Realizacja Aplikacji Internetowych 2019
ASP.NET Core MVC
laboratorium
K. M. Ocetkiewicz

Środowisko

Są trzy możliwości tworzenia aplikacji .NET MVC. Pierwsza z nich to skorzystanie z Visual Studio i .NET Framework. Tak stworzoną aplikację można uruchomić w serwerze IIS lub w IIS Express wbudowanym w Visual Studio. Druga możliwość to skorzystanie z Mono środowiska MonoDevelop. Umożliwi to uruchomienie aplikacji pod Linuxem (gotową aplikację hostować można np. w Apache z modułem mod_mono lub serwerem XSP), trzeba jednak przygotować się na nieco dodatkowej pracy związanej z konfiguracją tych narzędzi, zwłaszcza jeżeli chcemy skorzystać z nowszej wersji MVC (4 lub 5). Trzecia możliwość to skorzystanie z .NET Core – aplikacje stworzone dla tego środowiska można będzie uruchomić zarówno pod Windowsem jak i Linuxem. Dodatkowo, aplikacja MVC dla .NET Core może być samohostująca – nie będzie potrzebny żaden dodatkowy serwer aby uruchomić ją w docelowym środowisku, wystarczy samo .NET Core Runtime. Niestety, wszystkie te środowiska są ze sobą niekompatybilne: aplikacji MVC dla .NET Framework nie uruchomimy (a już na pewno nie bez wysiłku i modyfikacji kodu) ani na Mono ani pod .NET Core, podobnie w drugą stronę. Środowiska te różnią się także pod względem wygody użytkowania – zestaw Visual Studio + .NET Framework oferuje najbogatszy zestaw narzędzi i wygodę za cenę najtrudniejszej konfiguracji środowiska docelowego i najmniejszej przenośności.

Instalacja MVC

Narzędzia i środowiska dla MVC jest dostępne w instalatorze Visual Studio. Wystarczy pamiętać o włączeniu pakietu „Opracowywanie zawartości dla platformy ASP.NET i sieci Web”.

Na laboratorium będzie omówione ASP.NET Core MVC na platformie .NET Core 2.1 w środowisku Visual Studio 2019 – to oprogramowanie jest dostępne na komputerach w laboratorium.

Dodatkowe informacje na temat ASP.NET Core MVC można znaleźć pod adresem: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1.

Nowy projekt

Należy wyszukać projekt dla języka C# typu “Aplikacja Internetowa platformy ASP.NET Core”. Następnie wybieramy .NET Core i ASP.NET Core 2.1 i szablon “Aplikacja internetowa (Model-View-Controller)”. Mamy również możliwość wyboru sposobu uwierzytelniania. Wybór sposobu innego niż “no authentication” doda do naszego projektu kod uwierzytelniający użytkowników wybraną przez nas metodą. Zaznaczenie “Konfiguruj dla protokołu HTTPS” automatycznie doda do aplikacji obsługę szyfrowanych połączeń i wygeneruje samodzielnie podpisane certyfikaty.

Nowoutworzony projekt jest działającą aplikacją i można ją uruchomić – prezentuje ona kilka przykładowych, powiązanych ze sobą linkami podstron. .NET Core daje nam również możliwość wyboru sposobu uruchomienia comboboxem doklejonym do przycisku uruchamiania: poprzez IIS lub self-hosting. W pierwszym przypadku aplikacja będzie hostowana w IIS Express wbudowanym w Visual Studio, w drugim będzie uruchomiona jako samodzielna aplikacja. Uruchomienie projektu automatycznie uruchamia przeglądarkę i przekierowuje ją na adres naszej aplikacji (czym można manipulować w opcjach projektu).

W przypadku korzystania z .NET Core warto zapoznać się z arykułem

https://docs.microsoft.com/en-us/dotnet/core/tools/telemetry
i rozważyć ustawienie zmiennej środowiskowej `DOTNET_CLI_TELEMETRY_OPTOUT` na `1` lub `true`.

Zawartość projektu

Zawartość wygenerowanego projektu różni się nieco w zależności od wersji MVC i wykorzystanego środowiska, jednak kluczowe elementy są takie same. W naszym przypadku mamy:

`<projekt> \ Controllers`
katalog zawierający kontrolery, kliknięcie prawym przyciskiem myszy na Controllers, wybranie „Add” z menu kontekstowego prowadzi nas m.in. do polecenia dodania nowego kontrolera („Controller…");

`<projekt> \ Views`
katalog zawierający widoki; widoki powiązane z kontrolerem umieszczane są w podkatalogu o nazwie zgodnej z nazwą kontrolera; dodatkowo mamy podkatalog Shared przeznaczony na wspólne widoki lub ich fragmenty;

`<projekt> \ Views \ _ViewStart.cshtml`
jest to „główna” strona naszej aplikacji, jednak zawiera tylko odwołanie do Views\Shared_Layout.cshtml;

`<projekt> \ Views \ Shared \_Layout.cshtml`
jest tu umieszczony szablon naszej strony;

`<projekt> \ Models`
miejsce przeznaczone na nasze modele;

`<projekt> \ Program.cs`
tu umieszczona jest funkcja Main aplikacji, jednak całość kodu konfigurującego znajduje się w pliku Startup.cs;

`<projekt> \ Startup.cs`
plik zawierający konfigurację naszej aplikacji;

`<projekt> \ wwwroot`
miejsce na statyczne zasoby typu JavaScript, obrazki, CSS.

Aplikacja ASP.NET Core ma budowę modułową. Poszczególne funkcje dostępne są w postaci usług. Na przykład, aby skorzystać z sesji HTTP, musimy dodać do aplikacji usługę Session w funkcji `ConfigureServices` w pliku `Startup.cs` dodając:

```
services.AddSession();
```

oraz użyć jej w funkcji `Configure` w tym samym pliku, dopisując:

```
app.UseSession();
```

Dodając lub włączając usługę czasem trzeba ją skonfigurować. Robimy to przekazując parametry do funkcji dodającej lub włączającej usługę.

W utworzonym projekcie domyślnie użyte są usługi:

- MVC,
- HttpsRedirection (o ile wybraliśmy konfguracę dla protokółu HTTPS), która przekierowuje żądania HTTP na HTTPS,
- StaticFiles hostująca pliki statyczne (katalog wwwroot),
- CookiePolicy zajmująca się wyskakiwaniem okienek ze zgodą na przechowywanie ciasteczek,
- DeveloperExceptionPage lub ExceptionHandler – w zależności od środowiska (kompilacja Debug lub Release) wyjątki obsługiwane są przez stronę wyświetlającą ich pełną zawartość lub przez wskazany widok.

Architektura MVC

Jak sama nazwa wskazuje, MVC bazuje na trzech typach komponentów: modelach, widokach i kontrolerach. Modele reprezentują logikę naszej aplikacji. Często albo komunikują się z bazą danych albo reprezentują dane przechowywane w bazie. Widoki są prezentowane użytkownikowi. Nie powinny zawierać logiki aplikacji (tym zajmują się modele). Ich celem jest jedynie prezentacja danych użytkownikowi. Kontrolery to obiekty reagujące na polecenia użytkownika. Komunikują się z modelami i decydują, jakie widoki zaprezentować użytkownikowi. Typowy przepływ sterowania wygląda następująco. Działanie użytkownika trafia do kontrolera. Ten przekazuje sterowanie do odpowiedniego modelu, który wykonuje wydane polecenie. Rezultat, najczęściej reprezentowany przez jakiś model, wraca do kontrolera, który przekazuje go do odpowiedniego widoku. Widok jest wykonywany i prezentowany użytkownikowi.

