

**ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej**

# Aplikacje webowe na platformę .NET

W08 – Wzorce i techniki (DIP, IoC,DI) , ASP  
.Net - Wzorzec MVC Core

# Syllabus

- Powiązane wzorce projektowe i techniki:
  - Dependency Inversion Principle (DIP) - zasada,
  - Inversion of Control (IoC),
  - Dependency Injection (DI) - technika
    - Przez konstruktor
    - Przez metodę
    - Przez właściwość
- Kontenery wstrzykiwanych zależności
- Wzorec MVC Core
  - `Program.cs`
  - `Startup.cs`
  - Wstrzykiwanie MVC
  - Użycie MVC
- Kontekst danych
  - przez wstrzykiwanie zależności w konstruktorze
  - przez kontener serwisów
- Tworzenie kontrolera
- Tworzenie widoków
  - Podstawy Razora
- Routing:
  - Przez konfigurację
  - Przez atrybuty
- Wstęp do silnika Razor
- Przekazywanie danych tymczasowych do widoków
  - `ViewData`, `ViewBag`
  - `TempData`, `TempBag`
- Bootstrap – informacja wprowadzająca
- jQuery – informacja wprowadzająca
- Dodatek: testowe URL-y

# **TRZY WZORCE PROJEKTOWE: DIP, IOC, DI**

## Ogólne zasady

- To co jest stałe w programowaniu to ... ZMIANY
- Należy tak łączyć komponenty/obiekty, aby wymiana jednego (np. na nową wersję, nową technologię itd.) nie powodowała potrzeby zmiany innych
  - Zmiana może być w danej chwili mała, ale powoduje potrzebę ponownej kompilacji wszystkich zależnych bezpośrednio lub pośrednio klas
- Podział na mniejsze, wymienne komponenty pozwala też na łatwiejsze testowanie:
  - Zamiast używać klas wykonujących pewne operacje trudniejsze do testowania (zapis do bazy, wysyłanie maili, długotrwałe operacje), można podmienić je na klasy-wydmuszki udające wykonywanie tych operacje
- itd.

## Dependency Inversion Principle (DIP)

- Dependency Inversion Principle (DIP), czyli zasada odwracania zależności.
- Zasada odwracania zależności jest wzorcem projektowym, który mówi nam o pisaniu luźno powiązanych klas:
  - moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu
  - abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji
- Wprowadzone przez Roberta C. Martina

## Przykład braku DIP

- Klasa wyższego poziomu zależy od klasy (jej implementacji) niższego poziomu

```
public class LogWriter{
    public void Write(string message)
    {
        Console.WriteLine($"Logger: {message}");
    }
}
public class Device{
    LogWriter logWriter;
    public void Notify(string message) {
        if (logWriter == null)
            logWriter = new LogWriter();
        logWriter.Write(message);
    }
    public void DoSomething() {
        Notify("start of " + nameof(DoSomething));
        // hard work
        Notify("end of " + nameof(DoSomething));
    }
}
```



```
class TestOfUse{
    public static void Test() {
        Device device = new Device();
        device.DoSomething();
    }
}
```

## Inversion of Control (IoC)

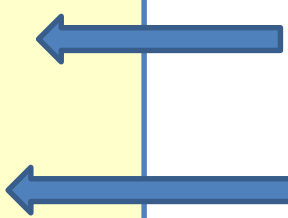
- Inversion of Control (IoC), czyli mechanizm, dzięki któremu moduły wyższego poziomu mogą zależeć od abstrakcji, a nie od konkretnej implementacji modułu niższego poziomu.
- Utworzona musi zostać abstrakcja/interfejs
  - Oraz ewentualne jej implementacje

```
public interface ILogNotification {  
    public void Notify(string message);  
}  
  
public class LogWriter: ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Logger: {message}");  
    }  
}  
  
public class EmailSender : ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Sending email: {message}");  
    }  
}  
  
public class SMSSender : ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Texting: {message}");  
    }  
}
```

## Rozwiązanie poprzez użycie DIP

- Poprawione rozwiązanie, ale nie do końca (zależność nadal istnieje).
  - Nadal klasa wyższego poziomu zależy od klasy niższego poziomu

```
public class Device
{
    private ILogNotification logWriter;
    public void Notify(string message)
    {
        if (logWriter == null)
            logWriter = new LogWriter(); // still here
        logWriter.Notify(message);
    }
    public void DoSomething()
    {
        Notify("start of " + nameof(DoSomething));
        // hard work
        Notify("end of " + nameof(DoSomething));
    }
}
```



```
public class TestOfUse{
    public static void Test(){
        Device device = new Device();
        device.DoSomething();
    }
}
```



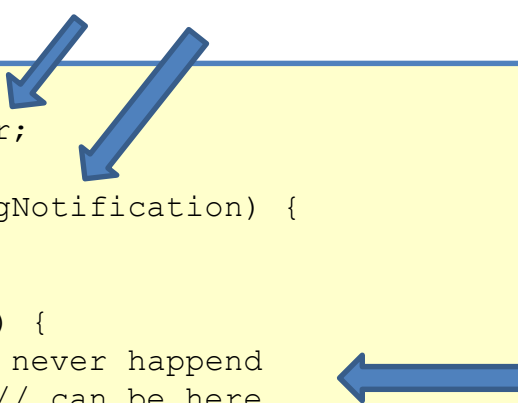
# Dependency Injection (DI)

- Dependency Injection, czyli technika wstrzykiwania zależności, aby całkiem zastosować wzorzec DIP. Są 3 sposoby wstrzykiwania zależności:
  - przez konstruktor
  - przez metodę
  - przez właściwość
- ASP .Net Core ma wbudowaną (poprzez mechanizm odbicia) metodę wstrzykiwania zależności przez konstruktor. Pozostałe można zaimplementować samemu.
- Implementacja interfejsów i klas go implementujących jak w poprzednim rozwiązaniu:

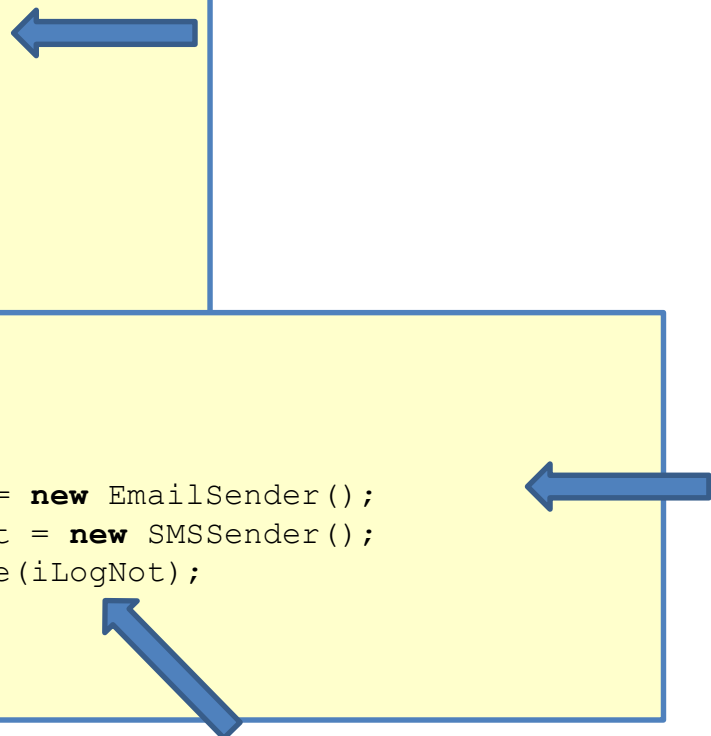
```
public interface ILogNotification {  
    public void Notify(string message);  
}  
  
public class LogWriter: ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Logger: {message}");  
    }  
}  
  
public class EmailSender : ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Sending email: {message}");  
    }  
}  
  
public class SMSSender : ILogNotification {  
    public void Notify(string message)  
    {  
        Console.WriteLine($"Texting: {message}");  
    }  
}
```

## DI przez konstruktor

- Bardzo dobry rozwiązaniem jest wstrzykiwanie zależności przez konstruktor, szczególnie, jeśli wiemy, że nigdy to nie będzie pusta referencja (**null**)



```
public class Device {  
    private ILogNotification logWriter;  
  
    public Device(ILogNotification logNotification) {  
        logWriter = logNotification;  
    }  
    public void Notify(string message) {  
        if (logWriter == null) // maybe never happend  
            logWriter = new LogWriter(); // can be here  
        logWriter.Notify(message);  
    }  
    public void DoSomething() {  
        Notify("start of " + nameof(DoSomething));  
        // hard work  
        Notify("end of " + nameof(DoSomething));  
    }  
}
```

