

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
- Wzorzec MVC Core
 - `Program.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 ELEMENTY WZORCA PROJEKTOWEGO: 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)

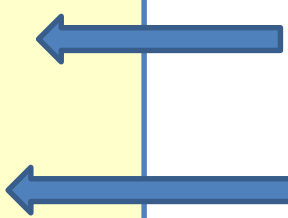
- **Inversion of Control (IoC)**, czyli **mechanizm**, dzięki któremu moduły wyższego poziomu mogą zależeć od abstrakcji, a nie od konkretnej implementacji modułu niższego poziomu.
- Utworzona musi zostać abstrakcja/interfejs
 - Oraz ewentualne jej implementacje

```
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
- W ASP .Net Core najczęściej stosowana jest technika (poprzez mechanizm odbicia) wstrzykiwania zależności **przez konstruktor**. Jednak również **pozostałe** są **stosowane**.
- 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}");
    }
}

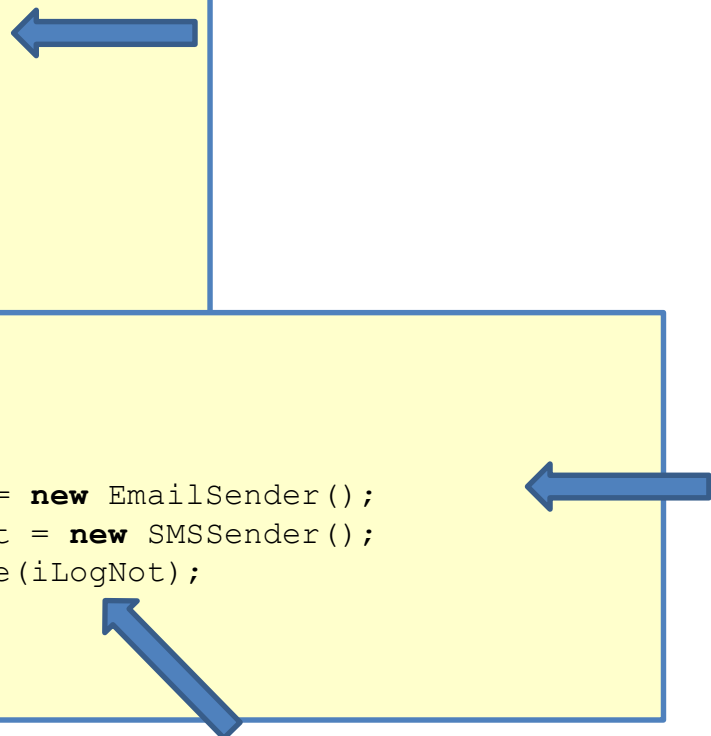
public class SMSSender : ILogNotification {
    public void Notify(string message)
    {
        Console.WriteLine($"Texting: {message}");
    }
}
```

DI przez konstruktor

- Bardzo dobrym 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));  
    }  
}
```

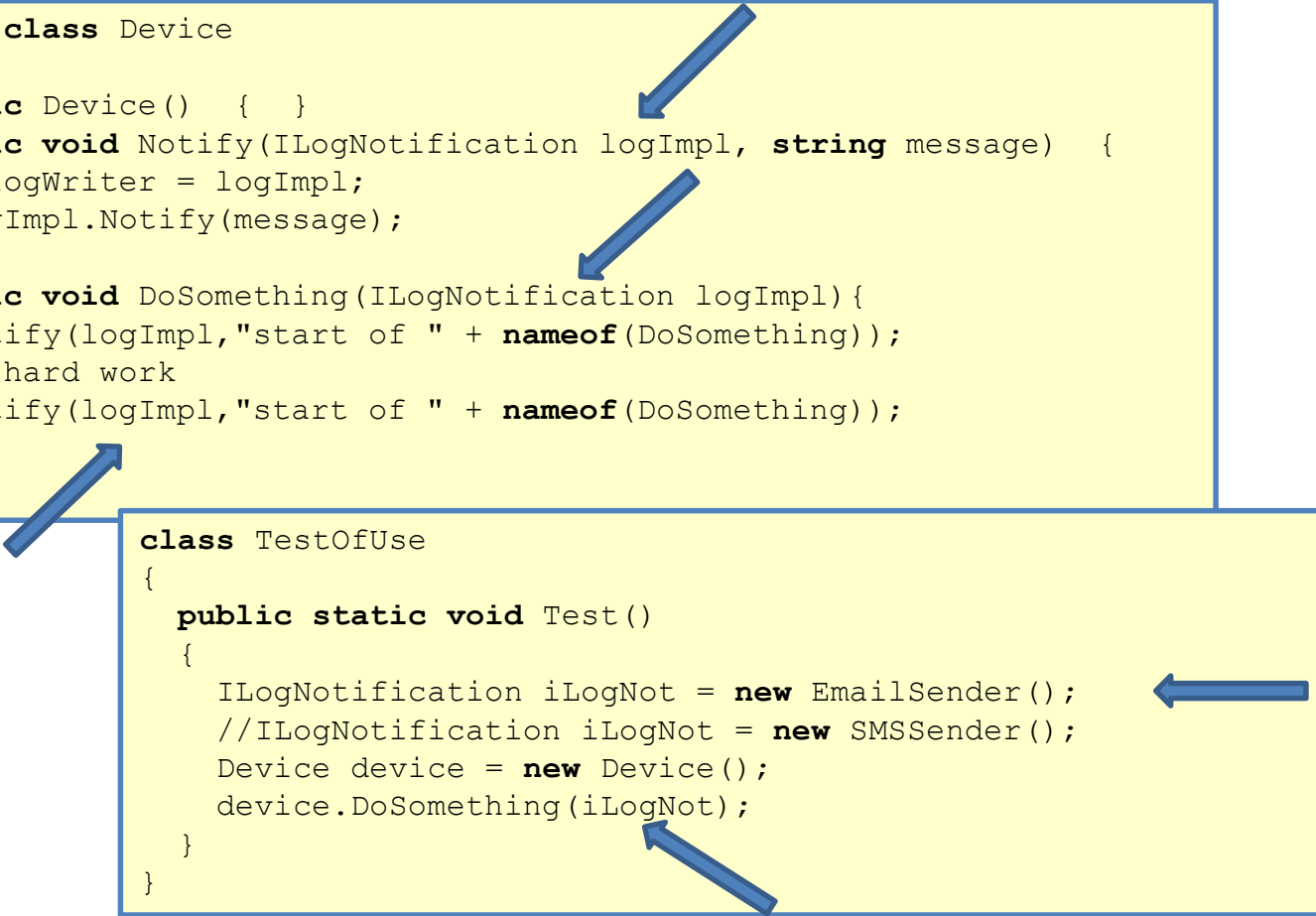


```
class TestOfUse  
{  
    public static void Test()  
    {  
        ILogNotification iLogNot = new EmailSender();  
        //ILogNotification iLogNot = new SMSSender();  
        Device device = new Device(iLogNot);  
        device.DoSomething();  
    }  
}
```

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();
    }
}
```