Istotą MVC jest minimalizacja powiązań pomiędzy poszczególnymi komponentami. Interfejs użytkownika (widoki) nie musi nic wiedzieć o logice aplikacji – otrzymuje tylko dane do wyświetlenia w formie modeli i pozwala skorzystać użytkownikowi z akcji udostępnianych przez kontrolery. Logika aplikacji (modele) nie ma żadnego związku z UI. Jej zadaniem jest obsługa poleceń wydawanych przez kontrolery i odpowiadanie na nie danymi. Kontrolery z kolei sprowadzają się do przekazywania danych: z akcji użytkownika do wołania odpowiedniej metody i z wyniku działania modelu do widoku.

W przypadku ASP.NET MVC obraz ten uzupełnia jeszcze routing. Jest on umieszczony pomiędzy działaniem użytkownika a kontrolerami i jego celem jest wskazanie kontrolera, który obsłuży wydane polecenie.

Routing

Rolą routingu jest wskazanie, na podstawie adresu zasobu z zapytania HTML, kontrolera i akcji, która ma obsłużyć dane zapytanie. Wskazania te konfigurujemy poprzez zdefiniowanie tras podczas startu aplikacji. Trasa jest szablonem opisującym grupę adresów zasobów. Podczas obsługi zapytania trasy dopasowywane są do adresu w kolejności ich definiowania. Jeżeli trasa zostanie dopasowana (czyli adres będzie zgodny z szablonem), wówczas decyduje ona o kontrolerze i podjętej akcji. Jeżeli adres nie pasuje do szablonu, testowana jest kolejna trasa itd. Jeżeli żadna trasa nie zostanie dopasowana, rezultatem jest kod HTTP 404. Adresem z punktu widzenia MVC i tras jest część URI po adresie aplikacji, bez parametrów i fragmentu, zatem gdy w pole adresu przeglądarki wpiszemy

```
http://localhost:12345/Users/Add?name=user&pass=pwd#fragment
```

do routingu trafi fragment

`Users/Add`

Szablon trasy jest napisem i przypomina nieco wyrażenie regularne. Zwykły tekst wymaga obecności podanych znaków, przy czym wielkość liter jest ignorowana. Np.:

szablon	**pasujący adres**
Home/Index	home/index
	Home/index
dodaj-a,b	dodaj-A,B

Fragmenty szablonu otoczone nawiasami klamrowymi oznaczają parametry. Napis w nawiasach klamrowych jest nazwą parametru. O ile nie ma nałożonych dodatkowych ograniczeń, parametr pasuje do dowolnego, niepustego napisu,. Np.:

szablon	**pasujący adres**	**parametry**
{controller}/{action}	Home/Index	controller=Home, action=Index
dodaj-{a},{b}	dodaj-ABC,DEF	a=ABC, b=DEF
plik/{nazwa}/{format}	plik/obraz.txt/doc	nazwa=obraz.txt, format=doc

ale:

szablon	**niepasujący adres**
{controller}/{action}	Home/Index/parametr
dodaj-{a},{b}	dodaj-a,
plik/{nazwa}/{format}	plik//
plik/{nazwa}/{format}	plik/obraz.txt
plik/{nazwa}/{format}	plik/obraz.txt/

Jeżeli poprzedzimy nazwę parametru gwiazdką oznacza to, że dany parametr pasuje do całej reszty. Taki parametr może pojawić się tylko na końcu szablonu, np.:

szablon	**pasujący adres**	**parametry**
plik/{*nazwa}	plik/etc/passwd	nazwa=etc/passwd

Pomiędzy dwoma parametrami musi znajdować się co najmniej jeden znak nie będący parametrem. Np. szablon postaci „strona/{controller}{action}” nie jest poprawny. Szablon nie może rozpoczynać się od znaków ~ i / oraz nie może zawierać znaku ?. Nie ma sensu stosować w szablonie znaków # oraz &. # zostanie usunięty z adresu jako początek fragmentu, chyba że zastąpimy go przez %23 (wtedy zostanie dopasowany do # w szablonie trasy). Pojawienie się znaku & (lub %26) przed listą parametrów w adresie skutkuje wyjątkiem „Potencjalnie niebezpieczna wartość pochodząca z klienta”, mimo że w szablonie trasy & jest dozwolony.

W przypadku udanego dopasowania adresu do szablonu wynikiem jest zestaw parametrów Co najmniej dwa muszą zostać zdefiniowane. Są to:

`controller` – określa nazwę kontrolera, który ma obsłużyć zapytanie; będzie to klasa, której nazwą jest wartość dla klucza `controller` z doklejonym napisem `Controller`; np. dla controller=User sterowanie zostanie przekazane do klasy `UserController`;

`action` – określa nazwę akcji w kontrolerze, która ma zostać wykonana.

Wielkość liter w nazwie kontrolera i akcji jest ignorowana. Jeżeli mamy akcję `Index` w klasie `HomeController` i trasę o szablonie „{controller}/{action}” to każdy z adresów Home/Index, HOME/INDEX, HoMe/InDeX czy home/index zaprowadzi nas w to akcji `Index` w kontrolerze `Home`. Z drugiej strony, jeżeli kontroler `Home` będzie miał zarówno akcję `Index` jak i `INDEX`, wynikiem będzie wyjątek zgłoszony w trakcie próby wykonania tej akcji. Należy pamiętać, że routing nie sprawdza istnienia kontrolera i akcji. Jeżeli z trasy wyniknie nieistniejąca klasa kontrolera lub metoda (akcja) trasa zostanie dopasowana ale próba wykonania akcji skończy się wyjątkiem. Pozostałe parametry będą źródłem danych dla akcji.

Trasy możemy definiować na dwa sposoby. Pierwszy z nich to opisanie trasy w metodzie `Configure` w pliku `Startup.cs`. W funkcji tej do metody `UseMvc` przekazujemy opis tras w postaci funkcji. Funkcja ta jako parametr dostaje obiekt o nazwie `routes` i woła jego metodę `MapRoute` przekazując do niej następujące parametry:

- `name` – nazwa trasy – każda trasa powinna mieć inną nazwę,
- `template` – szablon URL-a – to właśnie ten napis będzie dopasowywany do adresu z zapytania,
- opcjonalnie: `defaults` – domyślne wartości parametrów,
- opcjonalnie: `constraints` – ograniczenia, o nich później.

Parametr `defaults` określa wartości parametrów, które nie wynikają z dopasowania szablonu. Jeżeli np. chcemy, aby szablon „Get/Plik/{nazwa}” prowadził do kontrolera `Files` i akcji `GetFile`, możemy (a nawet musimy, gdyż każda trasa musi definiować parametr `controller` i `action`) podać nazwę kontrolera i akcji w wartościach domyślnych, np.:

```
routes.MapRoute(“trasa”, “Get/Plik/{nazwa}”,
                new { controller = “Files”, action = “GetFile”});
```

Jeżeli rozdzielamy fragmenty adresu znakiem / to podając wartości domyślne możemy korzystać ze skróconych adresów, pomijając parametry z prawej strony adresu. Np. dla trasy domyślnie zdefiniowanej w nowym projekcie:

```
routes.MapRoute(“Default”, “{controller}/{action}/{id}”,
                new { controller = “Home”, action = “Index”,
                      id = UrlParameter.Optional } );
```

szablon pasuje do każdej z poniższych tras:

adres	**dopasowane parametry**
	controller=Home, action=Index
Users	controller=Users, action=Index
Users/get	controller=Users, action=get
Users/get/123	controller=Users, action=get, id=123
Users//321	nie zostanie dopasowane, jeżeli chcemy pominąć action musimy też pominąć id

`UrlParameter.Optional` oznacza, że dany parametr nie ma mieć wartości jeżeli nie został podany.