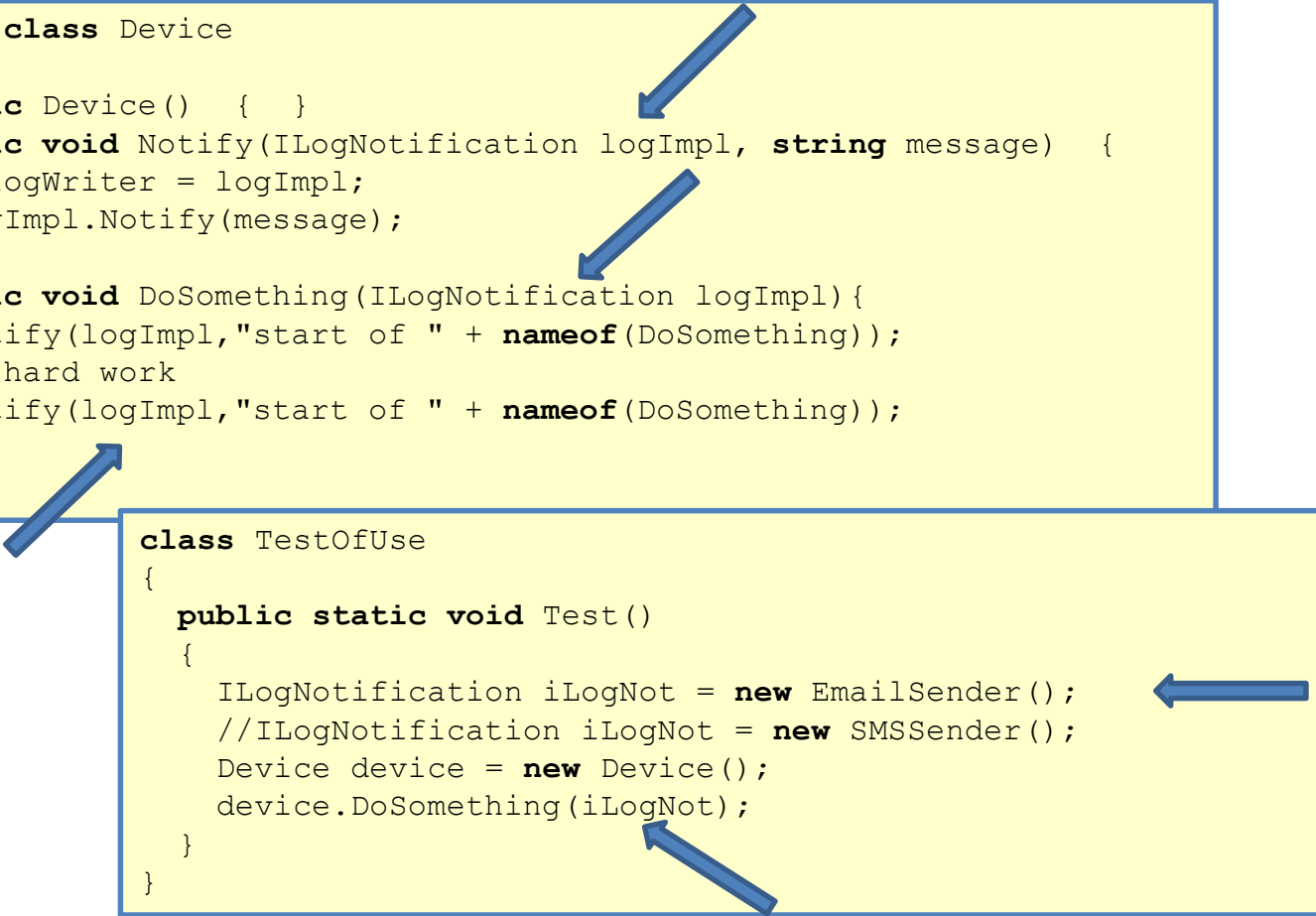


```
class TestOfUse  
{  
    public static void Test()  
    {  
        ILogNotification iLogNot = new EmailSender();  
        //ILogNotification iLogNot = new SMSSender();  
        Device device = new Device(iLogNot);  
        device.DoSomething();  
    }  
}
```

## DI przez metodę

- W przypadku, gdy w trakcie życia obiektu wyższego poziomu będzie potrzeba zmiany implementacji interfejsu obiektów niższego poziomu można w metodach podawać implementację jako **parametr metody**

```
public class Device
{
    public Device() { }
    public void Notify(ILogNotification logImpl, string message) {
        //logWriter = logImpl;
        logImpl.Notify(message);
    }
    public void DoSomething(ILogNotification logImpl){
        Notify(logImpl,"start of " + nameof(DoSomething));
        // hard work
        Notify(logImpl,"start of " + nameof(DoSomething));
    }
}
```



```
class TestOfUse
{
    public static void Test()
    {
        ILogNotification iLogNot = new EmailSender();
        //ILogNotification iLogNot = new SMSSender();
        Device device = new Device();
        device.DoSomething(iLogNot);
    }
}
```

# DI przez właściwość

- Zamiast za każdym razem podawać dodatkowy parametr, lepiej przechować aktualnie wybraną implementację we właściwości:
  - Szczególnie, jeśli dłużej stosujemy daną implementację
  - Kod podobny do pierwszej wersji, ale zamiast pola prywatnego jest właściwość

```
public class Device
{
    public ILogNotification LogWriter { private get; set; }
    public Device() { }
    public void Notify(string message) {
        LogWriter.Notify(message);
    }
    public void DoSomething() {
        Notify("start of " + nameof(DoSomething));
        // hard work
        Notify("end of " + nameof(DoSomething));
    }
}
```

```
class TestOfUse
{
    public static void Test()
    {
        ILogNotification emailSender = new EmailSender();
        ILogNotification smsSender = new SMSSender();
        Device device = new Device();
        device.LogWriter = emailSender;
        device.DoSomething();
        device.LogWriter = smsSender;
        device.DoSomething();
    }
}
```

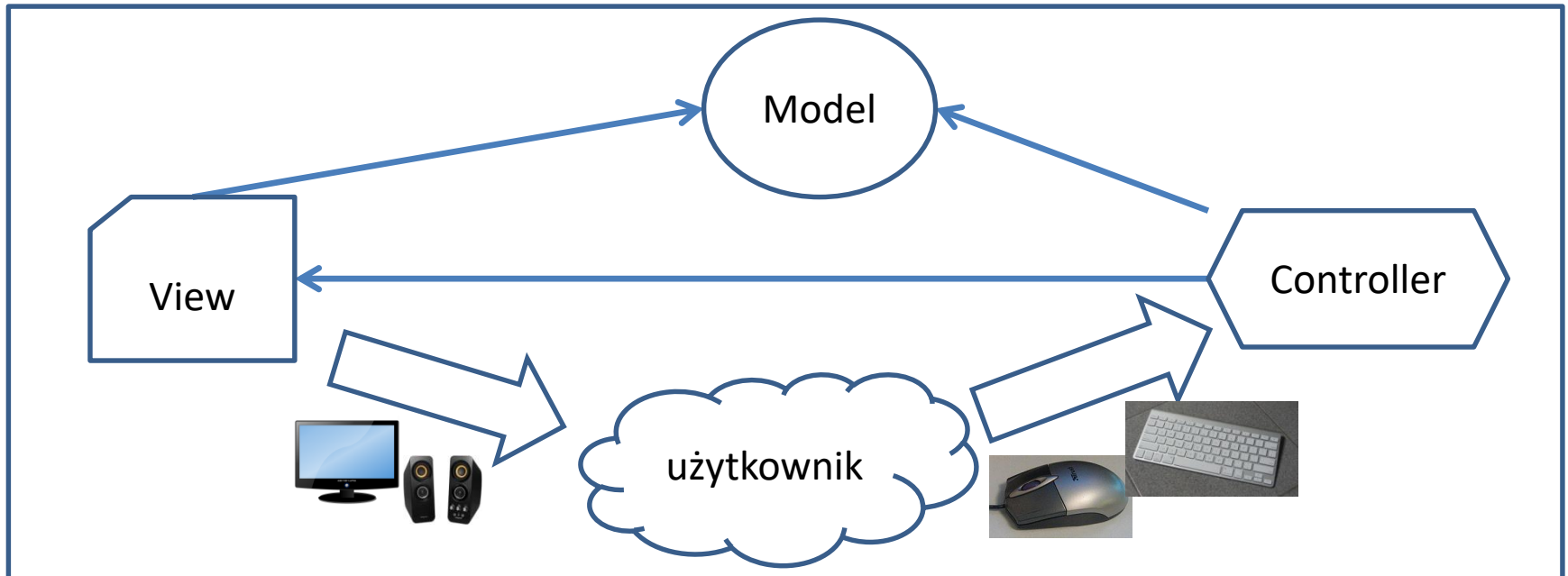
# Kontenery obiektów wstrzykiwanych

- Można łączyć te 3 wersje wstrzykiwania zależności.
- W przypadku wielu (niezmiennych w trakcie życia systemu) obiektów wstrzykiwanych niezłym rozwiązaniem są kontenery, do których na początku programu wstawia się wszystkie elementy (zwane **serwisami**). Z takiego kontenera poprzez **użycie mechanizmu odbicia** można wstawiać do konstruktora odpowiednie elementy.
- Kontenery zawierają **pary: interfejs** (lub klasa, najczęściej klasa abstrakcyjna) serwisu oraz **klasa implementująca** go.
  - Klasa, a nie obiekt, który zostanie **stworzony dopiero, gdy będzie potrzebny**.
  - W związku z powyższym w C# użyte zostaną klasy generyczne parametryzowane tą parą)
- Kontenery te mogą zawierać rozbudowany „świat” takich klas powiązanych ze sobą w konstruktorach.
- Istnieją moduły (do zainstalowania) zawierające implementacje takich kontenerów.
- Kontener serwisów w ASP .Net (klasa implementująca `IServiceCollection`) będzie przykładem takiego kontenera.
- Aplikacja ASP korzystająca z powyższego kontenera nie uruchomi się poprawnie, jeśli w nagłówku konstruktora jest obecny serwis, który nie został dodany do kontenera
  - Wyjątek pojawi się **podczas uruchamiania** serwera