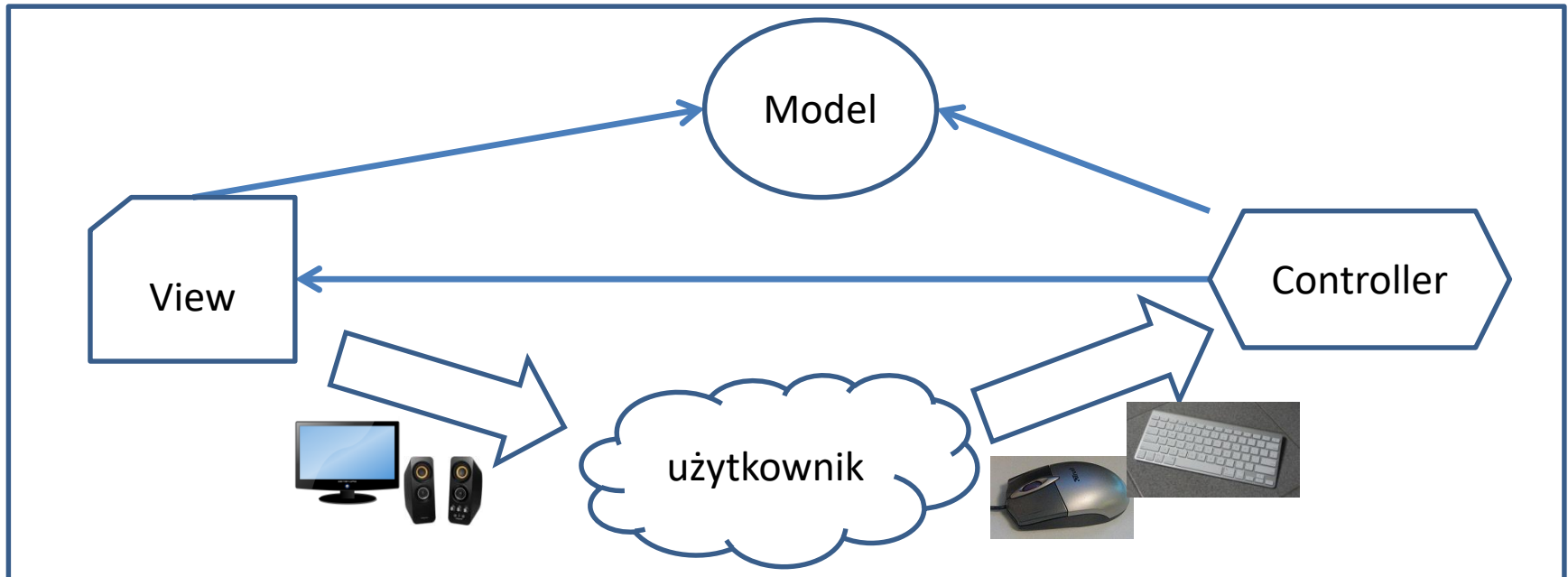
Kontenery obiektów wstrzykiwanych

- Można łączyć te 3 wersje wstrzykiwania zależności.
- W przypadku wielu obiektów wstrzykiwanych (niezmiennych w trakcie życia systemu) niezłym rozwiązaniem są kontenery serwisów, 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).
- Siła 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 2022 wybrać projekt „Aplikacja internetowa platformy ASP.NET Core (Model-View-Controller)”

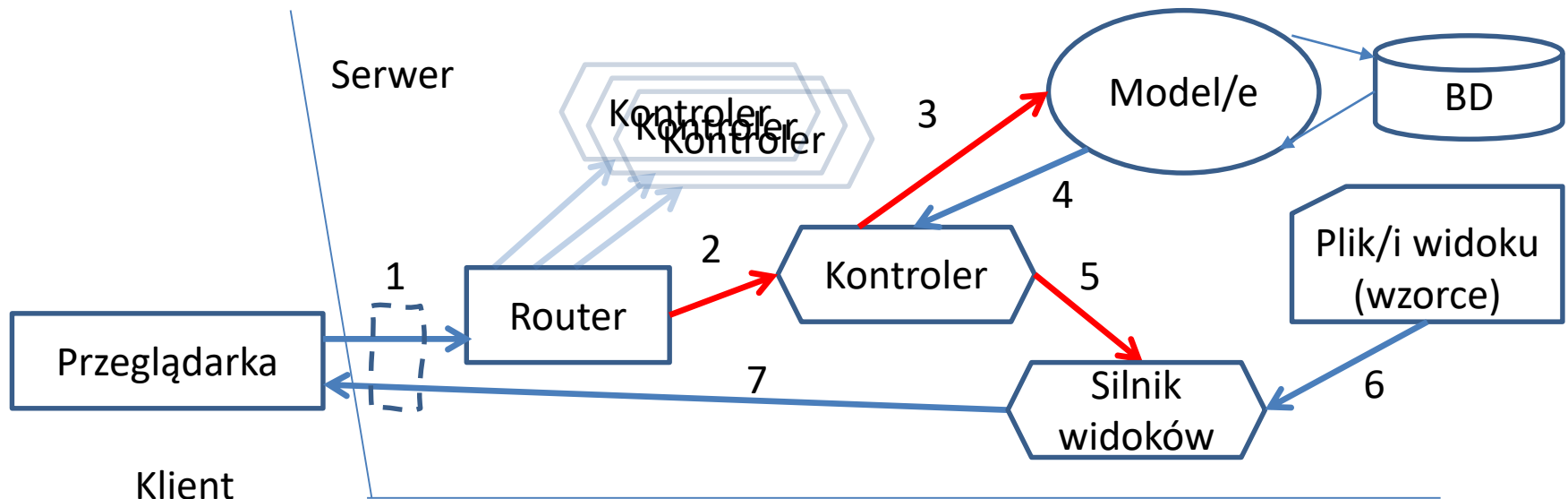
The screenshot shows the 'Create new project' dialog in Visual Studio 2022. At the top, the language is set to 'C#' and the template is 'Internet'. A red rectangle highlights the selected project type: 'Aplikacja internetowa ASP.NET Core (Model-View-Controller)'. Below this, a description states: 'Szablon projektu służący do tworzenia aplikacji platformy ASP.NET Core z przykładowymi widokami i kontrolerami platformy ASP.NET Core MVC. Tego szablonu można także użyć dla usług HTTP RESTful.' Below the description are buttons for different deployment targets: 'C#', 'Linux', 'macOS', 'Windows', 'Chmura', 'Usługa', and 'Internet'. The 'C#' button is selected. The main form is divided into two sections. The left section, titled 'Nazwa projektu', contains fields for 'Nazwa projektu' (set to 'FirstNet8MVC'), 'Lokalizacja' (set to 'C:\Users\darius\source\repos'), and 'Nazwa rozwiązania' (set to 'FirstNet8MVC'). There is a checkbox for 'Umieść rozwiązanie i projekt w tym samym katalogu' which is unchecked. The right section, titled 'Platforma', contains fields for 'Platforma' (set to '.NET 8.0 (Długoterminowa pomoc techniczna)'), 'Typ uwierzytelniania' (set to 'Brak'), 'System operacyjny kontenera' (set to 'Linux'), and 'Typ kompilacji kontenera' (set to 'Dockerfile'). There are checkboxes for 'Konfiguruj dla protokołu HTTPS' (checked), 'Włączanie obsługi kontenerów' (unchecked), and 'Nie używaj instrukcji najwyższego poziomu' (checked).

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.