Zamiast podawać słownik wartości domyślnych, możemy przekazać je bezpośrednio w opisie trasy, podając wartość domyślną po nazwie parametru i znaku równości, np.:

```
{controller=Home}/{action=Index}/{id?}
```

znak zapytania oznacza tu parametr domyślny.

Należy pamiętać, że trasy dopasowywane są w kolejności dodania, uwzględniając wartości domyślne, więc dla konfiguracji:

```
routes.MapRoute(“Default”, “{controller}/{action}/{id}”,
          new { controller = “Home”, action = “Index”, id = 12 } );
routes.MapRoute(“trasa”, “Ludzie/Lista”,
          new { controller = “Users”, action = “List”});
```

zapytanie o Ludzie/Lista wywoła metodę `Lista` z parametrem `id=12` z klasy `LudzieController` – zostanie dopasowana pierwsza trasa, natomiast w przypadku konfiguracji:

```
routes.MapRoute(“trasa”, “Ludzie/Lista”,
          new { controller = “Users”, action = “List”});
routes.MapRoute(“Default”, “{controller}/{action}/{id}”,
          new { controller = “Home”, action = “Index”, id = 12 } );
```

zapytanie o Ludzie/Lista wywoła metodę `List` (bez parametrów) z klasy `UsersController`. Jeżeli rozdzielimy parametry w szablonie innym znakiem niż ukośnik, nie będziemy mogli skorzystać z wartości domyślnych.

Parametr `constraints` metody `MapRoute` pozwala nam nałożyć ograniczenia na wartości parametrów. Jeżeli po dopasowaniu trasy ograniczenia nie będą spełnione, trasa nie będzie dopasowana i zostanie podjęta próba dopasowania innych tras. Na przykład, jeżeli z dopasowania wynika, że parametr `p1` ma wartość `“abcd”`, a ograniczenie mówi, że `p1` może zawierać tylko cyfry, takie dopasowanie zostanie odrzucone. Wartością `constraints` jest obiekt z polami postaci *nazwa_parametru=ograniczenie. ograniczenie* może być napisem, wtedy opisuje on wyrażenie regularne i sprawdzana jest zgodność wartości parametru z wyrażeniem. Ograniczenie może być również obiektem, który implementuje interfejs `IRouteConstraint`. Podczas dopasowania wykonywana jest metoda `Match` tego obiektu i jeżeli zwraca ona `true`, parametr jest dopasowany, jeżeli `false` – nie jest. Np.:

```
routes.MapRoute(“dodawanie”, “Dodaj/{p1},{p2}”,
               new { controller = “Adder”, action = “Dodaj” },
               new { p1 = @“\d+”, p2 = @“\d+”} );
```

definiuje trasę, której parametry mogą być tylko liczbami (\d to to samo co [0123456789], @ przed napisem oznacza wyłączenie specjalnego znaczenia znaku \), zaś:

```
using Microsoft.AspNetCore.Routing;
...
public class Div10Test: IRouteConstraint {
   public bool Match(HttpContext httpContext,
                     IRouter route,
                     string routeKey,
                     RouteValueDictionary values,
                     RouteDirection routeDirection) {
     if(Int32.TryParse(values[routeKey].ToString(), out int v)) {
          return v % 10 == 0;
          }
     return false;
     }
```

```
...
routes.MapRoute(“dodawanie2”, “Dodaj/{p1},{p2}”,
     new { controller = “Adder”, action = “Dodaj” },
     new { p1 = @“\d+”, p2 = new Div10Test()} );
```

definiuje podobną trasę, przy czym tu `p2` musi być liczbą całkowitą podzielną przez 10. Parametry do `Match` to:

`httpContext`	– obiekt definiujący kontekst HTTP zapytania,
`route`	– obiekt trasy, której dotyczy ograniczenie,
`routeKey`	– nazwa parametru, który jest testowany (np. `p1`, `p2`),
`values`	– słownik wartości parametrów trasy, czyli np. `values[“p1”]` jest wartością parametru `p1` (jako `object`, więc trzeba jeszcze wziąć `ToString()`),
`routeDirection`	– wartość opisująca, czy wartość parametru pochodzi z adresu zapytania, czy z generowanego adresu.

Drugi sposób definiowania tras wykorzystuje atrybuty. Do definiowania tras korzystamy z atrybutu `Route` np.:

```
[Route(“/uzytkownicy”]
public class UsersController {
   ...
   [Route(“Dodaj/{login}”]
   public IActionResult AddUser(string login) { ... }
   ...
   }
```

spowoduje, że akcja `AddUser` będzie dostępna pod adresem pasującym do szablonu „uzytkownicy/Dodaj/{login}”. Trasy akcji łączone są z trasą kontrolera znakiem /. Jeżeli trasa akcji rozpoczyna się od znaku /, wtedy nie jest dołączana do niej trasa kontrolera. W szablonie trasy możemy skorzystać z parametrów oraz ograniczeń dla nich. Ograniczenia dodajemy tu do parametrów po dwukropku, np.:

ograniczenie	**pasująca wartość**
`param:alpha`	– ciąg małych i wielkich liter alfabetu łacińskiego
`param:bool`	– wartość logiczna
`param:datetime`	– wartość typu DateTime
`param:decimal`	– wartość dziesiętna
`param:double`	– liczba typu double
`param:float`	– liczba typu float
`param:guid`	– guid
`param:int`	– liczba typu int
`param:length`	– napis określonej długości, np. `length(3)` lub `length(3, 10)` dla zakresu 3–10
`param:long`	– liczba typu long
`param:max`	– liczba całkowita nie przekraczająca podanej wartości np. `max(100)`
`param:maxlength`	– napis o długości nie większej niż podana, np. `maxlength(32)`
`param:min`	– liczba całkowita nie mniejsza niż podana wartość, np. `min(3)`

ograniczenie	**pasująca wartość**
`param:minlength`	– napis o długości co najmniej takiej jak podana, np. `minlength(2)`
`param:range`	– liczba całkowita z przedziału, np. `range(0, 100)`
`param:regex`	– napis pasujący do wyrażenia regularnego, np. `regex(^\d+$)`

Kilka ograniczeń podajemy po kolejnych dwukropkach, np. `param:int:min(a)`. Dodanie znaku zapytania po ograniczeniach (np. `param?`, `param:int?`) powoduje, że parametr jest opcjonalny. Jeżeli chcemy trasie nadać nazwę, robimy to ustawiając parametr `Name` w atrybucie `Route`. Przykład trasy z ograniczeniem i nazwą:

```
Route(“Lista/{dzien:regex(^\d\d-\d\d-\d\d\d\d$)?}”, Name=“lst”]
```

Kontroler

Kontroler jest zwykłą klasą, dziedziczącą po klasie `Controller`. Jedynym dodatkowym wymaganiem jest zakończenie jej nazwy na `Controller`, np. `KoszykController`. W takim przypadku mówimy o kontrolerze o nazwie `Koszyk` i tę nazwę podajemy w różnych miejscach (podczas routingu, wskazując docelowy kontroler itp.), jednak cały czas chodzi nam o klasę o nazwie `KoszykController`. Kontroler jest powoływany do życia na nowo przy obsłudze każdego zapytania, zatem jakikolwiek stan zapamiętany w tym obiekcie zostanie utracony po zakończeniu obsługi zapytania.