# **ASP .NET – WZORZEC MVC - PODSTAWY**

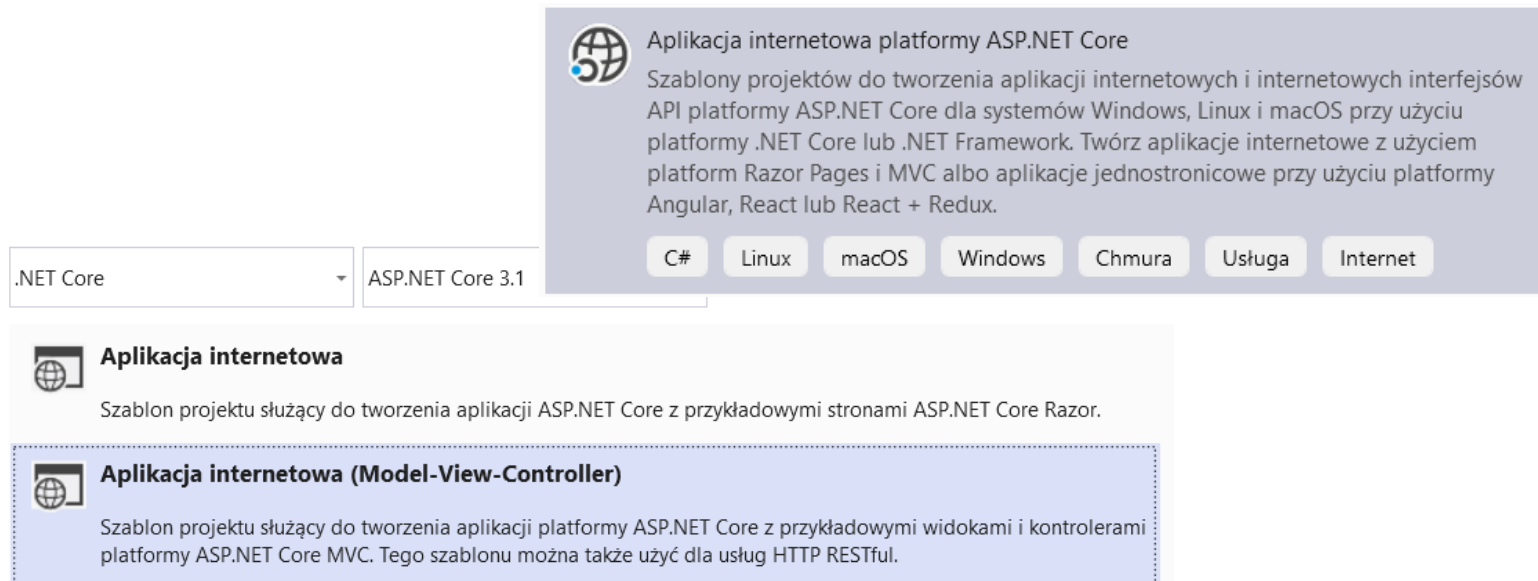
# Wzorzec MVC

- MVC : Model-Widok-Kontroler ( ang. Model-View-Controller)
- Wzorzec stosowany głównie do interfejsu użytkownika
- W zasadzie jest złożeniem kilku wzorców prostych takich jak :Obserwator, Strategia, Kompozyt.
- Model - jest pewną reprezentacją problemu bądź logiki aplikacji (odpowiada za dane)
- Widok - opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika
- Kontroler - przyjmuje dane wejściowe od użytkownika i reaguje na jego poczynania, zarządzając aktualizacje modelu oraz odświeżenie widoków



# Wzorzec MVC – informacje różne

- Poprzedni slajd – klasyczna sytuacja, model statyczny, widok tylko pobiera dane z modelu.
- Odmiany:
  - aktywny model – może zmieniać swój stan niezależnie od działań użytkownika i w zależności od zmiany musi o tym fakcie poinformować kontrolera lub, rzadziej, widok.
  - Widok modyfikuje dane modelu, gdy w modelu są informacje potrzebne tylko do realizacji widoku (podwidoku, innego widoku).
- Siłą tego wzorca jest również w tym, że może być wiele widoków oraz wiele kontrolerów dla jednego modelu. Kontroler decyduje, który widok pokazać/zaktualizować oraz może zdecydować o zmianie kontrolera.
- Może być nawet widocznych wiele widoków (widoków częściowych) oraz wiele kontrolerów do jednego modelu działających jednocześnie.
- W Visual Studio 2019 wybrać projekt „Aplikacja internetowa platformy ASP.NET Core, następnie po nadaniu nazwy wybrać „Aplikacja internetowa (Model-View-Controller)”



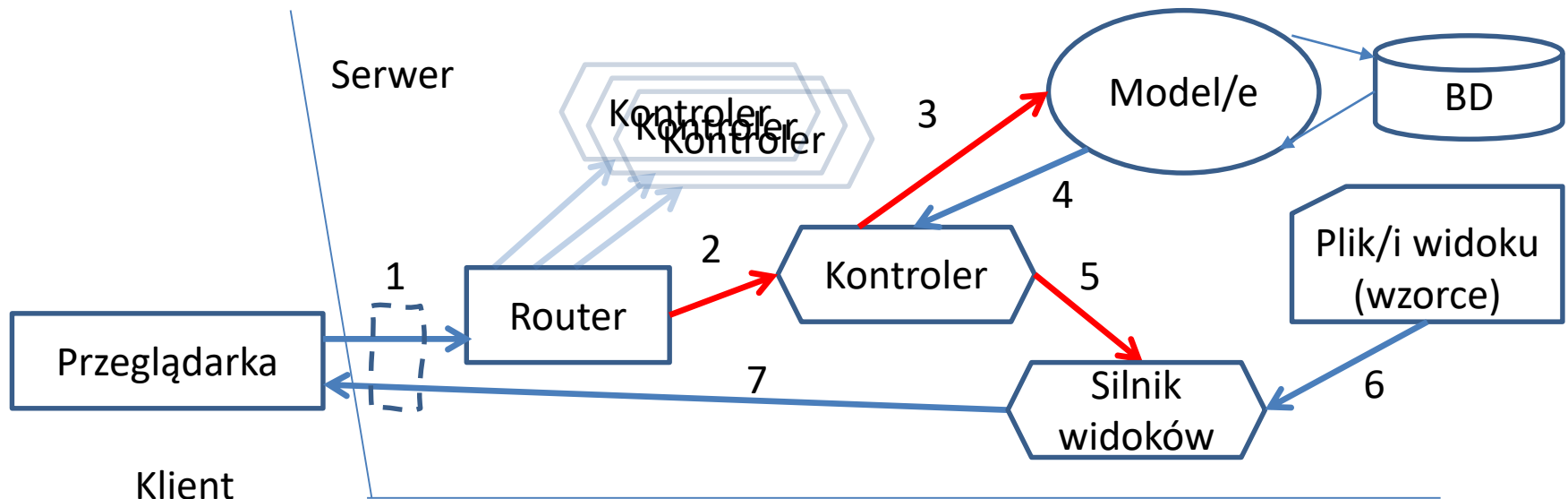


## Konsekwencje użycia MVC

- Zalety:
  - Brak zależności modelu od widoków
  - Łatwiejsza rozbudowa widoków – zmiany interfejsu następują częściej niż zmiany logiki biznesowej.
- Wady:
  - Złożoność: co najmniej 3 klasy dla jednego widoku
  - Kosztowne zmiany modelu: trzeba zmienić wiele/wszystkie widoki, często też kontroler.
  - Trudne testowanie widoków
- Zalety dodatkowe:
  - Wiele środowisk programistycznych, bibliotek języków itp. dostarcza szkielety klas i wspiera model MVC.
  - Znajomość MVC jest często oczekiwana przez pracodawców.

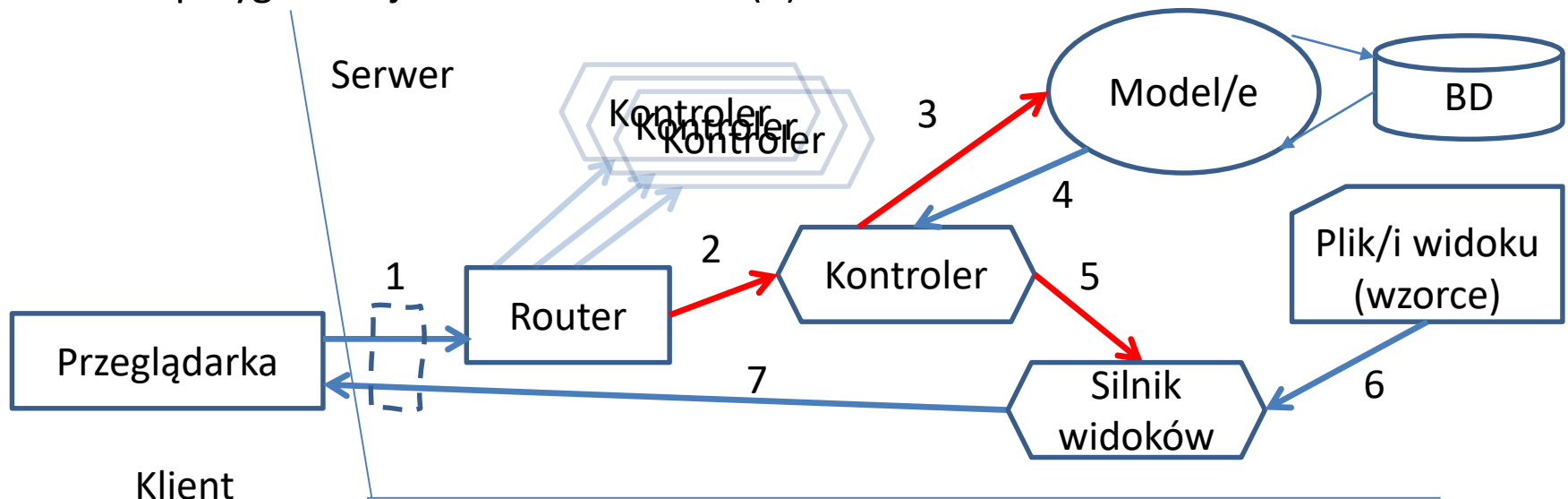
# MVC w kontekście aplikacji webowych

- Przebieg obsługi żądania HTTP w MVC:
  - Żądanie HTTP (1), przetworzone przez serwer, tworzy obiekt `HttpContext` (m. in. z właściwością `Request`) zawierający wszystkie informacje z żądania przetworzone na odpowiednie właściwości (ścieżka URL, parametry zapytania POST/GET/inne, ciasteczka, inne elementy nagłówka lub ciała żądania)
  - W większość przypadków nie obiekt ten nie będzie używany wprost (starsze podejście), ale informacje w nim zawarte będą używane wraz z mechanizmem odbicia do kolejnych kroków.
  - Na drodze (1) działa jeszcze tzw. oprogramowanie pośredniczące, które może zmodyfikować obiekt `HttpContext`.



