ASP.net Core to platforma internetowa o cechach:

* open source,
* modular
* **cross-platform**
* cloud optimazed.
* **Fast:** ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages which we need for our application. This reduces the request pipeline and improves the performance and scalability.
* **Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap etc. using Bower (a package manager for the web).
* **Hosting:** ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework.
* **Smaller Deployment Footprint: Jeśli app używa tylko części .NET CoreFX to deployment size będzie mniejszy (mniejszą app webową postawisz, bo core jest lekki, a nie taki obudowana bryła jak .net framework)**

Różnica między .NET Core, a ASP.NET Core - .NET Core to platforma, na której działa aplikacja ASP.NET Core. (.NET Core to szybka, lekka, modularna i otwarta platforma do tworzenia aplikacji internetowych i usług działających w systemach Windows, Linux i Mac.)

Folder główny sieci to wwwroot - tylko te pliki, które znajdują się w nim, mogą być obsługiwane przez żądanie HTTP. Wszystkie inne pliki są zablokowane i nie mogą być domyślnie obsługiwane. W porównaniu z asp.net framewkork pliki statyczne zewnętrzne spoza roota mogły być też obsługiwane.

**Kestrel** jest serwerem używanym do hostowania aplikacji ASP.NET Core na dowolnej platformie.

Metoda z Program.cs CreateDefaultBuilder(args) te konfiguracje ogarnia, jak i IISIntegration.

**Hosting** lokalny strony mamy skonfigurowany automatycznie, ale jeśli chcemy postawić naszą stronę gdzieś w sieci to musimy na Azure, AWS, czy innym raczej płatnym hoscie ogarnąć = tak to wygląda.

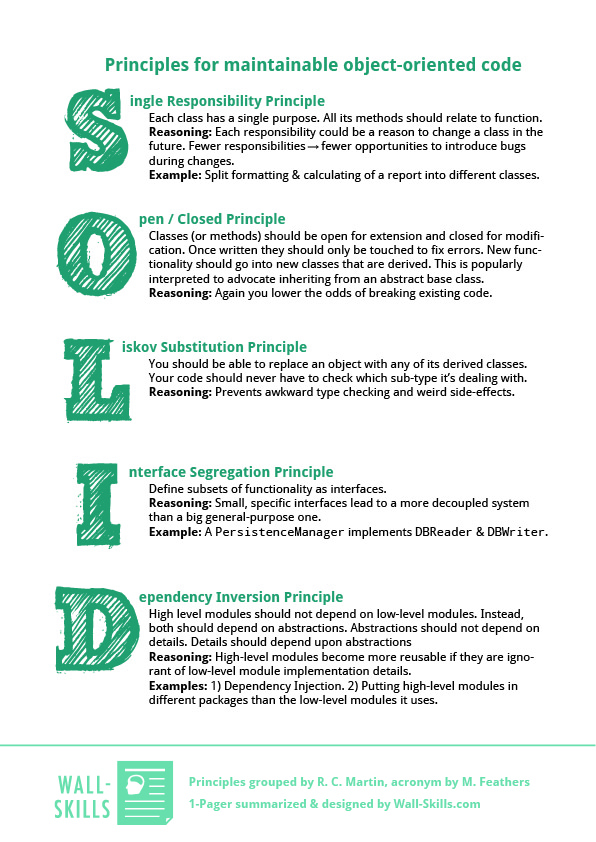
**IIS**

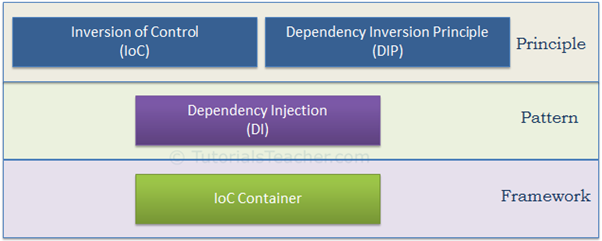
**CLI = comad-line interface** VS używa wewnętrznie tego cross=platformowego narzędzia do tworzeni, przywracania, budowania, running, publikowania .NET apps. VS jako IDE tak inaczej mówiąc/prościej wspiera pracę CLI. Instalowane jest to na naszych urządzeniach/systemach/u mnie na moim Windowsie razem z .NET Core SDK, a ona z VS, więc nom xd. Wystarczy, że wpiszę sobie w cmd zwykłego dotnet i enter i widzę, że mam CLI. Ale kto dziś tworzy tak projekty z cmd?! Warto wiedzieć i zostawić to.