Akcja

Akcją jest każda publiczna metoda, należy zatem uważać na konfigurację routingu i publiczne metody w kontrolerze, aby nie udostępnić za dużo. Akcja musi spełniać jednak kilka warunków:

- nie powinna być statyczna ani przeciążona (poza pewnymi wyjątkami, ale o nich później),
- nie powinno być dwóch akcji różniących się wielkością liter w nazwie,
- akcjami nie są metody odziedziczone po klasach bazowych,
- akcje nie mogą posiadać parametrów wyjściowych i referencyjnych (słowa kluczowe `out` i `ref`).

Jeżeli nie chcemy, aby metoda była akcją, możemy zablokować to, nadając tej metodzie atrybut `[NonAction]`.

Parametry akcji (parametry metody) są pobierane z parametrów wynikających z trasy (zob. routing) oraz parametrów zapytania (parametry metod GET i POST protokołu HTTP). Parametry zostaną powiązane według nazw, tzn. parametr formularza o nazwie login zostanie włożony do parametru akcji o nazwie login. Miarę możliwości wartości zostaną dopasowane do typów parametrów akcji: np. napis będzie skonwertowany na liczbę czy datę. Niemożność dopasowania typu parametru skutkuje wyjątkiem, podobnie jak brak parametru wymaganego przez akcję (chyba, że parametr ma wartość domyślną lub jest opcjonalny a jego typ w metodzie może przyjmować wartości null czyli albo jest referencją albo typem opcjonalnym np. `int?`). Proszę zwrócić uwagę, że trasa która prowadzi do akcji lecz definiuje za mało parametrów zostanie dopasowana, a wyjątek wyskoczy dopiero podczas próby wywołania akcji. Z drugiej strony, nadmiarowe parametry będą po cichu odrzucanie i nie spowodują żadnego błędu.

Wartość zwrócona z akcji definiuje dalsze działanie aplikacji. Zazwyczaj jest to obiekt typu `IActionResult` i zazwyczaj jest to wynik wołania jednej z poniższych metod:

`View()`
polecenie wyświetlenia użytkownikowi widoku o takiej samej nazwie jak akcja (czyli Views \

nazwa_kontrolera \ *nazwa_akcji*);

`View(`*`nazwa_widoku`*`)`
polecenie wyświetlenia użytkownikowi widoku o podanej nazwie (czyli Views \ *nazwa_kontrolera* \ *nazwa_widoku*);

`RedirectToAction(`*`nazwa_akcji, nazwa_kontrolera,`*
`[`*`opcjonalne_parametry_trasy`*`])`
przekierowanie użytkownika do akcji *`nazwa_akcji`* w kontrolerze *`nazwa_kontrolera`*; jest to przekierowanie na poziomie HTTP (kod 30x) więc przekierowane zapytanie ponownie przechodzi przez routing; adres zostanie automatycznie wygenerowany na podstawie zdefiniowanych tras;

`Redirect(`*`url`*`)`
przekierowanie pod podany adres;

`Content(`*`tekst`*`)`
zwrócenie tekstu *`tekst`* jako wynik, zwrócenie z akcji nieobsługiwanego przez `ActionResult` typu (np. `string`) spowoduje jego opakowanie w `Content`;

`File(`*`ścieżka`*`, [`*`mime_type`*`, [`*`nazwa`*`]])`
`File(`*`zawartość`*`, `*`mime_type`*`, [`*`nazwa`*`])`
zwrócenie pliku o typie mime *`mime_type`* i podanej nazwie; ścieżka określa z jakiego pliku na dysku ma pochodzić zawartość; zawartość to bezpośrednio podana zawartość pliku (tablica bajtów);

`Json(`*`obiekt`*`)`
zwrócenie obiektu zakodowanego zgodnie z formatem Json.

Do komunikacji z widokiem służą obiekty:

`ViewData` – słownik z kluczami będącymi napisami,
`TempData` – słownik z kluczami będącymi napisami; różnica między `ViewData` a `TempData` polega na czasie życia; o ile `ViewData` jest czyszczone przed każdą akcją, to zawartość `TempData` zostaje zapamiętana podczas przekierowań (ale przejście do innej akcji bez przekierowania wyczyści zawartość `TempData`),
`ViewBag` – obiekt dynamiczny o podobnym zastosowaniu co `ViewData`;
`Model` – obiekt o dowolnej zawartości.

Więcej informacji o komunikacji pomiędzy akcją a widokiem znajduje się w sekcji „Widok”.

Akcje mogą odpowiadać na różne rodzaje zapytań. Domyślnie akcja odpowiada na zapytanie typu GET. Aby to zmienić, musimy nadać akcji jeden z atrybutów: `HttpDelete`, `HttpHead`, `HttpOptions`, `HttpPatch`, `HttpPost`, `HttpPut`. Możemy także skorzystać z `HttpGet` który jest równoważny domyślnemu zachowaniu. Obecność jednego z wymienionych atrybutów pozwala nam przeciążać nazwy akcji. Mamy możliwość utworzenia dwóch akcji o tej samej nazwie, różniących się metodą zapytania. Często korzystamy z tej możliwości podczas obsługi formularzy. Jeżeli chcemy, by był on obsługiwany przez akcję Formularz to implementacja ma często postać:

```
IActionResult Formularz() {        // akcja wyświetlająca formatkę
     return View();
     }
```

```
[HttpPost]
IActionResult Formularz(...) {     // akcja przetwarzająca dane
     // obsługa danych             // z formularza
     }
```

Do dyspozycji mamy również atrybut `RequireHttps` który wymusza protokół HTTPS podczas korzystania z danej akcji.

Widok

Widok jest stroną HTML-ową ze wstawkami zawierającymi kod wykonywany po stronie serwera. Dla danej akcji możemy łatwo dodać odpowiadający jej widok klikając prawym przyciskiem myszy na nazwie akcji w jej definicji w oknie edytora i z menu kontekstowego wybierając „Add View", co umieści widok w odpowiednim podkatalogu (jeżeli projekt się nie kompiluje operacja ta może nie zadziałać poprawnie). Dodać widok możemy także klikając w drzewie projektu prawym przyciskiem myszy na Views \ *nazwa_kontrolera* i wybierając Add -> View z menu kontekstowego, tu jednak sami musimy uważać, aby umieścić go w odpowiednim podkatalogu.