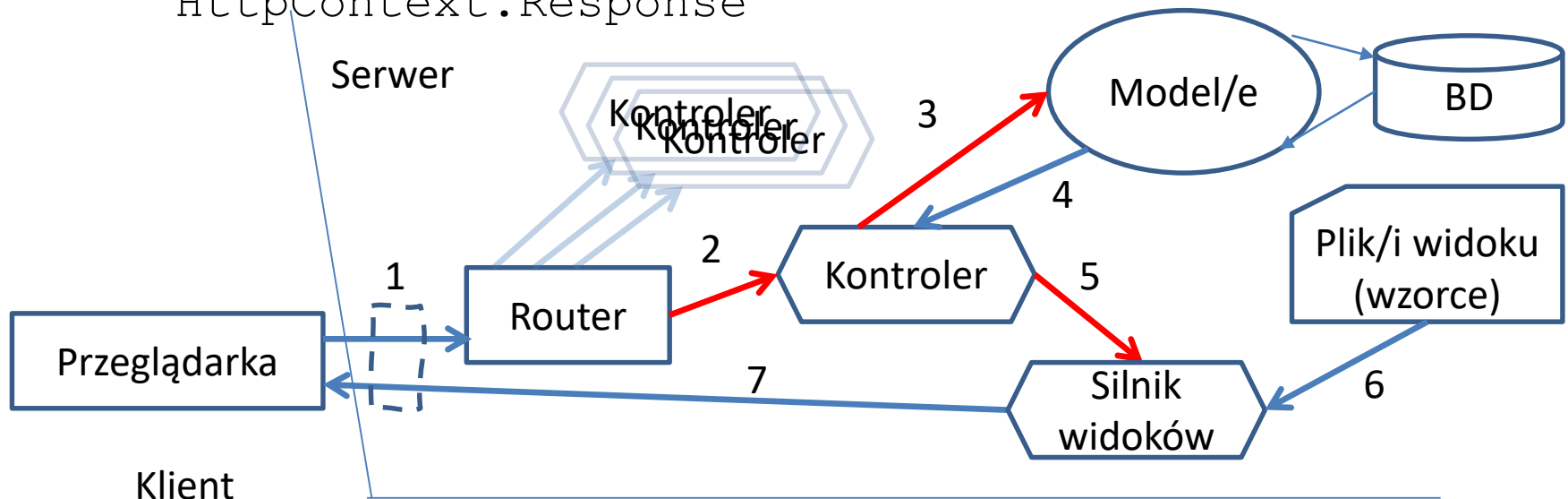
# MVC w kontekście aplikacji webowych

- Przebieg obsługi żądania HTTP w MVC:
  - Ostatecznie żądanie HTTP (1), odbiera router, który na podstawie zapamiętanych zasad określa, który kontroler powinien wykonać którą akcję (metodę).
    - Przekazując również parametry do tej metody
  - Router tworzy (2) obiekt kontrolera i wywołuje akcję.
  - Podczas konstrukcji kontrolera tworzą się (3) potrzebne instancje obiektów modelu a następnie rozpoczyna się wykonanie akcji. Wykonanie akcji przygotowuje dane dla widoku (4).



# MVC w kontekście aplikacji webowych

- Przebieg obsługi żądania HTTP w MVC:
  - Kontroler wybiera odpowiedni widok i przekazuje tę informację oraz dane do wyświetlenia do silnika widoków (5).
  - Silnik formatuje dane (korzystając z plików widoków, 6) i wysyła je użytkownikowi w postaci odpowiedzi HTTP (7), najczęściej jako stronę HTML.
  - Na drodze (7) również może zadziałać oprogramowanie pośredniczące, które może zmodyfikować np. właściwość `HttpContext.Response`

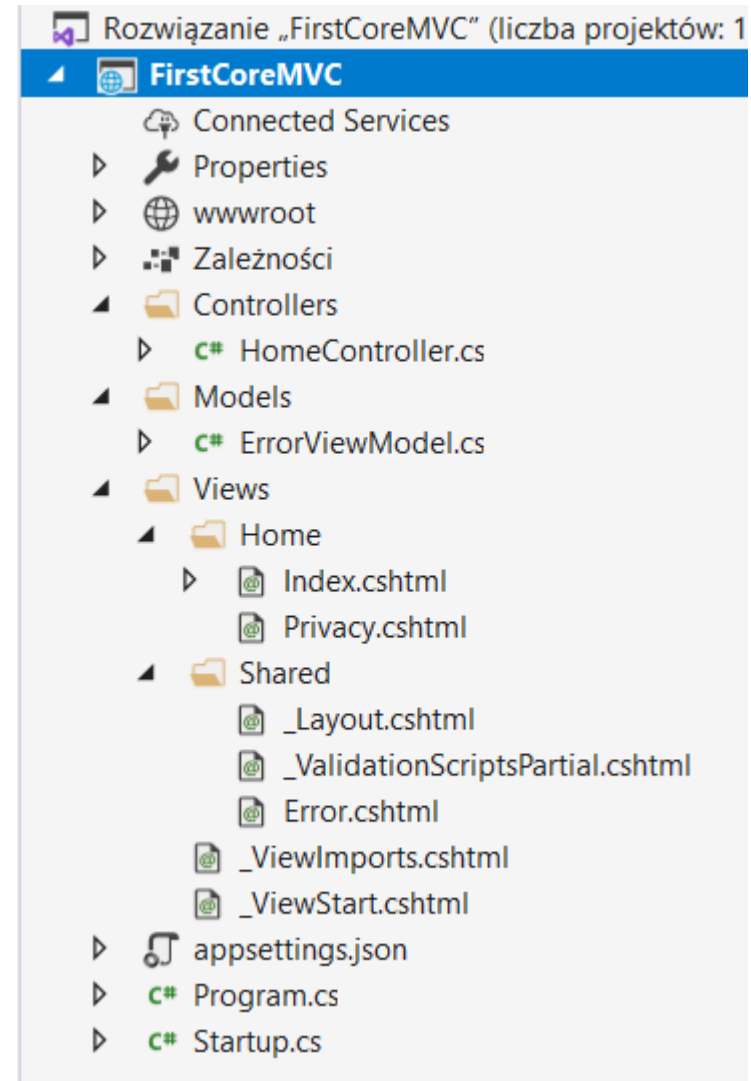


## ASP. NET Core MVC

- Implementacje wzorca MVC firmy Microsoft, a dokładnie „platforma aplikacyjna do budowy aplikacji internetowych opartych na wzorcu Model-View-Controller (MVC) oparta na technologii ASP.NET”.
- Wiele kolejnych wersji Core MVC do obecnej 3.1.1 – styczeń 2020 (Wcześniej .Net Framework 5.2.7 – listopad 2018).
- Platforma Visual Studio oprócz mechanizmów automatycznego budowania szkieletów klas dla wzorca MVC dostarcza wielu innych elementów ułatwiających budowanie aplikacji webowej:
  - Kontener serwisów `IServiceCollection` do wstrzykiwania zależności
  - silnik Razor do budowania widoków.
  - biblioteka Bootstrap do tworzenia widoków estetycznych oraz responsywnych (układ elementów zależy od wielkości widoku, dostępnej rozdzielczości itp.)
  - mechanizm routingu: zamiana adresu URL na wywołanie właściwego kontrolera/widoku.
  - mapowanie bazy danych na kolekcje obiektów i zależności między nimi.
  - i in.

# Założenia ASP. NET MVC Core

- W projekcie VS 2019 typu ASP.NET Core MVC przygotowane są konkretne foldery dla klas typu Model, View i Controller.
- Dla modeli przeznaczony jest folder „Models”, dla widoków – „Views”, dla kontrolerów – „Controllers”.
- Widoki, których jest najczęściej dużo więcej, są dodatkowo poukładane w podfolderach. Nazwy podfolderów pochodzą od nazw kontrolerów. Np. dla kontrolera HomeController (kod klasy znajduje się w pliku HomeController.cs) jest przygotowany folder Views/Home, w którym są widoki dla niego.
- Kontrolery powinny nazywać się według schematu <nazwaWłaściwa>Controller, np. HomeController.
- Kontrolery dziedziczą po klasie Microsoft.AspNetCore.Mvc.Controller
- Folder wwwroot jest korzeniem struktury serwera WWW i zawiera jego statyczne elementy.



## Program.cs

- Za tworzenie aplikacji odpowiada Program.cs
  - Tworzy (CreateHostBuilder(args).Build()) i uruchamia (.Run()) serwer
  - Podczas tego procesu uruchamia metody konfiguracyjne klasy Startup z pliku Startup.cs
  - Kod tworzony domyślnie (jak poniżej) przygotowuje szkielet przebiegu obsługi żądania w sposób jak zostało opisane wcześniej.

Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

## Startup.cs - 1/2

- Plik `Startup.cs` konfiguruje różne elementy wcześniej omawianego procesu przetwarzania żądania HTTP.
  - Przy projekcie MVC Core – zawartość jak poniżej
- Pierwszą uruchamianą metodą jest `ConfigureServices()`.
- Poprzez metodę rozszerzającą `AddControllersWithViews` dodajemy do projektu możliwość używania wzorca MVC
- Poprzez kolekcję serwisów `IServiceCollection` będą wstrzykiwane inne klasy przydatne kontrolerom i i innym klasom.
  - Klasy kontrolerów są serwisami, które są tworzone i uruchamiane poprzez kontener serwisów z użyciem mechanizmu refleksji

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        //services.AddMvc();
    }
}
```

Startup.cs



## Startup.cs - 2/2

- W metodzie `Configure()` ustawiamy kolejne elementy działania naszego serwera: reakcje na błędy zależnie od typu kompilacji, użycie https, użycie plików statycznych (z `wwwroot`), routingu, autoryzacji i podstawowa zasada routingu (`app.UseEndpoints()`).
- **Argumenty** dla tej metody są **wstrzykiwane** z ww. kontenera serwisów.

```
// This method gets called by the runtime. Use this method
// to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        // The default HSTS value is 30 days. You may want to change this
        //for production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Startup.cs

## Różne zestawy serwisów

- `AddMvcCore()` – minimalny zestaw, aby działał serwer i odbierał żądania. Ale brakuje np. walidacji modelu poprzez atrybuty (adnotacje), brak autoryzacji itp. Można łatwo dodać dalsze elementy zestawu poprzez kropkę:

```
services.AddMvcCore()  
    .AddDataAnnotations() // for model validation  
    .AddApiExplorer(); // for Swagger
```

- `AddControllers()` – zawiera to co `AddMvcCore()` oraz obsługę: autoryzację serwisów, API explorer, adnotacje danych, mapowanie formatera, CORS (Cross-origin resource sharing)
  - Nie tworzy widoków (np. w Razorze). Głównie do back-endu

## Różne zestawy serwisów

- `AddControllersWithViews()` – to co `AddControllers()` plus rejestruje silnik widoków Razor oraz TagHelper-y
- `AddRazorPages()` – to co `AddMvcCore()` oraz umożliwia programowanie stron (Page) w Razorze.
  - Trochę inne podejście niż MVC, oczywiście można dodać kolejne możliwości np.:

```
// ready for Razor Pages development
// ready for API development
services.AddRazorPages().AddControllers();
```

- `AddMvc()` – połączenie `AddControllersWithViews()` oraz `AddRazorPages()`
- Źródło: <https://www.strathweb.com/2020/02/asp-net-core-mvc-3-x-addmvc-addmvccore-addcontrollers-and-other-bootstrapping-approaches/>

# Metody rozszerzające

- Zdecydowana większość metod w ramach kodu klasy `Startup` to **metody rozszerzające**.
  - Można to sprawdzić w VS 2019 ustawiając kursor myszki nad nazwą metody.

```
{
    services.AddControllersWithViews();
    //services.AddMvc()
}
```