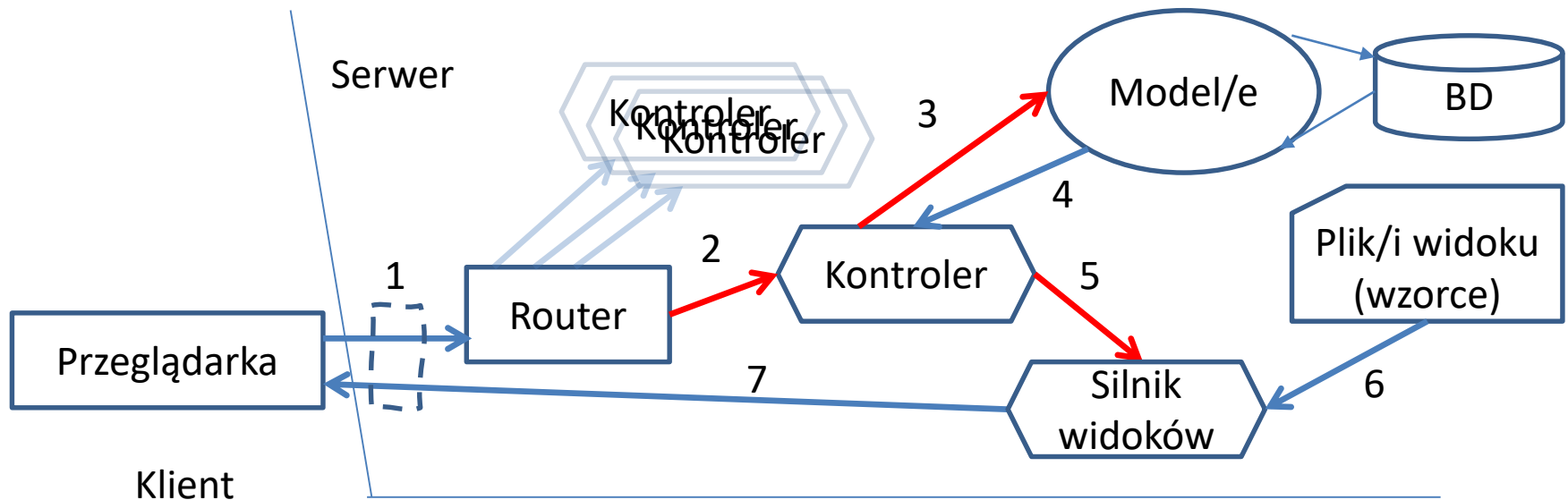
MVC w kontekście aplikacji webowych 1/3

- 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 obiekt ten nie będzie używany wprost przez programistę, 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`.



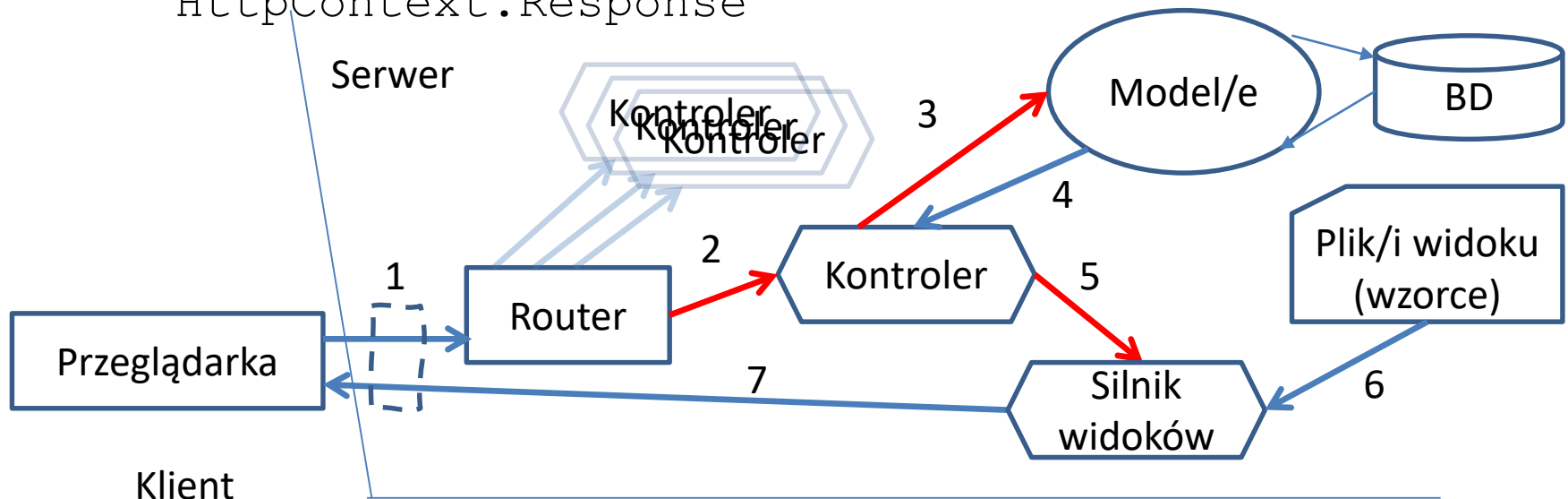
MVC w kontekście aplikacji webowych 2/3

- 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 (wstrzykując potrzebne serwisy) i wywołuje akcję.
 - Podczas konstrukcji kontrolera tworzą się (3) potrzebne instance obiektów modelu a następnie rozpoczyna się wykonanie akcji. Wykonanie akcji przygotowuje dane dla widoku (4).



MVC w kontekście aplikacji webowych 3/3

- 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`

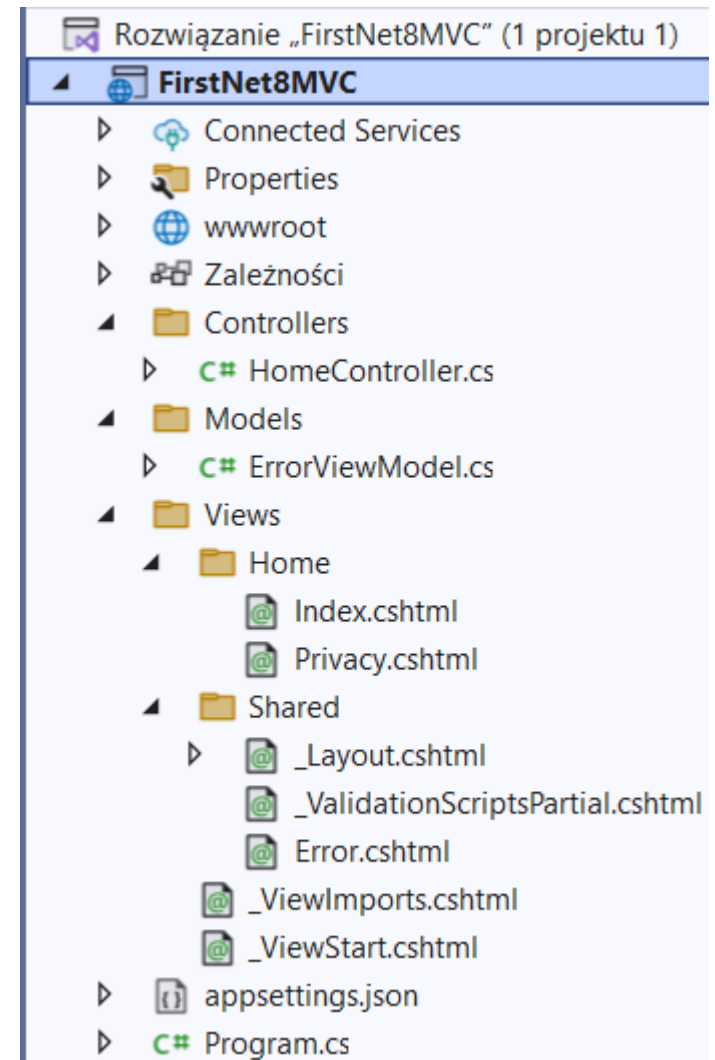


ASP. NET Core MVC

- 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 9.0 (dla .Net 9.0) – 10/2024.
 - Wersja 8.0 jest wersją stabilną, która ma dłuższe wsparcie techniczne.
- Platforma Visual Studio oprócz mechanizmów automatycznego budowania szkieletów klas dla wzorca MVC dostarcza wielu innych elementów ułatwiających budowanie aplikacji webowej:
 - Kontener serwisów implementujący `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 2022 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.
- **Widoki** to tak naprawdę **wzorce stron** HTML, których treść będzie często dynamicznie zmieniana.
- 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.