Okno dialogowe pozwalające tworzyć widok daje nam kilka opcji do skonfigurowania:

Nazwa	– nazwa widoku.
Szablon	– szablon widoku – jeżeli naszym celem jest utworzenie typowej formatki do tworzenia, edycji czy listowania obiektów pewnej klasy możemy wybrać odpowiedni szablon oraz wskazać tę klasę w polu „Klasa modelu"; spowoduje to wygenerowanie odpowiedniej formatki na podstawie pól klasy,
Klasa modelu	– klasa, której dotyczy szablon,
Widok częściowy	– decyzja, czy widok ma być częściowy; częściowe widoki opisują fragmenty strony i ułatwiają nadanie stronie struktury, jest to coś, co pojawia się na każdej stronie i zawiera pewną logikę, niepowiązaną jednak z konkretną zawartością strony; przykładowym częściowym widokiem może być boczne menu, fragment strony obsługujący logowanie, czy sekcja koszyka,
Odwołanie do bibliotek skryptów	– pytanie, czy widok ma odwoływać się do gotowych skryptów w projekcie; jeżeli chcemy skorzystać z walidacji po stronie klienta lub zdalnej musimy pamiętać o włączeniu tej opcji,
Strona układu	– jeżeli chcemy skorzystać z innego niż domyślny szablonu strony (Views \ _ViewStart.cshtml), ta opcja pozwala nam go wskazać.

Widoki są interpretowane przez silnik Razor. Wymaga on, aby kod rozpoczynał się od znaku @. W przypadku pojedynczych wyrażeń, ich wartość wstawiana jest bezpośrednio do treści strony np.:

```
<span>wynik = @score</span>
```

Skomplikowane wyrażenia należy otaczać nawiasami:

```
wynik = @(a+b)
```

Aby wygenerować treść strony wewnątrz wyrażenia musimy posłużyć się tagami HTMLowymi:

```
@for(int i = 0; i < 10; i++) {
   <span>wartość i=@i <br></span>
   }
```

Jeżeli odpowiedni tag nie wynika z konstrukcji strony, możemy posłużyć się tagiem <text> np.:

```
@if(a < b) {
   <text>a jest mniejsze od b</text>
} else {
   <text>a jest niemniejsze od b</text>
   }
```

Możemy także skorzystać z `@:` oznaczającego „`<text>` stąd do końca linii”, np.:

```
@if(a < b) {
   @:a jest mniejsze od b
} else {
   @:a jest niemniejsze od b
   }
```

Dłuższe fragmenty kodu możemy otoczyć klamerkami:

```
@{
   var zmienna = 3;
   var inna_zmienna = zmienna.ToString();
}
```

Znaczenie znaku @ zazwyczaj będzie wywnioskowane z kontekstu, np. w napisie adres@email.pl @ nie zostanie potraktowane jako znak rozpoczynający wyrażenie, o ile email nie jest zmienną; jednak jeżeli chcemy mieć pewność, możemy napisać adres@@email.pl, co nie pozostawi wątpliwości co do znaczenia znaku @.

Razor domyślnie podmienia znaki specjalne HTMLa. Wynikiem:

```
@“<b><i>text</i></b>”
```

będzie widoczne na stronie

text

Aby wstawić do HTMLa dokładną wartość napisu należy skorzystać z funkcji `Html.Raw`:
`@Html.Raw(“<b><i>text</i></b>”)`

czego wynikiem będzie:
text

Widok zazwyczaj definiuje tylko pewne fragmenty strony, np. główną treść czy stopkę zaś cała „otaczająca” zawartość (menu, nagłówek, układ strony itp.) zdefiniowany jest we wspólnym dla wszystkich widoków szablonie (domyślnie: `_Layout.cshtml`).

Sekcje widoków

Widoki możemy podzielić na sekcje. Sekcja to po prostu nazwany fragment widoku. Ta część widoku, która nie jest przypisana do żadnej sekcji jest ciałem widoku. Treść ciała widoku wstawiana jest przez wywołanie funkcji `RenderBody` natomiast zawartość sekcji wstawiamy wołając

```
RenderSection(nazwa)
```

lub

```
RenderSection(nazwa, wymagane)
```

gdzie *`nazwa`* jest nazwą sekcji do wyświetlenia zaś wymagane określa, czy sekcja musi istnieć. Jeżeli *`wymagane`* jest równe `true` a sekcja jest niezdefiniowana, wystąpi wyjątek. Jeżeli *`wymagane`* jest fałszywe i sekcja nie jest zdefiniowana, `RenderSection` rozwinie się w pusty ciąg. Pierwsza wersja wołania `RenderSection` jest równoważna wołaniu `RenderSection(nazwa, true)`.

Zarówno ciało widoku jak i każdą sekcję można wyświetlić co najwyżej raz. Dwukrotne wywołanie `RenderBody` lub `RenderSection` z tą samą nazwą skutkuje wystąpieniem wyjątku.

Sekcje w widokach definiujemy otaczając odpowiedni fragment nawiasami klamrowymi, poprzedzonymi dyrektywą:

```
@section nazwa
```

np.:

widok
```
@section Menu {
<li>opcja</li>
<li>opcja2</li>
}
<b>lorem</b>
ipsum dolor
```

layout
```
<ul>
@RenderSection('menu')
</ul>
<div>
@RenderBody()
</div>
```

wynik
```
<ul>
<li>opcja</li>
<li>opcja2</li>
</ul>
<div>
<b>lorem</b>
ipsum dolor
</div>
```

Komunikacja akcji z widokiem

Komunikację pomiędzy akcją a widokiem zapewniają cztery obiekty: `ViewData`, `ViewBag`, `TempData` oraz `Model`. `ViewData` jest słownikiem o kluczach typu string i wartościach typu `object`. Możemy zatem włożyć do niego cokolwiek, lecz odczytując zawartość zazwyczaj musimy dokonać odpowiedniej konwersji. `ViewBag` jest obiektem dynamicznym, zachowuje więc typy ustawionych właściwości, np.:

```
IActionResult Akcja() {
   ViewData[“a”] = 1;
   ViewData[“b”] = 2;
   ViewBag.c = 3;
   ViewBag.d = 4;
   return View();
   }
```

```
...
@ViewData[“a”] + @ViewData[“a”] =
          <b>@((int)ViewData[“a”] + (int)ViewData[“b”])</b><br>
@ViewBag.c + @ViewBag.d = <b>@(ViewBag.c + ViewBag.c)</b><br>
```

Zawartość `ViewData` i `ViewBag` jest wspólna – klucze ustawione w `ViewData` są widoczne przez `ViewBag` i odwrotnie.

Obiekt `TempData` jest bardzo podobny do `ViewData`. Różni się od niego brakiem skojarzenia z `ViewBag` oraz czasem życia. `ViewData` jest czyszczone przed każdym wykonaniem akcji. Zawartość `TempData` nie jest czyszczona przy przekierowaniach.

Model jest obiektem przekazanym jako dodatkowy parametr do wywołania `View()` w akcji. To co włożymy do `View` w parametrze model, zobaczymy w widoku w zmiennej `Model`. Domyślnie Model jest typu `object` więc musimy rzutować go na oczekiwany typ. Możemy także powiązać widok z typem modelu tworząc silnie typowany widok. Robimy to umieszczając na początku widoku dyrektywę:

```
@model nazwa_typu
```

Powoduje ona, że obiekt Model w widoku będzie typu *`nazwa_typu`*. Jeżeli tworzymy model na podstawie szablonu, dyrektywa ta jest automatycznie dodawana do widoku ze wskazanym w oknie dialogowym typem. Widok nie może modyfikować modelu.