(extension) `IMvcBuilder IServiceCollection.AddControllersWithViews()` (+ 1 overload)  
Adds services for controllers to the specified `IServiceCollection`. This method will not register services u  
This method configures the MVC services for the commonly used features with controllers with views. T

```
    }
    app.UseHttpsRedirection();
    app.Use
    app.Use
```

(extension) `IApplicationBuilder IApplicationBuilder.UseHttpsRedirection()`  
Adds middleware for redirecting HTTP Requests to HTTPS.

```
    app.UseStaticFiles();
    app.UseRouti
```

(extension) `IApplicationBuilder IApplicationBuilder.UseStaticFiles()` (+ 2 overloads)  
Enables static file serving for the current request path

```
    app.UseRouting();
    app.UseAut
    app.UseEnd
```

(extension) `IApplicationBuilder IApplicationBuilder.UseRouting()`  
Adds a `Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware` middleware to the specified `IApplicationBuilder`.  
A call to `EndpointRoutingApplicationBuilderExtensions.UseRouting(IApplicationBuilder)` must be followed by a call to `EndpointRoutingApplicationBuilderExtensions.UseEndpoints(IApplicationBuilder, Action<Microsoft.AspNetCore.Routin`

# Metody kontrolera

- W klasie kontrolera muszą zostać zdefiniowane publiczne metody zwracające `ActionResult`.
- Dla interfejsu `ActionResult` istnieje kilka klas w bibliotece, które go implementują np.: `ViewResult` (wzraca stronę WWW), `RedirectResult` (przekierowuje na inną akcję na podstawie adresu URL), `JsonResult` itp.
- Klasa `Controller` posiada metody do tworzenia odpowiednich wyników akcji. Nazwy tych metod są jak w/w klas bez ostatniego członu `Result`, czyli np. metoda `View()` zwraca typ `ViewResult`.
- Jeśli wywołamy np. metodę `View()` bez parametru, to domyślnie (poprzez mechanizm odbicia) pobierze nazwę metody, z której została wywołana. Np.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(); // return View("Index");
    }
}
```

...

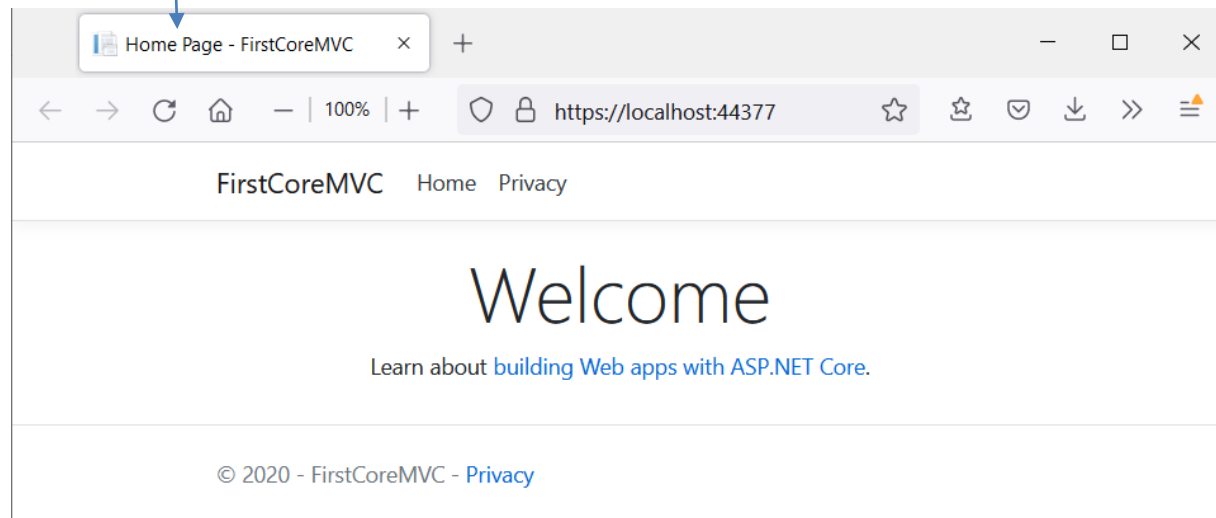
Controllers/HomeController.cs

- Spowoduje to przetworzenie i wysłanie strony WWW na podstawie pliku `/Views/Home/Index.cshtml`
- W domyślnym projekcie wytworzonym w środowisku VS można to przetestować po uruchomieniu projektu i wpisaniu w przeglądarkę adresu `http://localhost:12345/Home/Index` (zamiast 12345 może być inny numer portu).

# Przykład działania

```
@{  
    ViewData["Title"] = "Home Page";  
}  
  
<div class="text-center">  
    <h1 class="display-4">Welcome</h1>  
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET  
Core</a>.</p>  
</div>
```

View/Home/Index.cshtml



## Stare, dobre(?) czasy

- Na początku adres URL oznaczał w zasadzie adres pliku w odpowiedniej kartotece zamapowanej na początek adresu URL

sun10.pwr.edu.pl/~koniecz/mac/macierz.html

sun10.pwr.edu.pl  
/users/staff/koniecz/wwwroot/mac/macierz.html

- Obecnie adres url w żądaniu podlega bardziej zaawansowanej obróbce, natomiast plik nie musi być w folderze, tylko zostanie wytworzony w odpowiedzi na żądanie.

# Routing

- To, że adres <http://localhost:12345/Home/Index> spowodował, że uruchomił się kontroler `HomeController`, a w nim metoda `Index()` nie jest regułą bezwzględną. Reguły routingu (zamiany adresu URL na wywołanie konkretnej akcji konkretnego kontrolera) są zapisane w pliku `/Startup.cs`. Początkowy routing dodany w tym pliku wygląda następująco:

```
app.UseRouting(); // uruchomienie zasad routingu

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
}
```

`Startup.cs`

- Najważniejsza część to wywołanie metody `MapControllerRoute` opisującej wzorzec URL i ewentualnie domyślne wartości (np. `controller=Home`). Nazwa kontrolera jest bez końcówki `Controller`, czyli dla `HomeController` nazwa będzie „Home”.
- Znak zapytania (?) oznacza wartość opcjonalną, która może nie wystąpić w adresie URL
- Dla reguły „default” oznacza to możliwość użycia równoważnych adresów:
  - <https://localhost:44377/Home/Index>
  - <https://localhost:44377/Home/>
  - <https://localhost:44377/>
  - <https://localhost:44377/Home/Index/5>
- Ale już nie:
  - <https://localhost:44377/Home/Index/5/4>



## Własne widoki i akcje

- Dodajmy własną nową stronę, dodatkowe akcje w kontrolerze Home oraz dodatkowe własne metody routingu.

View/Home/MyPage.cshtml

```
@{
    ViewBag.Title = "My page";
}
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>

<p>My web page.</p>
<p>@ViewBag.ValueX </p> @ViewBag.ValueText
```

Controller/HomeController.cs