Układ poprzednich wersji ASP .Net

- Wcześniej instrukcje uruchamiające aplikację rozłożone były na dwa pliki `Program.cs` i `Startup.cs` oraz ich różne metody. Zbudowanie podstawowej aplikacji wymagało przez to dłuższego kodu.
- W metody wstrzykiwano odpowiednie argumenty podczas ich wywoływania.

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

```
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();
    }
}
```

```
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

Plik Program.cs - 1/2

- W wersji .Net 8.0 cały kod tworzenie aplikacji znajduje się w Program.cs
 - Istnieje klasa `WebApplication`, która najpierw tworzy builder (`CreateBuilder()`), który zawiera w sobie wiele właściwości, w tym kontener serwisów (`builder.Services`) typu `IServiceCollection`. Aby stworzyć aplikację w architekturze MVC, należy użyć metodę rozszerzającą `AddControllersWithViews()`.
- 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
 - Dodanie wszystkich potrzebnych serwisów należy wykonać przez zbudowaniem aplikacji poprzez metodę `Build()`.

Program.cs

```
namespace FirstNet8MVC
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddControllersWithViews();

            var app = builder.Build();
        }
    }
}
```


Program.cs - 2/2

- Następnie należy ustawić 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.MapControllerRoute`).

Startup.cs

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
}
```

Różne zestawy serwisów 1/2

- `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 operator kropki:

```
builder.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 1/2

- `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
builder.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 `Program` to **metody rozszerzające**.
 - Można to sprawdzić w VS 2022 ustawiając kursor myszki nad nazwą metody.

```
// Add services to the container.
```

```
builder.Services.AddControllersWithViews();
```


```
var app = builder.Build();
```

 (rozszerzenie) `IMvcBuilder IServiceCollection.AddControllersWithViews()` (+ 1 przeciążenie)
Adds services for controllers to the specified `IServiceCollection`. This method will not register ser

```
}
```

```
app.UseHttpsRedirection();
```


```
app.Use
```

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

```
app.Use
```


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

```
app.UseRouti
```

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

```
app.UseRouting();
```

```
app.UseAut
```

 (extension) `IApplicationBuilder IApplicationBuilder.UseRouting()`
Adds a `Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware` middleware to the specified `IApplicationBuilder`.

```
app.UseEnd
```

A call to `EndpointRoutingApplicationBuilderExtensions.UseRouting(IApplicationBuilder)` must be followed by a call to `EndpointRoutingApplicationBuilderExtensions.UseEndpoints(IApplicationBuilder, Action<Microsoft.AspNetCore.Routin`

Metody kontrolera

- Kontrolery w MVC to klasy dziedziczące po klasie `Controller`.
- 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 IActionResult Index()
    {
        return View(); // return View("Index");
    }
}
```

...

Controllers/HomeController.cs

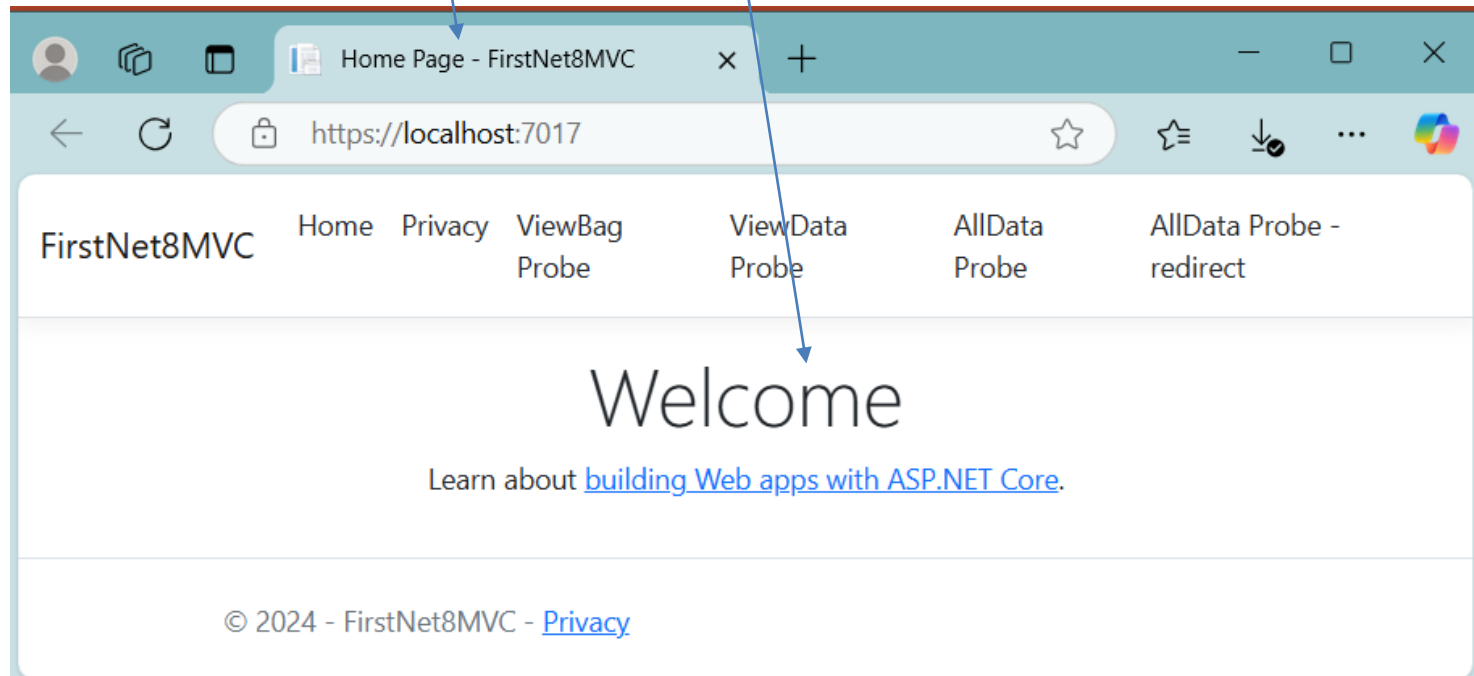
- Spowoduje to przetworzenie i wysłanie strony WWW na podstawie pliku `/Views/Home/Index.cshtml`
- W domyślnym projekcie wytworzonym w środowisku VS można to przetestować po uruchomieniu projektu i wpisaniu w przeglądarkę adresu `http://localhost:12345/Home/Index` (zamiast 12345 może być inny numer portu).