Prosty przykład skorzystania z modelu możemy zobaczyć poniżej:

```
IActionResult Details(int userId) {
   MVCApp.Models.User user = db.GetUserById(userId);
   return View(user);
   }
...
@model MVCApp.Models.User
Login: @Model.login<br>
E-mail: @Model.email<br>
```

jeżeli nie użylibyśmy dyrektywy `@model`, widok musiałby mieć postać:

```
Login: @(Model as MVCApp.Models.User).login<br>
E-mail: @(Model as MVCApp.Models.User).email<br>
```

Typ modelu może być taki, jak sobie zażyczymy i najlepiej wybrać ten, który jest najbardziej wygodny, np.:

```
IActionResult ListUsers() {
   IEnumerable<MVCApp.Models.User> users = db.ListAllUsers();
   return View(users);
   }
...
@model IEnumerable<MVCApp.Models.User>
@foreach(var user in Model) {
   ...
   }
```

Widoki są kompilowane na żądanie. Modyfikacja widoku nie wymaga zatem przebudowy całego projektu. Wystarczy przeładować stronę i poprawka powinna być od razu widoczna.

Jeżeli w widoku musimy użyć przestrzeni nazw, możemy skorzystać z dyrektywy:

```
@using namespace
```

np.

```
@using MVCApp.Models.Database
```

Model

Model jest zwykłą klasą. Zazwyczaj przechowujemy je w katalogu Models. Dodać nowy możemy klikając na Models prawym przyciskiem myszy i wybierając „Add” następnie „New item” oraz „Class”. W starszych wersjach MVC/Visual Studio po modyfikacji modelu warto przebudować projekt, aby wszelkie zmiany zostały zauważone przez środowisko (np. aby były wyświetlane aktualne nazwy obiektów w okienkach z podpowiedziami).

Dane w modelach powinny być właściwościami (nie polami); inaczej środowisko może ich nie zauważyć podczas generowania np. formularzy.

Czasem zachodzi konieczność skonfigurowania modelu. Chcielibyśmy np. zabronić pokazywania użytkownikowi zawartości pewnych pól czy ustalić nazwę w formularzu dla danego pola. Możemy to zrobić nadając polom modelu atrybuty. Jednym z nich jest atrybut `Display` (`System.ComponentModel.DataAnnotations`). Pozwala on manipulację „wyglądem” właściwości. Możemy, między innymi, ustawić nazwę elementu pokazywaną w formularzach poprzez parametr `Name` atrybutu (`Display(Name = ...)`) i kolejność (parametr `Order`). Istnieje podobny do niego atrybut `DisplayName`, także umożliwiający ustawienie nazwy, ale preferowane jest użycie `Display`. Umożliwia on, w przeciwieństwie do `DisplayName`, tłumaczenie komunikatów.

Atrybut `HiddenInput` (`Microsoft.AspNetCore.Mvc`) pozwala na decydowanie, jak pole modelu ma być widoczne dla użytkownika. Parametr `DisplayValue = true` określa, że w silnie typowanych widokach tylko prezentujących dane wyświetlana będzie wartość pola. Wartość `false` mówi, że w silnie typowanych widokach tylko prezentujących dane pole ma być polem ukrytym. Niezależnie od `DisplayValue`, w silnie typowanych widokach pozwalających na edycję danych pole z atrybutem `HiddenInput` zawsze występuje jako pole hidden.

```
public class ProductViewModel {
     [HiddenInput] // równoważne [HiddenInput(DisplayValue=true)]
     public int Id { get; set; }
     [Display(Name = “Nazwa użytkownika”)]
     public string Name { get; set; }
     [HiddenInput(DisplayValue = false)]
     public long TimeStamp { get; set; }
     }
```

Sesja

Ponieważ kontroler nie może przechowywać stanu, potrzebujemy innej metody powiązania danych z klientem. Tak jak w większości innych środowisk, także tu dane takie przechowujemy w sesji. W MVC sesję reprezentuje obiekt o nazwie `Session`.

Sesja dostępna przez `HttpContext.Session` po dołączeniu przestrzeni nazw `Microsoft.AspNetCore.Http`. We wcześniejszych wersjach .Net Core potrzebne też było doinstalowanie pakietu `Microsoft.AspNetCore.Session` (poprzez NuGet). W .Net Core

2.1 pakiet ten jest dodawany w razem z metapakietem `Microsoft.AspNetcCore.App`. Następnie należy dodać w metodzie `ConfigureServices` w pliku `Startup.cs` fragment:

```
services.AddDistributedMemoryCache();
services.AddSession(options => {
     // tak można ustawić timeout sesji
     options.IdleTimeout = TimeSpan.FromMinutes(10);
     // tu wymuszamy ciastka typu HttpOnly
     options.Cookie.HttpOnly = true;
     });

// poniższa wartość false wyłącza wyskakujące okienka o zgodę na
// ciastka
services.Configure<CookiePolicyOptions>(options => {
               options.CheckConsentNeeded = context => false;
               options.MinimumSameSitePolicy = SameSiteMode.None;
               });
...
services.AddMvc(...);
```

Pierwsza linia określa, że dane sesji będą przechowywane w pamięci (możliwe są także inne miejsca, np. baza danych), druga dodaje usługę sesji, ustawiając jednocześnie timeout sesji na 10 minut (sesja wygaśnie po 10 minutach braku aktywności klienta) i typ ciastek sesji HttpOnly. Trzecia linia zmienia domyślne zachowanie ciasteczek (gdzie przechowywany jest identyfikator sesji) na niewymagający zgody użytkownika.

Na koniec metodę `Configure` w pliku `Startup.cs` musimy uzupełnić o:

```
app.UseSession();
...
app.UseMvc(...)
```

Sesja w .Net Core jest słownikiem, w którym klucze są napisami lecz może przechowywać tylko kilka podstawowych typów. Są to int, string i tablica bajtów. Wartości w sesji ustawiamy wołając `SetInt32`, `SetString` lub `Set`, zaś za dostęp do danych odpowiadają metody `GetInt32`, `GetString` i `Get`. Zapamiętanie bardziej złożonych obiektów wymaga zatem skonwertowania ich do obsługiwanych typów, na przykład przy pomocy:
`JsonConvert.DeserializeObject`
`JsonConvert.SerializeObject`
z przestrzeni nazw `Newtonsoft.Json`. Identyfikator sesji dostępny jest w polu `Id`. Metoda `Clear` czyści zawartość sesji.

Obiekt Request

W kodzie akcji dostępny jest obiekt Request. Reprezentuje on zapytanie przychodzące od klienta i przechowuje informacje zawarte w tym zapytaniu. Niektóre z jego pól to:

`Method`	– reprezentują metodę HTTP („GET”, „POST” itp.)
`Query`	– kolekcja parametrów zapytania (parametry GET),
`QueryString`	– kolekcja parametrów zapytania w postaci ciągu,
`Form`	– kolekcja parametrów zapytania (parametry POST),
`Form.Files`	– kolekcja załączonych plików; jest to słownik w którym nazwą jest nazwa pliku (z elementu input) zaś wartością obiekt reprezentujący

	plik z polami/metodami `ContentType`, `ContentDisposition`, `Length`, `FileName`, `CopyTo`, `OpenInputStream`, itd.
`Cookies`	– kolekcja zawierająca ciasteczka,
`Path`	– ścieżka zapytania (nie uwzględnia adresu serwera),
`ContentType`	– typ ciała zapytania,

Formularze

Jak już widzieliśmy wcześniej, obsługa formularza wymaga dwóch akcji. Pierwsza z nich reaguje na zapytanie typu GET i jest to zapytanie o stronę z formularzem. Druga reaguje na zapytanie typu POST i obsługuje dane wysłane w formularzu. Dostęp do danych w formularzu możemy zdobyć na kilka sposobów.