```
// ...
public IActionResult MyPage()
{
    ViewBag.Message = "My first page in MVC";
    ViewBag.ValueX = 1234;
    ViewBag.ValueText = "text value";
    return View();
}
public IActionResult MyPage2(int number, string name, string other)
{
    ViewBag.Message = "Parametric page number=" + number + " name=" +
        name + " , other=" + other;
    return View("MyPage");
}
```

## Dodanie reguł routingu

- Z wielu względów możemy chcieć sami ustalić inne reguły routingu. W tym celu w wywołaniu metody `app.UseEndpoints` należy dodać nową regułę z nazwą, wzorcem URL (ale przed regułą ogólną!)

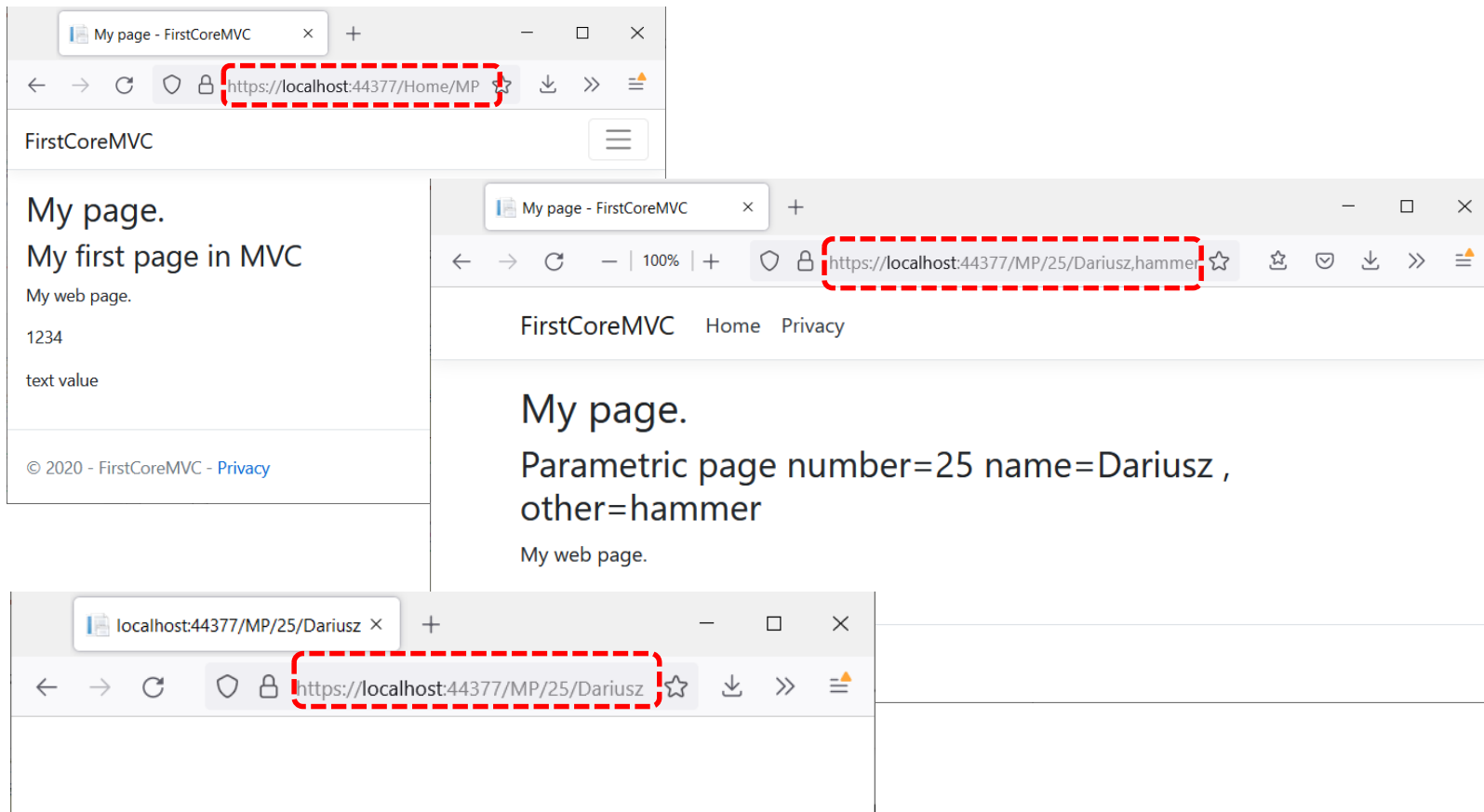
Startup.cs

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "MP",
        pattern: "Home/MP",
        defaults: new { controller = "Home", action = "MyPage" });

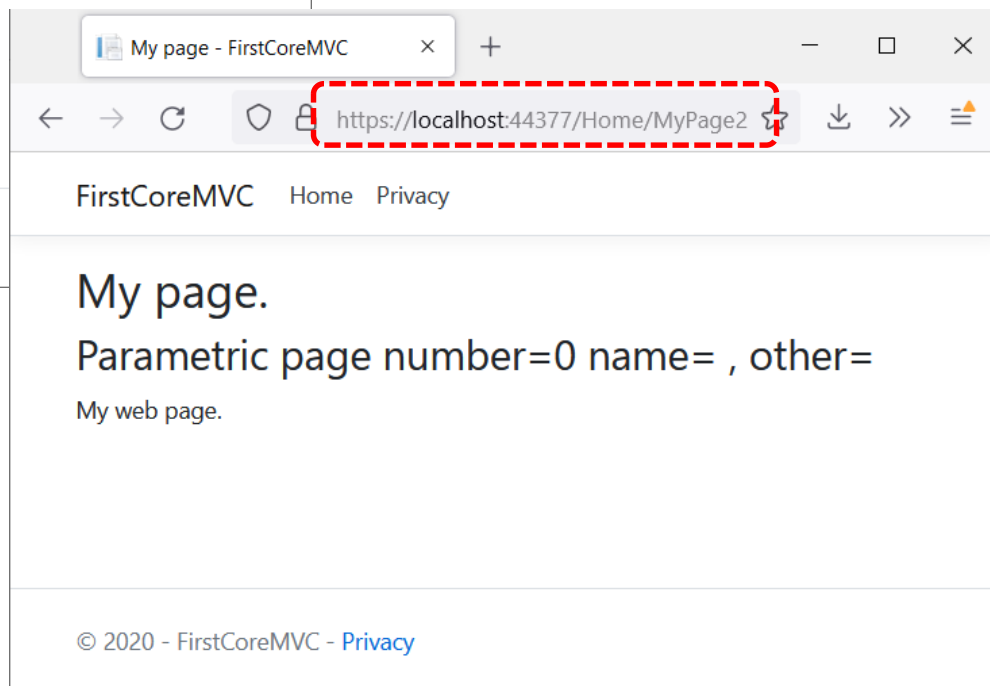
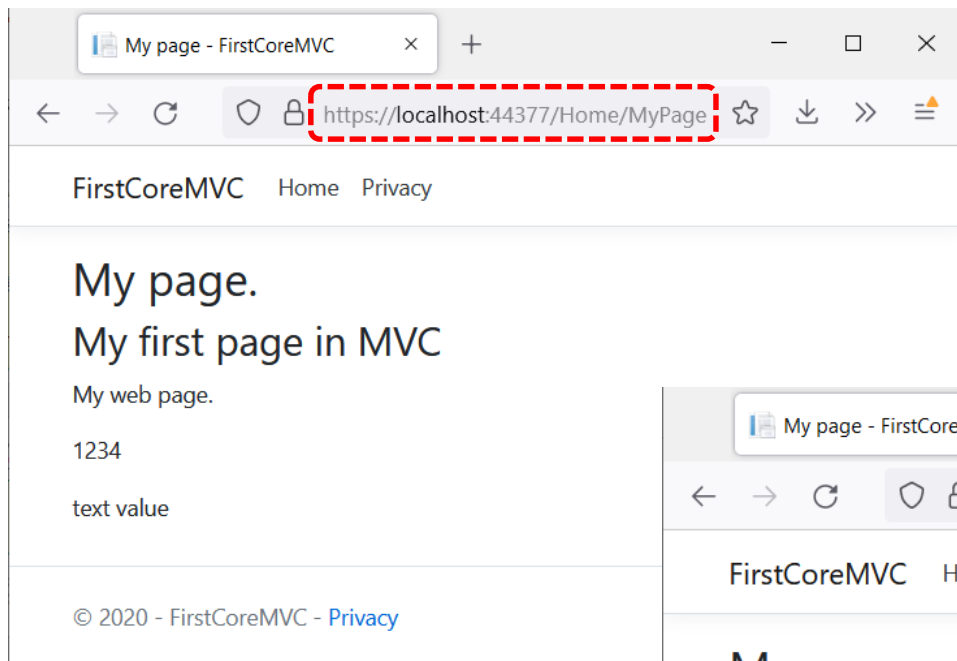
    endpoints.MapControllerRoute(
        name: "MP2",
        pattern: "MP/{number}/{name},{other}",
        defaults: new { controller = "Home", action = "MyPage2" });
    // default
})
```

- Scenariusz użycia:
  - Uruchomić aplikację
  - Wpisać adres <http://localhost:19253/Home/MP> (działa reguła „MP”)
  - Wpisać adres <http://localhost:19253/MP/25/Dariusz,hammer> (działa reguła „MP2”)
  - Wpisać adres <http://localhost:19253/MP/25/Dariusz> (żadna reguła nie działa, nie pasuje do żadnego wzorca, dla „MP2” nie podano wartości domyślnych)
  - Wpisać adres <http://localhost:19253/Home/MyPage> (działa reguła domyślna)
  - Wpisać adres <http://localhost:19253/Home/MyPage2> (działa reguła domyślna, ale brakuje wartości)

# Przykład działania



# Przykład działania



## Lokalna metoda routingu

- Można też ustawić metodę routingu używając adnotacji przed metodą w sposób pokazany poniżej:

```
[Route("M3/{no}/{word},{something}")]  
public IActionResult MyPage3(int no, string word, string something)  
{  
    ViewBag.Message = $"Page with local routing rule {nameof(no)}={no}" +  
        $" {nameof(word)}={word}, {nameof(something)}={something}";  
    return View("MyPage");  
}
```

Controller/HomeController.cs



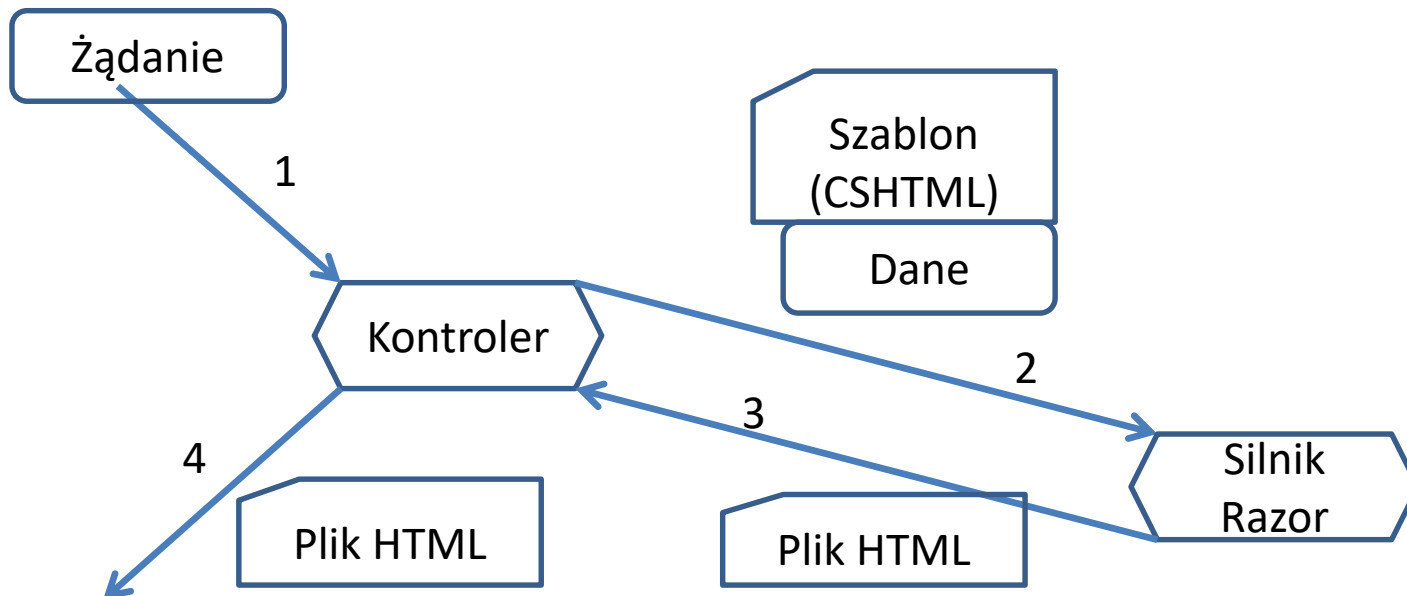
- Przykład użycia: <https://localhost:19253 /M3/123/hammer,ball>
- Istnieje też wiele innych adnotacji do zarządzania routingu (np. przed klasą kontrolera).
- Reguły routingu i mapowania argumentów z adresu to pewien język formalny, szczegóły można znaleźć w dokumentacji Microsoftu.
  - <https://docs.microsoft.com/pl-pl/aspnet/core/fundamentals/routing?view=aspnetcore-5.0>

## Silniki widoków – silnik Razor

- Do tworzenia widoków służą silniki. Przekształcają one szablony widoku w pewnym języku będący mieszanką HTML i języka tegoż silnika w stronę HTML.
- Obecnie w projektach ASP .Net od kilku lat dostępny jest silnik Razor.
- Język silnika Razor cechuje się tym, że większość wyrażeń, kodu itd., nie będących kodem HTML, zaczyna się od znaku '@' :
  - Mam `@dataView["age"]` lat.
- Pliki z szablonami dla tego silnika w przypadku MVC posiadają rozszerzenie `.cshtml`.