**Design Principe vs Design Pattern**

**Design Principe -** guidelines/Zasady projektowania zapewniają wytyczne dotyczące wysokiego poziomu lub abstrakcji, aby tworzyć lepsze software apps. Nie zawierają wytycznych implementacji i nie zależą od żadnego języka programowania. SOLID (SRP, OCP, LSP, ISP, DIP).

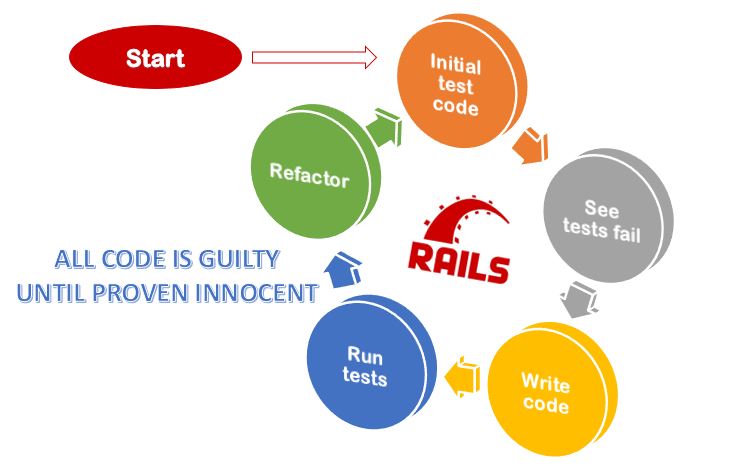
**Design Pattern -** Wzorce projektowe zapewniają rozwiązania niskiego poziomu związane z implementacją. Singleton, Abstract Factory itp.



****

**DI Dependency Injection**

**Chcemy pisać aplikacje TDD = Test Driven Development**

****

Jeśli chcę pisać takie apli (a chcę xd) to muszę znać IoC, bez którego TDD nie może działać.

**IoC** **= Inversion of Control** / Odwrócenie sterowania - w celu osiągnięcia luźno połączonych klas. Sterowanie/kontrola odnosi się do wszelkich dodatkowych obowiązków klasy, oprócz jej głównej odpowiedzialności jaką jest jej przeznaczenie (po co powstała SRP - Single Responsibility Principle).

Tymi dodatkowymi obowiązkami, których kontrola(a raczej odwrócenie jej) dotyczy to:

* przepływ programu od Main() dalej,
* tworzenie zależnego obiektu (od innej klasy)
* wiązanie ich/binding.

Klasa B jest zależnością klasy A, jeśli klasa A zależy od B (ma w sobie zmienną typu B, tworzy jej obiekt i zarządza jego czasem życia). Czyli potrzebuje interfacu pośredniczącego, który będzie kontrolował to tworzenie i życie obietu klasy B w klasie A/zależności klasy A.

public class A

{

B b;

public A()

{

b = Factory.GetObjectOfB ();

}

public void Task1() {

// do something here..

b.SomeMethod();

// do something here..

}

}

public class Factory

{

public static B GetObjectOfB()

{

return new B();

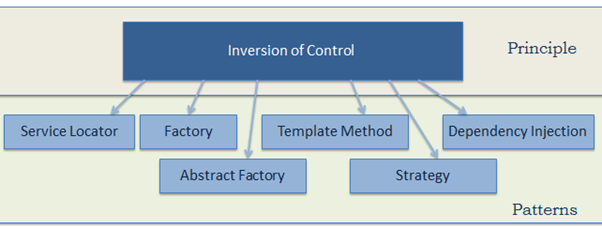
}

}

IoC daje app:

* testable,
* maintainable,
* extensible.

Sposób implementacji IoC dają nam następujące wzorce projektowe:



**DIP = Dependency Inversion Pronciple** - zasada mówi, ze

1. wysokopoziomowe moduły nie powinny zależeć od niskopoziomowych modułów. Oba typy modułów powinny zależeć od abstrakcji.
2. Abstrakcja nie powinna zależeć od szczegółów, szczegóły powinny zależeć od abstrakcji.

public interface ICustomerDataAccess

{

string GetCustomerName(int id);

}

public class CustomerDataAccess: ICustomerDataAccess

{

public CustomerDataAccess() {

}

public string GetCustomerName(int id) {

return "Dummy Customer Name";

}

}

public class DataAccessFactory

{

public static ICustomerDataAccess GetCustomerDataAccessObj()

{

return new CustomerDataAccess();