Pierwszy z nich to dodanie parametrów do akcji o nazwach takich samych, jak nazwy pól w formularzu (parametr `name` tagów). Jest to najprostsza metoda, ale przy większej ilości danych staje się problematyczna.

```
<form action=“/Test/Index” method=“POST”>
    <input type=“text” name="text” /><br />
    <input type=“text” name="value” /><br />
    <input type=“submit” />
    </form>
...
public IActionResult Index() { return View(); }
[HttpPost]
public IActionResult Index(string text, int value) { ... }
```

Drugi sposób to opisanie zawartości formularza modelem i dodanie parametru do akcji o typie taki jak nasz model. W tej sytuacji MVC utworzy nowy obiekt modelu, wypełni jego pola zawartością formularza łącząc pola modelu z parametrami formularza według ich nazwy i przekaże ten obiekt do akcji.

```
public class Data {
     public string text { get; set; }
     public int value { get; set; }
     }
...
[HttpPost]
public IActionResult Index(Data d) { ... d.text ... d.value ... }
```

Procesem wypełniania zawartości modelu możemy manipulować dodając atrybut `Bind` do parametru akcji. `Bind` z parametrem `include` określa, że tylko wymienione pola mają być wypełnione, np.:

```
IActionResult Akcja(
    [Bind(include:“nazwa,login,email”)]MVCApp.Models.Osoba o) ...
```

Parametr `Prefix` umożliwia nam rozróżnienie pól o takiej samej nazwie. Powoduje on, że pole *`pole`* modelu zostanie wypełnione parametrem *`prefix.pole`* z formularza, np.:

```
<input type=“text” name=“user1.nazwa”>
<input type=“text” name=“user2.nazwa”>
```

```
...
ActionResult Akcja(
              [Bind(Prefix=“user1”)]MVCApp.Models.User u1,
              [Bind(Prefix=“user2”)]MVCApp.Models.User u2)...
```

Proszę zwrócić uwagę, że wartość parametru `include` podjemy po dwukropku, zaś wartość `Prefix` po znaku równości.

Kolejna metoda obsługi formularzy to skorzystanie z obiektu `Request.Form`. Opisuje on dane z formularza, niezależnie od tego, czy zostały one przekazane przez parametry akcji. Dane w `Request.Form` przechowywane są jako napisy.

```
IActionResult Akcja(...) {
   var login = Request.Form[“login”];
   var password = Request.Form[“pass”];
   ...
   }
```

Taki sposób obsługi danych może spowodować pewien problem. Jeżeli zarówno akcja GETowa jak i POSTowa nie będą miały żadnych parametrów, kod nie skompiluje się. Będziemy mieli dwie metody o takiej samej nazwie i takiej samej sygnaturze. Możemy to rozwiązać np. dodając opcjonalny parametr do jednej z nich lub zmieniając wielkość liter nazwy jednej z metod. Możemy także skorzystać z czwartej metody.

Dane z formularza możemy także odebrać poprzez parametr akcji typu `IFormCollection` z przestrzeni nazw `Microsoft.AspNetCore.Http`. Jest to obiekt podobny do `Request.Form`. Przechowuje zawartość formularza w postaci słownika z kluczami i wartościami typu `string`.

```
IActionResult Akcja(IFormCollection form) {
   var login = form[“login”];
   var password = form[“pass”];
   ...
   }
```

Wszystkie te metody możemy łączyć ze sobą. Nic nie stoi na przeszkodzie, aby część parametrów odebrać w parametrach akcji zaś resztę wydobyć np. z `Request.Form`.

Upload plików

Aby wysłać plik na serwer, dane formularza muszą być odpowiednio kodowane. Do tagu `FORM` należy dodać parametr `enctype=“multipart/form-data”`. W MVC formularz zazwyczaj jest generowany helperem, więc ustawienie kodowania będzie miało postać:

```
@using(Html.BeginForm(“nazwa akcji”, “nazwa kontrolera”,
              FormMethod.Post,
              new { enctype=“multipart/form-data”})) { ... }
```

Następnie dodajemy do formularza możliwość wysyłania plików (`INPUT type=“file”`). Dzięki temu w Request.Files otrzymujemy kolekcję plików, np.:

```
byte[] data = new byte[1024];
// Request.Form.Files[name].ContentType, .Length, .CopyTo
var s = Request.Form.Files[name].OpenReadStream();
s.Read(data, 0, 1024);
string str = System.Text.Encoding.ASCII.GetString(data);
```

UWAGA: jeżeli plik jest przechowywany w modelu jako tablica bajtów, nazwa pliku w formularzu powinna być inna niż nazwa pola w modelu, inaczej MVC będzie próbowało przekazać jego zawartość w modelu, co się nie powiedzie i spowoduje rzucenie wyjątku.

Drugi sposób na odebranie pliku z formularza to dodanie do akcji parametru typu `IFormFile` o nazwie takiej, jak nazwa pliku w formularzu.

```
<input type="file" name="plik">
...
IActionResult Pliki(IFormFile plik) ...
```

Jest to dokładnie ten sam interfejs co w `Request.Form.Files`, korzystamy więc z tych samych metod. Jeżeli plik nie został podany, parametr będzie miał wartość `null`. Jeżeli chcemy przesłać grupę plików, możemy zrobić to w następujący sposób:

```
<input type="file" name="plik" multiple="multiple">
...
IActionResult Pliki(IEnumerable<IFormFile> plik) ...
```

Walidacja po stronie serwera

Walidacja po stronie serwera jest wspomagana przez obiekt `ModelState`. Reprezentuje on informację, czy działamy na poprawnym modelu, a jeżeli nie, dlaczego jest on niepoprawny. Posiada on wartość logiczną `IsValid`, która jest prawdziwa, gdy model jest poprawny i fałszywa, gdy pojawiły się jakieś błędy. Do dyspozycji mamy także metodę `AddModelError` z parametrami klucz (`string`) i wartość (`string`). Wywołanie jej oznacza, że okryliśmy w modelu błąd i chcemy opisać go parą klucz, wartość. Wywołanie to powoduje także, że `IsValid` będzie fałszywe.

Typowa walidacja po stronie serwera ma postać:

```
[HttpPost]
public ActionResult AddOsoba(MVCApp.Models.Osoba o) {
   if(o.name.Trim().Length < 3)
     ModelState.AddModelError("imie", "imię za krótkie");
   if(o.wiek < 18)
     ModelState.AddModelError("wiek", "za młody");
   if(!ModelState.IsValid) return View(); // dane do poprawki
   ... // model poprawny
   }
```

W akcji obsługi danych formularza (stąd atrybut `HttpPost`) odbieramy dane. Pierwsze dwa warunki sprawdzają, czy model jest poprawny. Jeżeli któryś z warunków nie jest spełniony, dodajemy błąd modelu z kluczem takim jak nazwa niepoprawnego pola i wartością opisującą błąd. Po zweryfikowaniu pól sprawdzamy czy wystąpiły błędy i jeżeli tak, odsyłamy użytkownika do widoku z formularzem (`return View()`).