## Działanie silnika widoków

- Kontroler, gdy żądanie (1) zostanie skierowane do konkretnego kontrolera:
  - Wybiera właściwy szablon strony WWW (2),
  - Dokłada dane potrzebne do wypełnienia szablonu (2)
  - Wysyła to wszystko do silnika widoków(2)
  - Otrzymuje w wyniku stronę HTML (3), którą przesyła do użytkownika(4)
    - niebezpośrednio



## Przekazywanie danych do widoku

- Do przekazywania danych tymczasowych do widoku mamy kilka gotowych składowych klasy `Controller`.
  - Do przekazywania danych z modelu (wzorzec MVC) będzie używany inny sposób.
- Są to kolekcje typu słownika, czyli zawierają pary <klucz, wartość>
  - Klucz jest typu `string`.
- Dane można wysłać poprzez słownik `ViewData` (typu `ViewDataDictionary<dynamic>`) wartości dynamicznych, lub poprzez składową typu dynamicznego (typu `dynamic`) `ViewBag`, który jest „opakowaczem” obiektu `ViewData`.
  - Ponieważ używamy typów dynamicznych można wstawić wartość **dowolnego typu**.
- Ten słownik jest pamiętany tylko przy przesyłaniu **z kontrolera do** powiązanego z nim **widoku**. Jeśli w kontrolerze następuje przekierowanie do innej akcji poprzez operację `RedirectToAction()`, należy użyć słownika `TempData`. Dane z tego słownika nie są tracone podczas przekierowywania strony.



# Metoda ViewDataProbe w HomeController

Controller/HomeController.cs

```
public IActionResult ViewDataProbe()
{
    ViewData["Message"] = "ViewDataProbe";
    List<string> colors = new List<string>();
    colors.Add("red");
    colors.Add("green");
    colors.Add("blue");

    // obiekt ViewData jest składową obiektu Controller
    ViewData["listColors"] = colors;
    ViewData["dateNow"] = DateTime.Now;
    ViewData["name"] = "Dariusz";
    ViewData["age"] = 20;

    // wynik metody View() zwracany jako wynik tej metody
    return View("ViewDataProbe");
    // return View("ViewBagProbe");
}
```

# Metoda ViewBagProbe w HomeController

Controller/HomeController.cs

```
public IActionResult ViewBagProbe()
{
    ViewBag.Message = "ViewBagProbe";
    List<string> colors = new List<string>();
    colors.Add("red");
    colors.Add("green");
    colors.Add("blue");

    // obiekt ViewData jest składową obiektu Controller
    ViewBag.listColors = colors;
    ViewBag.dateNow = DateTime.Now;
    ViewBag.name = "Dariusz";
    ViewBag.age = 20;

    // wynik metody View() zwracany jako wynik tej metody
    // return View("ViewDataProbe");
    return View("ViewBagProbe");
}
```

# Plik widoku ViewDataProbe.cshtml

View/Home/ViewDataProbe.cshtml

```
@{
    ViewBag.Title = "Data Probe - ViewBag";
}
<h2>@ViewBag.Title.</h2>
<h5>@ViewBag.Message.</h5>
My data:
<br />
<b> Name: @ViewData["name"]</b>
<br />
<b> Age: @ViewData["age"]</b>
<br />
Selected colors:
<ul id="colors">
    @foreach(var color in ViewData["listColors"] as List<string>){
        <li >
            <font color="@color"> @color</font>
        </li>
    }
</ul>
<p>
    @ViewData["dateNow"]
</p>
<p>In engine Razor language (CSHTML)</p>
```

# Plik widoku ViewBagProbe.cshtml

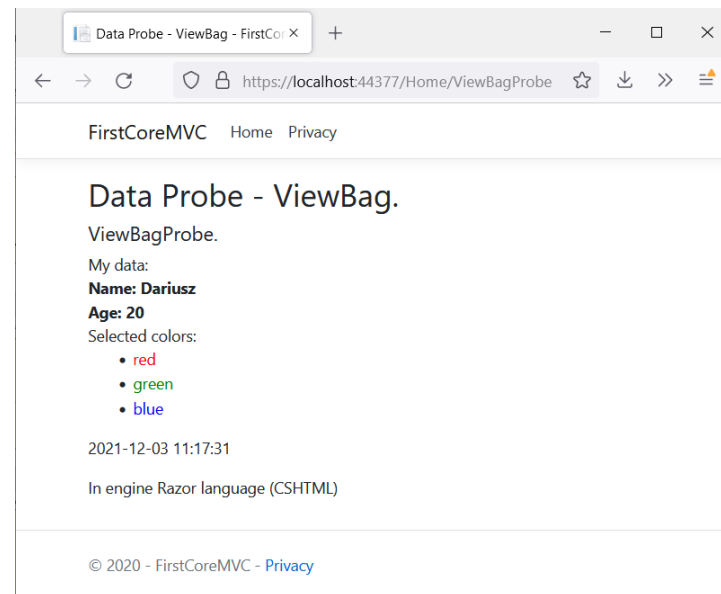
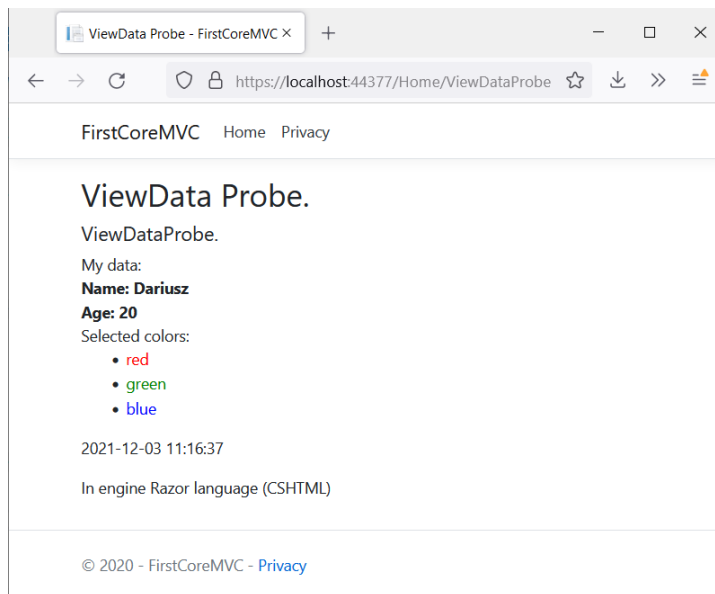
View/Home/ViewBagProbe.cshtml

```
@{
    ViewBag.Title = "Data Probe - ViewBag";
}
<h2>@ViewBag.Title.</h2>
<h5>@ViewBag.Message.</h5>
My data:
<br />
<b> Name: @ViewBag.name</b>
<br />
<b> Age: @ViewBag.age</b>
Selected colors:
<ul id="colors">
    @foreach(var color in ViewBag.listColors){
        <li >
            <font color="@color"> @color</font>
        </li>
    }
</ul>
<p>
    @ViewBag.dateNow
</p>

<p>In engine Razor language (CSHTML)</p>
```

# Scenariusz użycia

- Uruchomić aplikację
- Wpisać w adres przeglądarki:
  - <http://localhost:21493/Home/ViewDataProbe>
  - <http://localhost:21493/Home/ViewBagProbe>
- Zatrzymać aplikację.
- Zamienić na skomentowane return-y w metodach `ViewDataProbe()` i `ViewBagProbe()`
- Ponownie uruchomić aplikację
- Wpisać te same adresy
- Zamknąć aplikację.
- Wniosek: `ViewData` i `ViewBag` można używać zamiennie zarówno w kodzie C# jak i w kodzie CSHTML.



## Plik widoku AllDataProbe.cshtml

- Przemieszczone użycie TempData, ViewData i ViewBag.
- Brakujące elementy słownika zamieniane są na puste string-i.

View/Home/AllDataProbe.cshtml

```
@{
    ViewBag.Title = "All Data Probe"; // można tworzyć nowe pola
    ViewData["proba"] = "jest"; // lub elementy słownika (to jest to samo)
}
<h2>@ViewBag.Title.</h2>
<h5>@ViewBag.Message.</h5>
<br />
Probe: @ViewBag.proba.
<br />
My Data:
<br />
<b> Name: @ViewData["name"]</b>
<br />
<b> Age: @ViewBag.age</b>
<br />
<b> Error: @TempData["error"]</b>
<p> RandomNumber = @ViewBag.random</p>
<p> TempData["random"] = @TempData["random"]</p>
<br />
```