Przykład działania

```
@{  
    ViewData["Title"] = "Home Page";  
}
```

View/Home/Index.cshtml

```
<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>
```



Routing - stare, dobre(?) czasy

- Na początku czasów internetu 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 strumień znaków 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 (między innymi) w pliku `/Program.cs`. Początkowy routing dodany w tym pliku wygląda następująco:

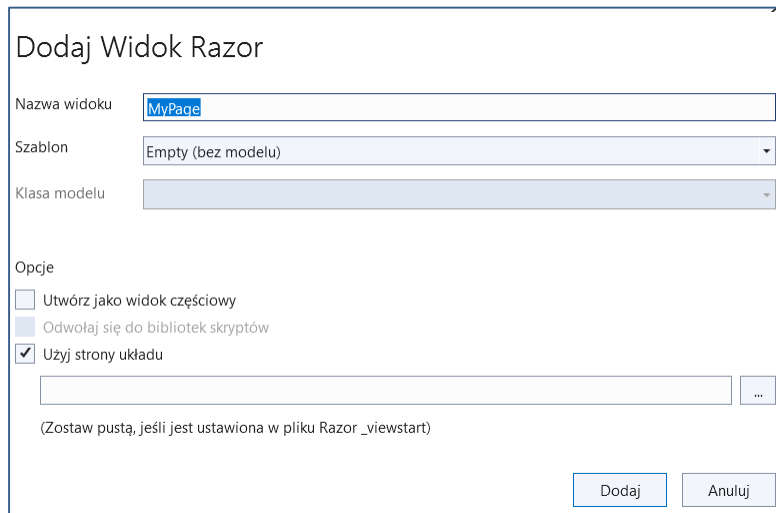
```
app.UseRouting(); // uruchomienie usługi zasad routingu
// ...
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

`Program.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

- Dodajmy własną nową stronę, dodatkowe akcje w kontrolerze `Home` oraz dodatkowe własne metody routingu.
- Dodanie nowego widoku można poprzez kliknięcie PPM na folder `Views/Home`, i wybranie opcji „Dodaj” a następnie „Widok” i jako element szkieletowy „Widok Razor”.
- Strona zostanie wypełniona domyślną treścią, którą zamienimy na poniższą treść:



Dodaj Widok Razor

Nazwa widoku:

Szablon:

Klasa modelu:

Opcje

☐ Utwórz jako widok częściowy

☒ Odwołaj się do bibliotek skryptów

☒ Użyj strony układu

...

(Zostaw pustą, jeśli jest ustawiona w pliku Razor _viewstart)

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
```

Własne akcje

- W kontrolerze `Home` stworzone zostaną dodatkowe akcje, które będą używały dopiero co stworzoną strony
 - Jedna, która do określenia nazwy wzorca strony wykorzysta mechanizm odbicia.
 - Druga, która podaje w argumencie wywołania metody `View()`, który wzorzec ma być użyty.

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

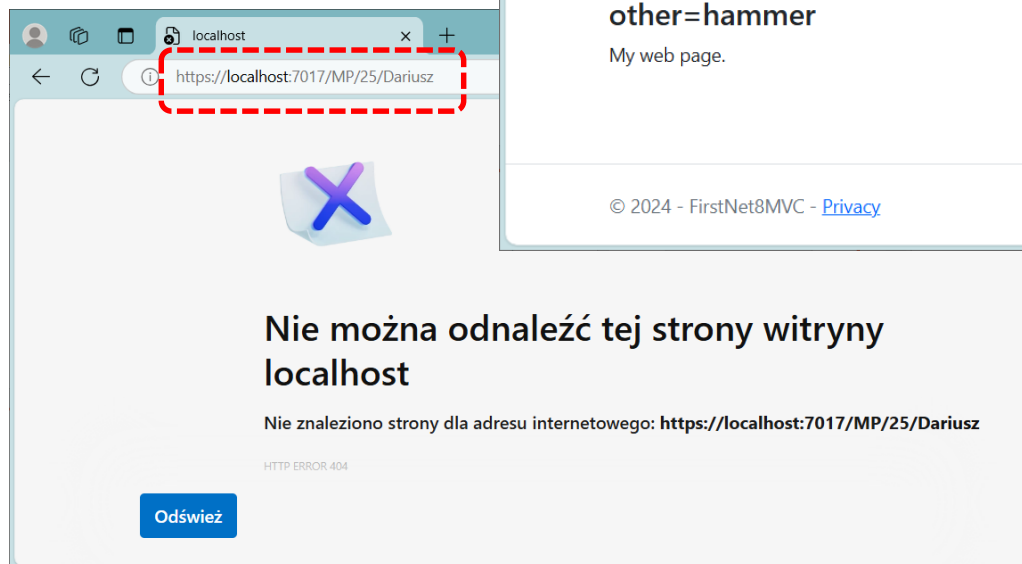
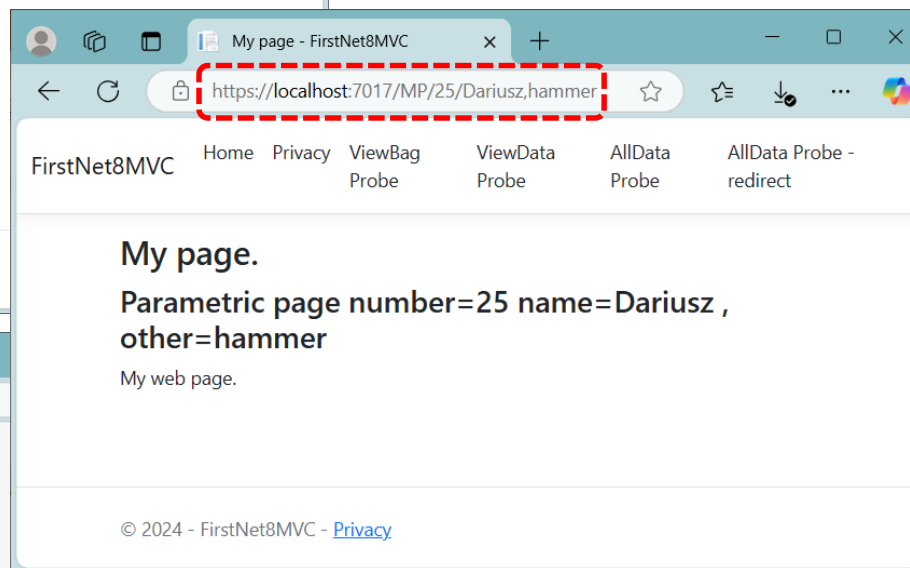
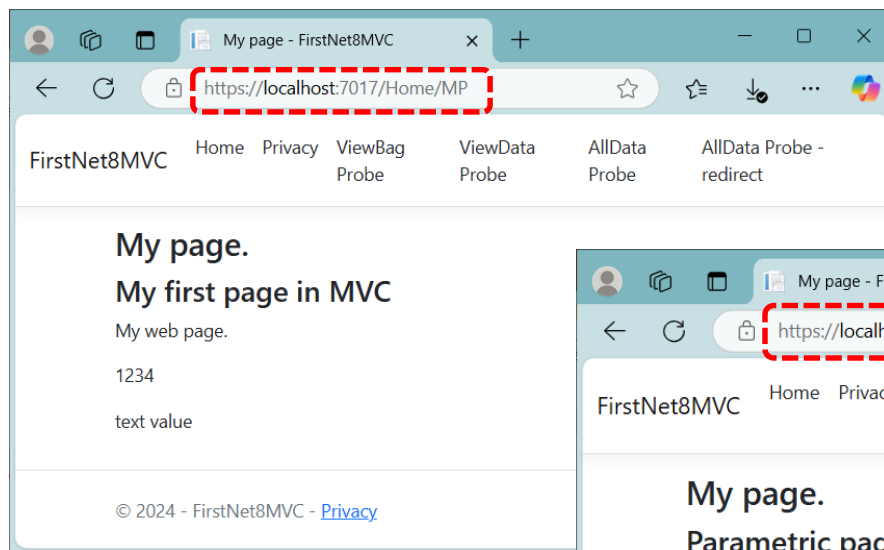
- Z wielu względów możemy chcieć sami ustalić inne reguły routingu. W tym celu należy wstawiać nowe reguły jako kolejne wywołania metody `MapControllerRoute()` dla aplikacji.
 - Jako ostatnią regułę warto zostawić domyślną.