}

}

public class CustomerBusinessLogic

{

ICustomerDataAccess \_custDataAccess;

public CustomerBusinessLogic()

{

\_custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();

}

public string GetCustomerName(int id)

{

return \_custDataAccess.GetCustomerName(id);

}

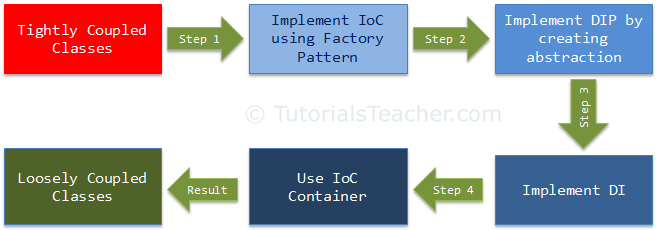
}

Tej zależności od DataAccessFactory chcemy się jeszcze pozbyć

IoC Container - jest frameworkiem do automatycznego implementowania DI.

Gdy tworzymy DI z pomocą IoC to musimy pamiętać o:

* Register/Rejestracja = gdy kontener napotyka jakiś typ to musi on wiedzieć jaka zależność ma zostać utworzona.
* Resolve/Rozwiązanie = częścią tego jest resolution = uchwała, polegająca na tym że IoC kontener za nas tworzy obiekt specyficznego typu, następnie wstrzykuje wymagane zależności (jeśli jakieś są) i zwraca ten obiekt. IoC posiada do tego konkretne metody to resolve the specified type/aby rozwiązać określony typ, czyli zrobić to co napisałam wyżej: utworzyć obiekt, itd.
* Dispose/Usuwanie = IoC kontener musi zarządzać czasem życia zależnych obiektów (różne IoC ma różnych managerów object's lifecycle).

Przykładowe open source containers: ...

Aby faktycznie i poprawnie osiągnąć luźno powiązane ze sobą klasy/luźno połączoną aplikację należy skorzystać ze wszystkiego naraz:

**DI wreszcie xd**

<https://www.tutorialsteacher.com/ioc/dependency-injection>

Wzorzec ten przenosi tworzenie obiektu i wiązanie zależnych obiektów poza zależną od nich klasę.

DI wzorzec zależności obejmuje 3 typy klas:

1. Client Class = klasa zależna (wyższego poziomu) (zależy od Service Class)
2. Service Class = klasa zależności, dostarcza usługi klientowi
3. Injector Class = wstrzykuje obiekt klasy usług/service do zależnej od niego (od service klasy/obiektu jej) klasy klienta

DI może wstrzykiwać zależności na 3 sposoby:

1. przez konstruktor
2. przez propertę
3. przez metodę - z danego interfejsa

Np w Main() wywołując klasę (CustomerService) tworzy i ustawia odpowiednią klasę DataAccess na klasę CustomerBusinessLogic. Kod DI wzorca przez konstruktor:

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

public class CustomerBusinessLogic

{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)

{

\_dataAccess = custDataAccess;

}

public CustomerBusinessLogic()

{

\_dataAccess = new CustomerDataAccess();

}

public string ProcessCustomerData(int id)

{

return \_dataAccess.GetCustomerName(id);

}

}

public interface ICustomerDataAccess

{

string GetCustomerData(int id);

}

public class CustomerDataAccess: ICustomerDataAccess

{

public CustomerDataAccess()

{

}

public string GetCustomerName(int id)

{

//get the customer name from the db in real application

return "Dummy Customer Name";

}

}

Przez propertę:

public class CustomerBusinessLogic

{

public CustomerBusinessLogic()

{

}

public string GetCustomerName(int id)

{

return DataAccess.GetCustomerName(id);

}

public ICustomerDataAccess DataAccess { get; set; }

}

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic();

\_customerBL.DataAccess = new CustomerDataAccess();

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

Poprzez metodę:

interface IDataAccessDependency

{

void SetDependency(ICustomerDataAccess customerDataAccess);

}

public class CustomerBusinessLogic : IDataAccessDependency

{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic()

{

}

public string GetCustomerName(int id)

{

return \_dataAccess.GetCustomerName(id);

}

public void SetDependency(ICustomerDataAccess customerDataAccess)

{

\_dataAccess = customerDataAccess;

}

}

public class CustomerService

{

CustomerBusinessLogic \_customerBL;

public CustomerService()

{

\_customerBL = new CustomerBusinessLogic();

((IDataAccessDependency)\_customerBL).SetDependency(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}