## Akcja z przekierowaniem do innej

- Testowe metody akcji AllDataProbe i AllDataProbeRedirect

```
public IActionResult AllDataProbe()
{
    ViewBag.random = RandomNumber;
    // gdybyśmy chcieli odczytać wartość TempData w Akcji
    //var message = TempData["error"];
    ViewData["name"] = "Dariusz";
    // nie ustawiam ViewData["age"], żeby też nie nadpisać
    return View();
}

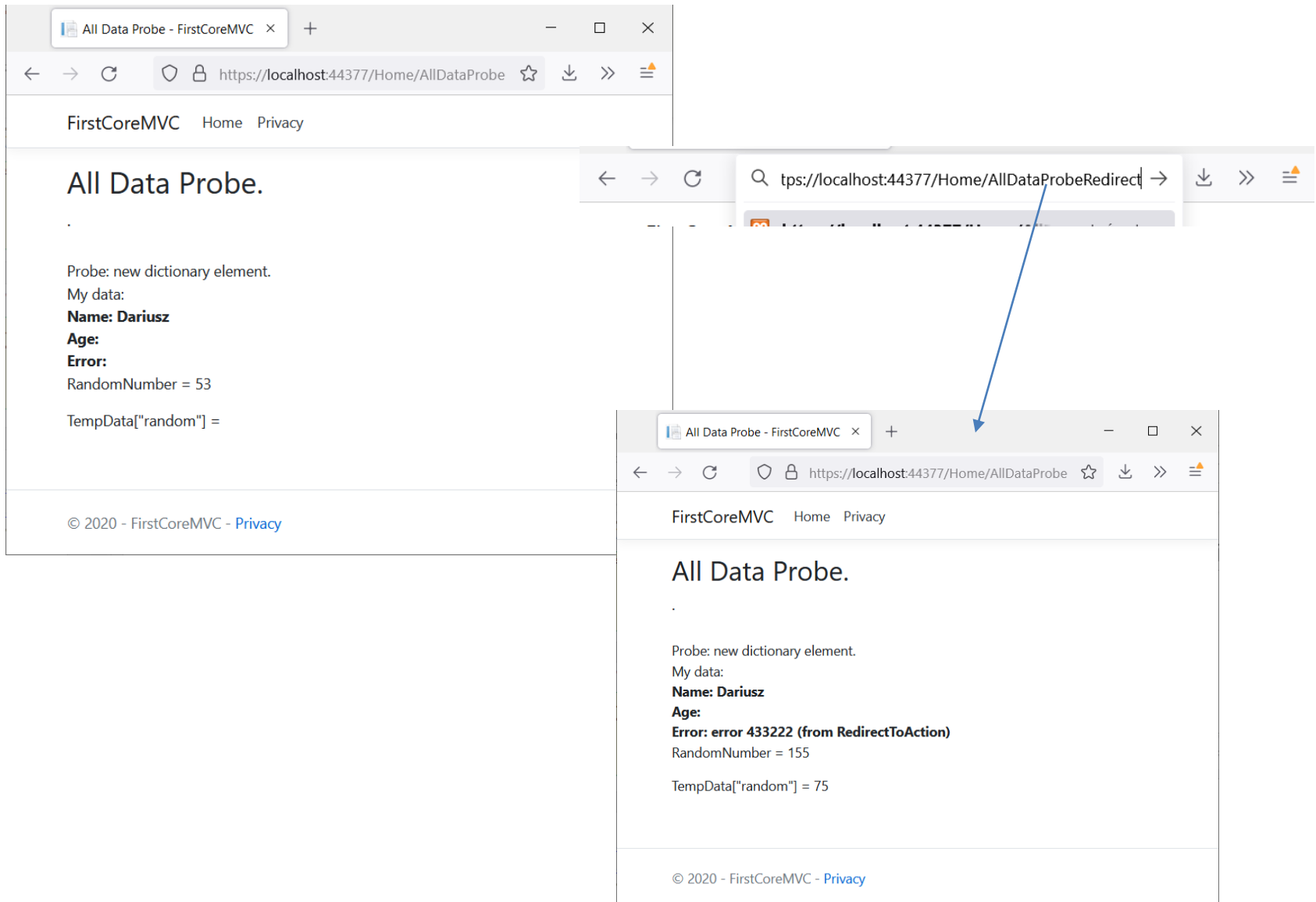
public IActionResult AllDataProbeRedirect()
{
    ViewBag.random = RandomNumber;
    TempData["random"] = RandomNumber;
    // ta dana zostanie przeniesiona
    TempData["error"] = "trzeba było zrobić redirect";
    // ta dana zostanie usunięta podczas redirect
    ViewData["age"] = 90;
    return RedirectToAction("AllDataProbe");
}
```

## Scenariusz użycia

- Uruchomić aplikację
- Wpisać w adres przeglądarki:
  - <http://localhost:21493/Home/AllDataProbe>
    - Stworzona w pliku CSHTML para-klucz jest
    - Nie ma wieku i błędu
  - <http://localhost:21493/Home/AllDataProbeRedirect>
    - Stworzona w pliku CSHTML para-klucz jest
    - Jest błąd z TempData, ale nie ma wieku, dane z ViewData nie zostały przekazane



# Przykład działania:



## Dane z modeli, cykl życia kontrolera

- Obiekty TempData i ViewData służą głównie do przekazywania danych niezwiązanych z modelami tworzonej aplikacji.
- Głównymi elementami aplikacji użytkownika będą dane zaczerpnięte z modeli. Jedne modele służą do tworzenia interfejsu (strony WWW), inne do danych od użytkownika, jeszcze inne do danych domenowych. Używanie do tego TempData i ViewData nie jest wskazane i jest niepoprawnym stylem programowania.
- Żądania HTTP są bezstanowe, stąd obiekt kontrolera istnieje tylko na czas jego obsługi.
- Po obsłużeniu żądania obiekt „ginie” – demonstracja liczby losowej RandomNumber w kontrolerze Home i akcji AllDataProbeRedirect() (na poprzednim slajdzie).

Controller/HomeController.cs

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    private static readonly Random random = new Random();
    // property for tests
    public int RandomNumber { get; set; }

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
        RandomNumber = random.Next(0, 1000);
    }
    // ...
}
```

# **ELEMENTY DODATKOWE FRONTENDU**

# Bootstrap

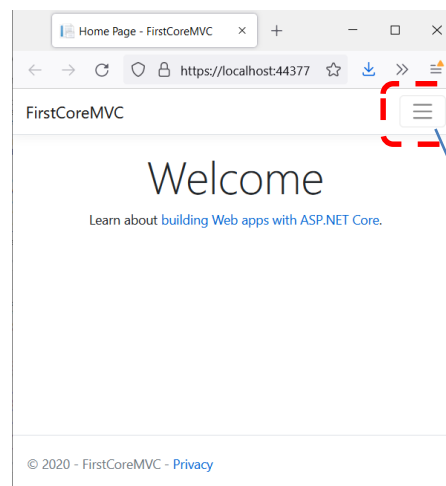
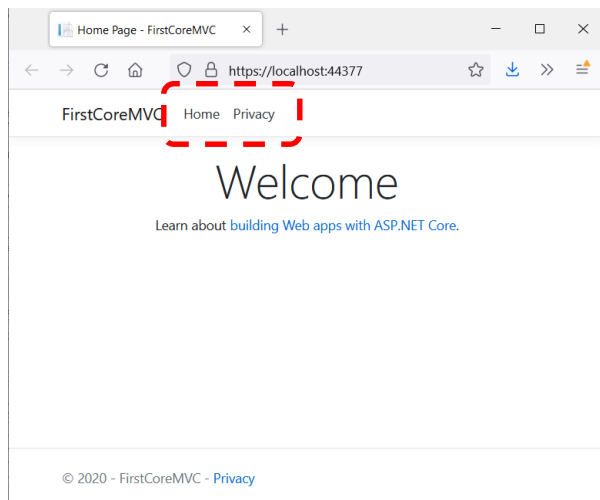
- Bootstrap - biblioteka CSS (i operacji na nich), rozwijana przez programistów Twittera
  - Oprócz stylów Bootstrapa należy zaimportować również skrypty jQuery (jak poniżej), najczęściej na końcu strony. Są potrzebne do działania i animacji.
- Służy głównie estetyce. Umożliwia łatwą **responsywność** aplikacji webowych (dostosowanie się automatycznie do wielkości ekranu, na którym jest wyświetlana)!
  - Wiele zdefiniowanych znaczników i klas („navbar” itd.)
  - Dodatkowe atrybuty
  - Część z animacją

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@RenderSection("Scripts", required: false)
</body>
</html>
```

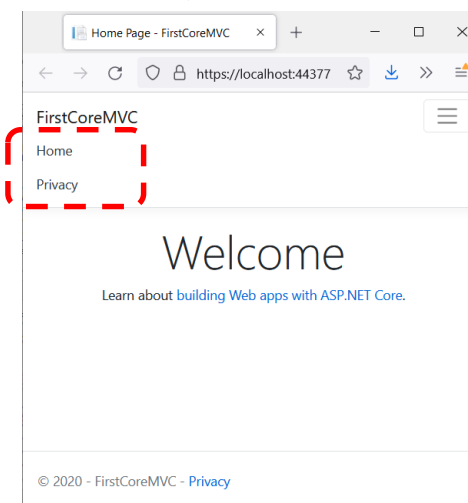
```
<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
  <div class="container">
    <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">FirstCoreMVC</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"
      aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

# Responsywność elementu <nav> w Bootstrapie

- Po zmianie szerokości okna przeglądarki na mniejszą, zamiast opcji menu pojawia się po prawej stronie „chamburger”, który można rozwinąć i zobaczyć opcje.



Click



# jQuery

- jQuery – biblioteka programistyczna dla języka JavaScript, ułatwiająca korzystanie z JavaScriptu (w tym manipulację drzewem DOM).
  - Wiele sposobów użycia funkcji `$()`. W argumencie użycie reguł CSS (np. `"#forjQuery"`), a po kropce co ma być dalej wykonywane na jednym lub więcej znalezionych elementach DOM.
- Przydatne: zmiany w importowanym pliku `.js` są po odświeżeniu strony widoczne w przeglądarce (nie trzeba wyłączać-włączać serwera).

The screenshot displays a web application interface with three main components:

- Code Editor (Top):** Shows the `site.js` file. It contains a `probe` function that updates the content of two elements: `document.getElementById("forJavascript").innerHTML` and `$("#forjQuery").html()`. The function is called at the bottom of the file.
- Code Editor (Bottom Left):** Shows the `TestJS.cshtml` file. It uses Razor syntax to set the page title to "Test JS" and render two divs with IDs `forJavascript` and `forjQuery`.
- Preview (Bottom Right):** Shows the rendered HTML page. It has a header "FirstCoreMVC" with links for "Home" and "Privacy". The main content area displays "Test JS." followed by the text generated by the JavaScript functions: "Text generate by function of element ID" and "Text generate by jQuery function".

Dodatek

# TESTOWE URL

# Testowe URL

- <https://localhost:44377/Home/Index>
- <https://localhost:44377/Home/>
- <https://localhost:44377/>
- <https://localhost:44377/Home/Index/5>
- <https://localhost:44377/Home/Index/5/4>
- <https://localhost:44377/Home/Privacy>
- <https://localhost:44377/Home/MP>
- <https://localhost:44377/MP/25/Dariusz,hammer>
- <https://localhost:44377/MP/25/Dariusz>
- <https://localhost:44377/Home/MyPage>
- <https://localhost:44377/Home/MyPage2>
  
- <https://localhost:44377/Home/ViewDataProbe>
- <https://localhost:44377/Home/ViewBagProbe>
  
- <https://localhost:44377/Home/AllDataProbe>
- <https://localhost:44377/Home/AllDataProbeRedirect>
  
- <https://localhost:44377/Home/AllDataProbe>
- <https://localhost:44377/Home/AllDataProbeRedirect>
- <https://localhost:44377/Home/TestJS>