Startup.cs

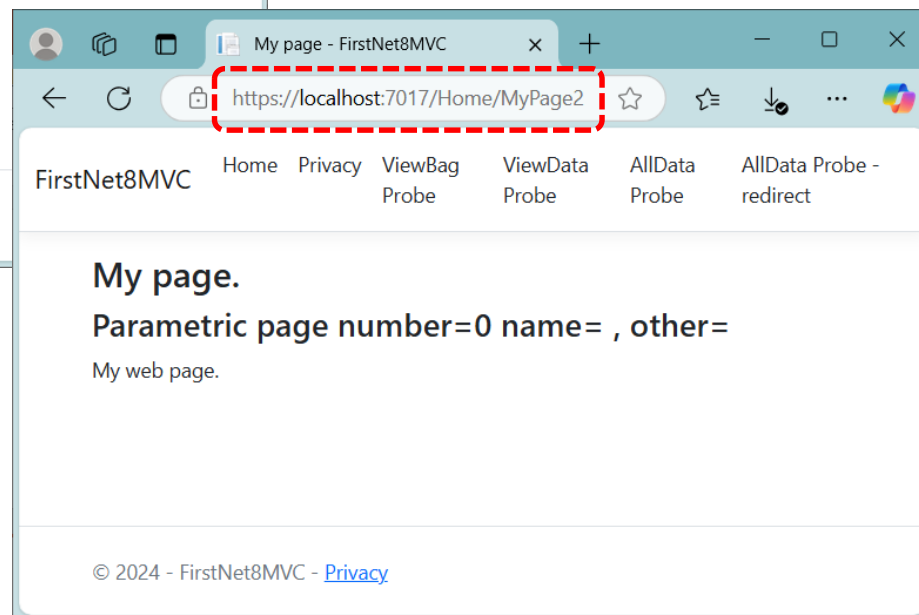
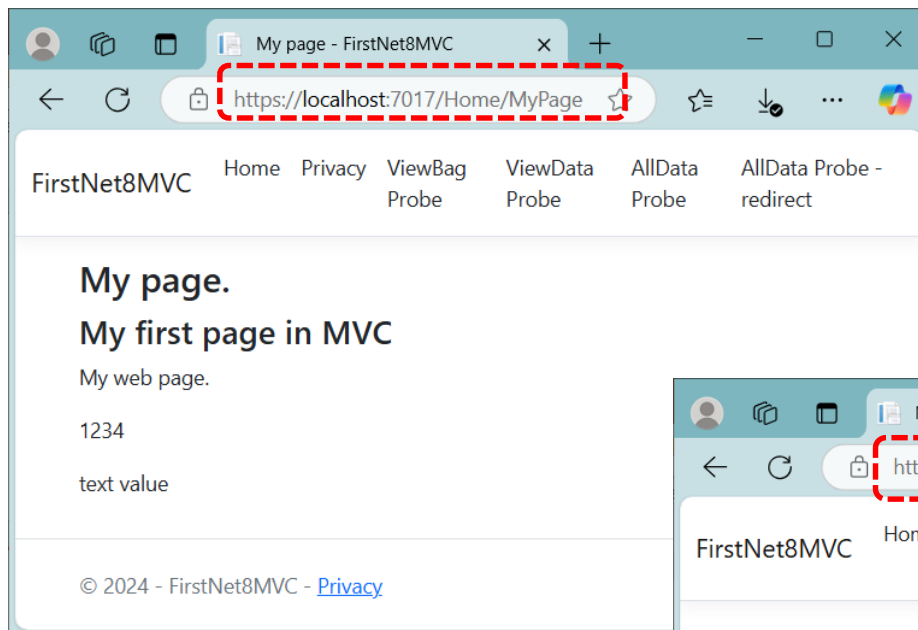
```
app.MapControllerRoute(  
    name: "MP",  
    pattern: "Home/MP",  
    defaults: new { controller = "Home", action = "MyPage" });  
app.MapControllerRoute(  
    name: "MP2",  
    pattern: "MP/{number}/{name},{other}",  
    defaults: new { controller = "Home", action = "MyPage2" });  
app.MapControllerRoute(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

- 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 1/2



Przykład działania 2/2

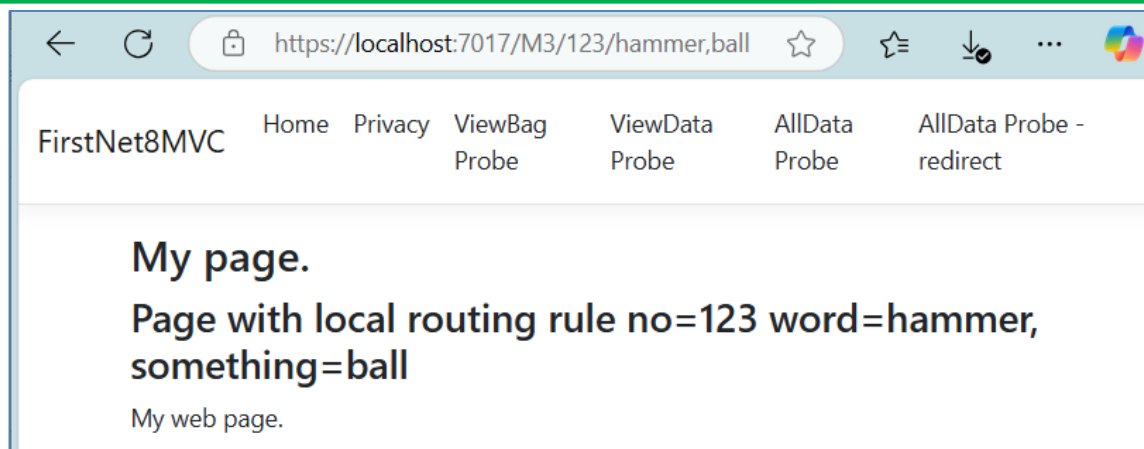


Lokalna reguła routingu

- Można też ustawić regułę routingu używając atrybutu/adnotacji [Route] 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



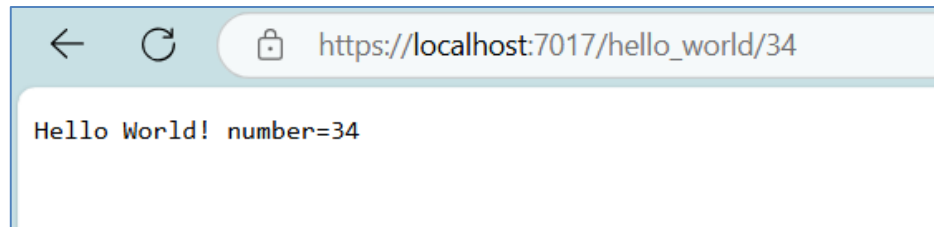
- Przykład użycia: <https://localhost:19253 /M3/123/hammer,ball>
- Istnieje też wiele innych adnotacji do zarządzania routingu (np. przed klasą kontrolera).
- Reguły routingu i mapowania argumentów z adresu to pewien język formalny, szczegóły można znaleźć w dokumentacji Microsoftu.
 - <https://docs.microsoft.com/pl-pl/aspnet/core/fundamentals/routing?view=aspnetcore-8.0>

Generowanie strony HTML

- Można generować stronę HTML wprost, np. dodając metodę routingu jak poniżej (nazwa parametr routingu i nazwa parametru lambda **musi** być taka sama):

```
app.MapGet("/hello_world/{nr}", (int nr) => "Hello World! number="+nr);
```

- Która uruchomi się dla adresu localhost:7017/hello_world/34



- Gdyby nazwy parametrów były różne np.:

```
app.MapGet("/hello_world/{nr}", (int i) => "Hello World! number="+i);
```

- podczas uruchamiania będzie błąd:

An unhandled exception occurred while processing the request.

BadRequestException: Required parameter "int i" was not provided from query string.

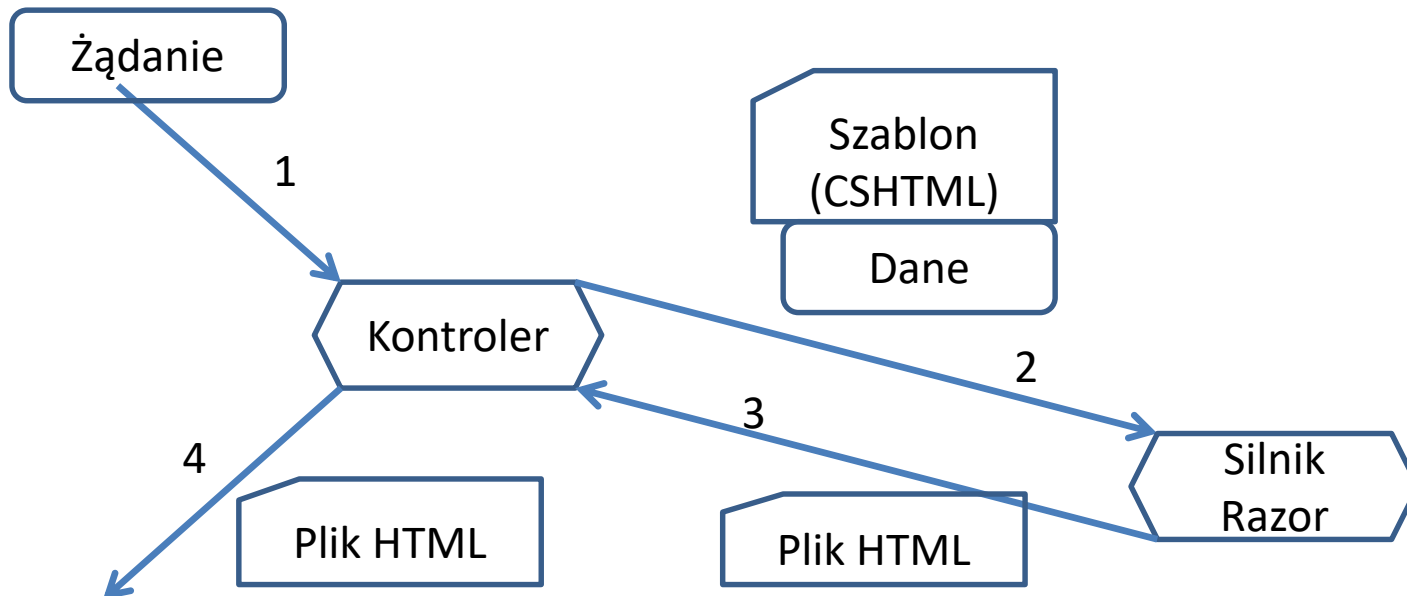
lambda_method2(Closure, object, HttpContext)

Silniki widoków – silnik Razor

- Do tworzenia widoków służą silniki. Przekształcają one szablon 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.
- Więcej szczegółów będzie zaprezentowanych na kolejnych wykładach.

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 metodę `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 ViewBag 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

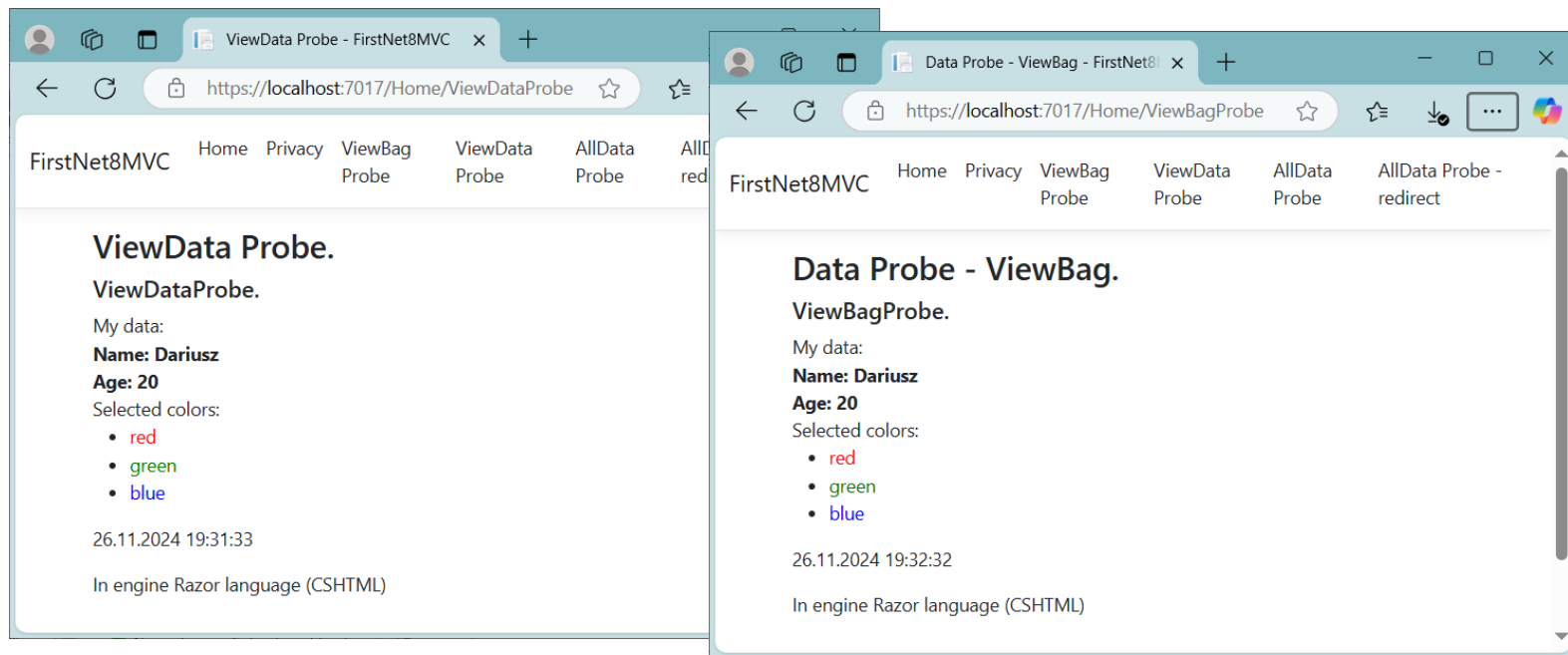
View/Home/ViewBagProbe.cshtml

```
@{
    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"; // you can create new fields
    ViewData["probe"] = "new dictionary element";
                                   // or dictionary elements (it's the same)
}
<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.

