated

Layers were tightly coupled as the UI was referencing the `Business Layer` and the `Data Access Layer`. Now after MEF, the UI is not tied to the `Data Access Layer` anymore. However, the UI has one `Business Layer` assembly reference as shown in the following screenshot. But that is not used anywhere inside the project. It is there because the assembly should be inside the `bin` folder so that the MEF Module is able to read it while initializing to export the `IModule` implementations. The following is a quick comparison of Project Dependencies before and after MEF:



If you try to remove the Business reference from the UI and run the app, it will throw an exception, as follows. And this is for obvious reasons. Unless you have the reference, MEF won't be able to extract the `IUsersService` implementation and register to the DI Container, simple!

Thus, the issue happens as soon as you hit the controller, as nothing has been registered to the `IServiceCollection`:

WOW! eBook
www.wowebook.org

localhost:53614/Users/Edit?userId=1

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'PacktDIExamples.Common.IUsersService' while attempting to activate 'PacktDIExamples.Controllers.UsersController'.

Microsoft.Extensions.Internal.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, bool isDefaultParameterRequired)

WOW! eBook
www.wowebook.org

All classes inside layers are now internal

Instead of having `public` as the access specifiers for the classes, we can make them `internal` so that they are only accessible inside the assembly (this means only internally inside the Layer):

Business Layer

```
2 references
internal class UsersService : IUsersService
```

Data Access Layer

```
2 references
internal class User : IUser

1 reference
internal class UsersRepository : IUsersRepository
```

>

WOW! eBook
www.wowebook.org

No concrete class instantiation inside the UI Layer

Before MEF implementation, we saw how, inside the `Edit` action method, we could directly reference the `Data Access Layer` and `Business Layer` classes. Now we won't be able to do this because the classes are not accessible anymore.

That is proved from the compiler errors inside `UsersController` as shown in the following screenshot. Notice the tooltip preview, which indicates the unavailability of the UsersService. So, misusing classes of other layers is restricted even if you add the references:

```
public IActionResult Edit(int userId)
{
    UsersService service = new UsersService(new UsersRepository());
    ret
}
```

The type or namespace name 'UsersService' could not be found (are you missing a using directive or an assembly reference?)

Show potential fixes (Alt+Enter or Ctrl+.)

```
0 references | 0 requests | 0 exceptions
public IActionResult Edit(int userId)
{
    UsersRepository repo = new UsersRepository();
    return View(repo.GetUser(userId));
}
```

```
0 references | 0 requests | 0 exceptions
public IActionResult Edit(int userId)
{
    return View(
                new DataContext()
                .Users
                .SingleOrDefault(u => u.UserId == userId)
            );
}
```

WOW! eBook
www.wowebook.org

More layers can be added to the architecture

When, in the future, you add more layers for optimisation or any other business-related requirements, that assembly can be simply referred to by the UI; then every dependency will automatically get registered to the IoC Container. Remember, you need have to a `ModuleInit` class implementing the interface `IModule` to automate the process. The application is totally flexible now, allowing new modules to be attached easily with no hassle.

WOW! eBook
www.wowebook.org

Conclusion

Most modern applications follow a layered architecture. Having a Dependency Injection technique to manage the dependencies required inside the layers is something we should be following. Having said that, we need to be careful with layer designs. That is because we applied DI, thinking that it would resolve issues of tight coupling with the dependencies of a class. However, we did not realize that the layers are getting connected with each other through concrete dependencies (such as the `User` instance return type).

Managed Extensibility Framework provided us with a few easy steps to dynamically bind the dependencies from different layers, package them into one, and then register it to the container for further use by the controllers in the `UI Layer`.

WOW! eBook
www.wowebook.org

Summary

After learning DI Techniques, Patterns, Anti-Patterns and so on, we still needed some pointers to deal with practical scenarios where DI comes into play.

We explored a few real-time application instances where DI can help eliminate coupling and introduce a cleaner dependency structure.

Tightly coupled systems are often seen when a developer tries to be lazy and instantiates the required dependencies directly inside the class using the `new` operator. Keep in mind that new is glue. This not only instills a bad practice in you as a coder but also makes your code untestable. Unit testing becomes difficult. We learned the technique to deal with the problem of tight coupling by following DI techniques.

Then we realized that layering in an application is a common approach with applications. Having DI is definitely going to help us in the long run for layered systems. However, there might be loose ends in the layered approach which should be addressed.

MEF is the technique which we adopted to solve a couple of problems with layered architecture. By implementing MEF, the business or data layer implementations can be marked as internal so that they are not exposed to other layers. Also, layer architectural rules for the separation of concerns can remain intact with a DI container. Thus, the layers can breathe independently without any interference and decorate the architecture in such a way that the introduction of new modules into it becomes very smooth.

WOW! eBook
www.wowebook.org