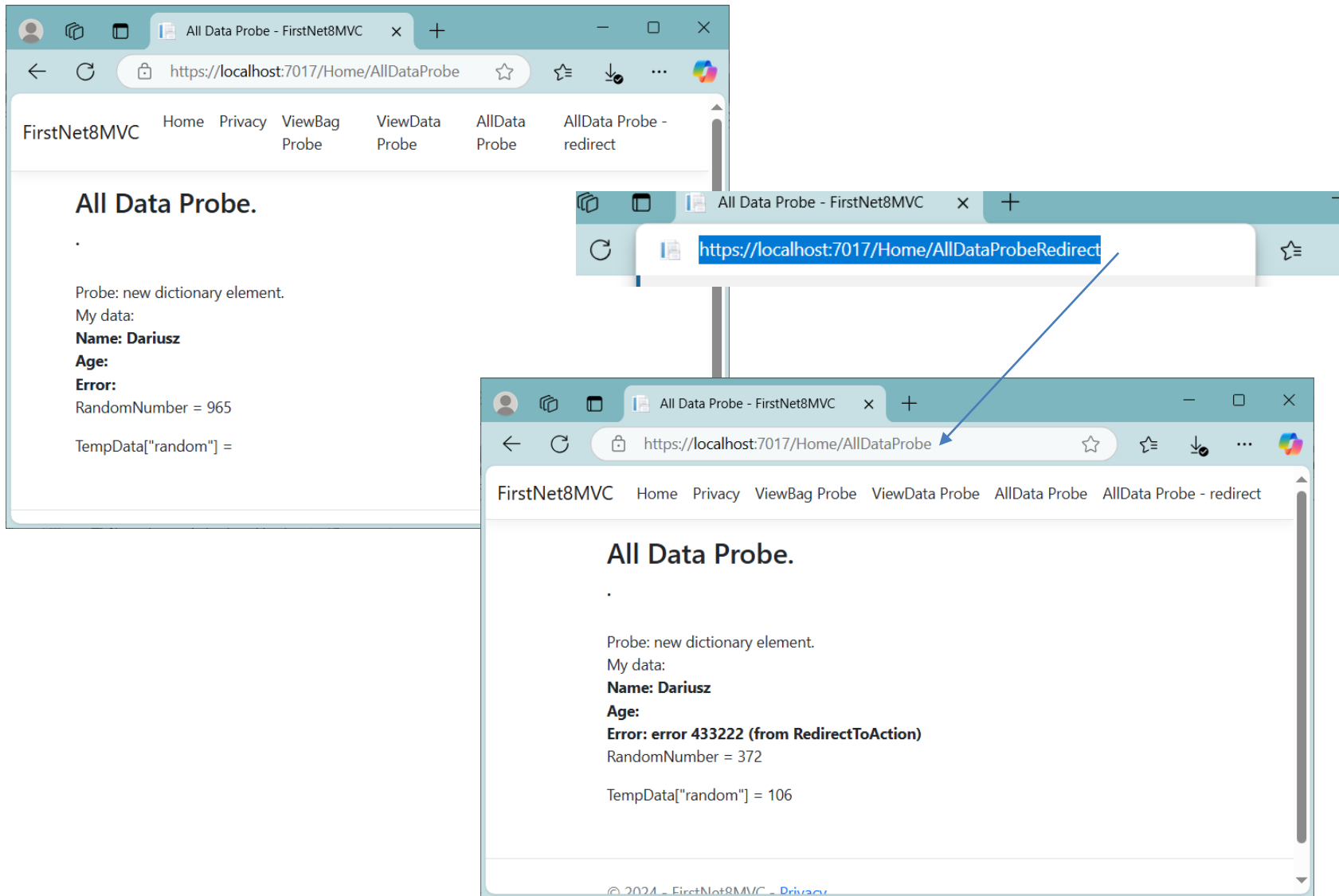
```
public IActionResult AllDataProbe()
{
    ViewBag.random = RandomNumber;
    // if we wanted to read the TempData value in Action
    // var message = TempData["error"];
    ViewData["name"] = "Dariusz";
    // I don't set ViewData ["age"] so as not to overwrite it
    return View();}

public IActionResult AllDataProbeRedirect()
{
    ViewBag.random = RandomNumber;
    TempData["random"] = RandomNumber;
    // this data will be transferred
    TempData["error"] = $"error 433222 (from {nameof(RedirectToAction)})";
    // this data will be deleted during 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.
- W nowotworzonych projektach ASP. NET dodawana automatycznie w układzie stron (layoutcie)
 - Zrzut z ekranu z początku i końca pliku Shared/_Layout.cshtml.
- 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.

```
7 <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
```

```
<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>
```

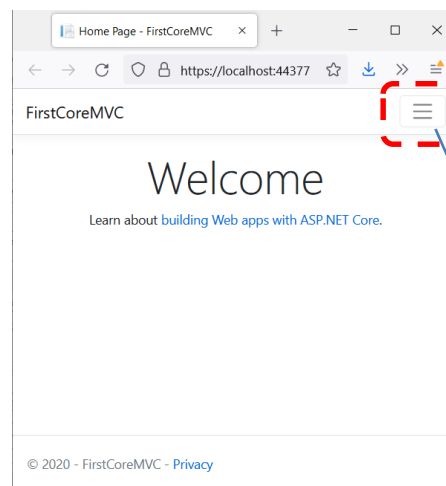
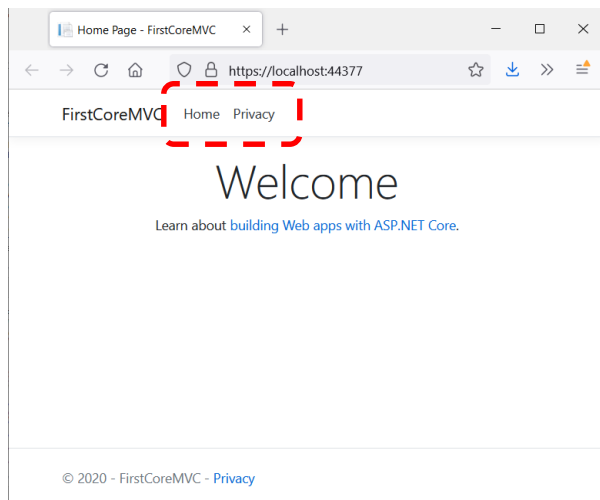
Dodawanie opcji w menu

- Dzięki użyciu Bootstrap-a bardzo łatwo można dodawać nowe opcje poziomego menu.
 - Więcej szczegółów nt. atrybutów zaczynających się od „asp-” pojawi się na kolejnych wykładach.

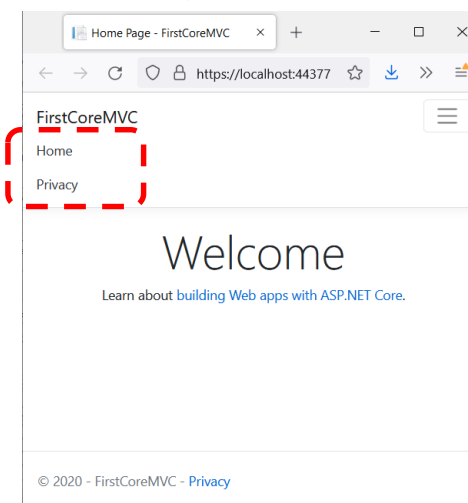
```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="ViewBagProbe">ViewBag Probe</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="ViewDataProbe">ViewData Probe</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="AllDataProbe">AllData Probe</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="AllDataProbeRedirect">AllData Probe - redirect</a>
    </li>
  </ul>
</div>
```

Responsywność elementu <nav> w Bootstrapie

- Po zmianie szerokości okna przeglądarki na mniejszą, zamiast opcji menu pojawia się po prawej stronie „hamburger”, 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 development environment with three main components:

- Top Panel (site.js):** Contains JavaScript code for a `probe()` function. The function uses `document.getElementById` to update the content of an element with ID `forJavascript` and jQuery's `$.html()` to update the content of an element with ID `forjQuery`.
- Bottom Left Panel (HomeController.cs):** Shows a C# controller action `TestJS()` that returns the `View()` method.
- Bottom Middle Panel (TestJS.cshtml):** Shows a Razor view that sets the `ViewBag.Title` to "Test JS", renders it as an `h2` tag, and includes two `div` elements with IDs `forJavascript` and `forjQuery`.
- Bottom Right Panel (Browser Preview):** Shows the rendered page in a browser at `https://localhost:7017/Home/TestJS`. The page title is "Test JS." and it displays the text generated by the 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>