Po stronie widoku, możemy skorzystać z następujących narzędzi:

`Html.ValidationSummary([excludePropertyErrors], [message])`
wyświetla nieuporządkowaną listę błędów (wartości z `AddModelError`), poprzedzając ją opcjonalnym komunikatem `message`; jeżeli `excludePropertyErrors` jest prawdziwe, lista nie uwzględnia błędów o kluczach których nazwy są takie same jak pola modelu;

`Html.ValidationMessage("klucz" [, "komunikat"])`
wyświetla komunikat, jeżeli dodano błąd z kluczem klucz; jeżeli komunikat nie jest podany, wyświetlana jest wartość ustawiona w `AddModelError`;

`Html.ValidationMessageFor(model => model.`*`pole`*`)`
wyświetla komunikat który dodano z kluczem takim samym jak nazwa podanego w *pole* pola obiektu.

Na przykład, dla przedstawionej wyżej walidacji:

```
@Html.ValidationSummary()
```

wyświetli:
- imię za krótkie
- za młody

```
@Html.ValidationMessage("wiek", "*")
```
wyświetli * jeżeli wiek był mniejszy niż 18

```
@Html.ValidationMessageFor(model => model.wiek)
```
wyświetli „za młody" jeżeli wiek był mniejszy niż 18
Silnie typowane widoki wygenerowane z szablonu przez Visual Studio zawierają już odpowiednie wywołania wyświetlające błędy walidacji (zob. np. silnie typowany widok dla Create).

Podobne efekty można uzyskać korzystając z pomocniczych tagów ASP.NET, np.:

```
<div asp-validation-summary="ModelOnly" ...></div>
<div asp-validation-summary="All" ...></div>
<span asp-validation-for="pole" ...></span>
```

Walidacja po stronie klienta

Walidacja po stronie klienta wykorzystuje JavaScript do sprawdzenia poprawności pól. Użytkownik będzie miał możliwość wysłania formularza tylko wtedy, gdy wszystkie pola będą miały poprawną wartość. Walidacją sterujemy dodając atrybuty z przestrzeni nazw `System.ComponentModel.DataAnnotations` do pól modelu:

`Required`	– mówi, że pole jest wymagane
`StringLength(max)`	– ustawia maksymalną długość napisu
`StringLength(max, MinimumLength=min)`	– ustawia minimalną i maksymalną długość
`Range(a, b)`	– ustawia zakres liczbowy
`RegularExpression(regex)`	– wymaga dopasowania do wyrażenia regularnego
`EmailAddress, Phone, Url`	– sprawdzenie zgodności z podanym typem wartości

Do każdego z tych atrybutów można dodać parametr `ErrorMessage`, którego wartością jest komunikat o błędzie, jaki zobaczy użytkownik.

Jeżeli walidacja nie działa poprawnie, należy sprawdzić, czy do widoku dołączone są przygotowane w projekcie skrypty (opcja „Reference script libraries” podczas tworzenia widoku). Są, jeżeli w treści widoku pojawia się fragment, zależny od wersji MVC, np.:

```
@section Scripts {
   @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
   }
```

Przykład walidacji po stronie klienta:

```
using System.ComponentModel.DataAnnotations;
...
public class ModelWalidowany {
   [Required(ErrorMessage=“pole wymagane”)]
   public string imie { get; set; }
   [RegularExpression(@“[A-Z][a-z]+”, ErrorMessage=“zły format”)]
   public string nazwisko { get; set; }
   [StringLength(9, MinimumLength=4, ErrorMessage=“len=4..9”)]
   public string login { get; set; }
   [Required, Range(0, 100)]
   public int wiek { get; set; }
   }
```

Przedstawione tu atrybuty działają zarówno po stronie klienta jak i serwera. Nawet jeżeli błędne dane z formularza dotrą do serwera (np. gdy użytkownik zablokował JavaScript w przeglądarce) to zostaną one zwalidowane po stronie serwera a wynik walidacji zostanie umieszczony w `ModelState`.

Zdalna walidacja (wykorzystująca AJAX)

Kolejnym sposobem weryfikacji danych jest walidacja zdalna. Wykorzystuje ona zapytania asynchroniczne do serwera (AJAX) i pozwala na sprawdzenie zawartości pola po stronie serwera bez konieczności przeładowania całej strony. Dopóki wszystkie sprawdzane pola formularza nie będą poprawne, użytkownik nie będzie mógł wysłać formularza.

Aby można było z niej skorzystać, należy pamiętać by widok odwoływał się go przygotowanych w projekcie skryptów (opcja „Reference script libraries” podczas tworzenia widoku). Następnym krokiem jest nadanie polu modelu atrybutu wskazującego kontroler i akcję, która będzie zajmowała się walidacją:

```
using Microsoft.AspNetCore.Mvc;
...
[Remote(“nazwa akcji”, “nazwa kontrolera”)]
```

Teraz należy zaimplementować wskazaną akcję. Powinna ona przyjmować jeden argument o nazwie takiej samej jak nazwa walidowanego pola. Wartość zwracana z akcji powinna być typu Json i określa ona wynik sprawdzenia. Jeżeli jest to prawda logiczna, walidacja przebiegła poprawnie. Jeżeli zawartość pola jest niepoprawna, należy zwrócić fałsz lub napis z komunikatem o błędzie. W przypadku zwrócenia fałszu zostanie wykorzystany domyślny komunikat o błędzie. Np.:

```
class Model {
   [Remote(“ValidateUzytkownik”, “Kontroler”)]
   string uzytkownik { get; set; }
   }
...
public IActionResult ValidateUzytkownik(string uzytkownik) {
   if(Poprawny(uzytkownik))   // walidacja powiodła się
      return Json(true);
   else                       // błąd walidacji
      return Json(“error message”);
   }
```

Dodatkowy parametr `AdditionalFields` atrybutu `Remote` pozwala przekazać do metody walidującej dodatkowe pola.

Helpery

W widoku dostępnych jest wiele metod ułatwiających tworzenie kodu HTML. Poniżej wymienionych jest kilka z nich. Wszystkich helperów jest o wiele więcej, każdy z wymienionych ma też kilka wariantów, różniących się przyjmowanymi argumentami. Pełną listę można znaleźć pod adresem https://msdn.microsoft.com/en-us/library/system.web.mvc.html(v=vs.118).aspx.

```
Html.CheckBox(string nazwa[, bool wartość[, object atrybutyHtml]])
Html.CheckBoxFor(model => model.pole
                [, bool wartość[, object atrybutyHtml]])
Html.Hidden(string nazwa[, object wartość[, object atrybutyHtml]])
Html.HiddenFor(model => model.pole
                [, object wartość[, object atrybutyHtml]])
Html.Password(string nazwa
                [, object wartość[, object atrybutyHtml]])
Html.PasswordFor(model => model.pole
                [, object wartość[, object atrybutyHtml]])
Html.RadioButton(string nazwa[, object wartość
                [, bool checked, [object atrybutyHtml]]])
Html.RadioButtonFor(model => model.pole[, object wartość
                [, bool checked, [object atrybutyHtml]]])
Html.TextBox(string nazwa
                [, object wartość[, object atrybutyHtml]])
Html.TextBoxFor(model => model.pole
                [, object wartość[, object atrybutyHtml]])
```

wstawia odpowiedni element HTML o nazwie `nazwa` lub nazwie określonej przez pole `pole` modelu; `wartość` określa wartość elementu; `atrybutyHtml` pozwalają ustawić dodatkowe parametry tagu; np.

```
@Html.CheckBox(“zgoda”, true,
                new { @class=“klasacss”, id=“identyfikator” })
```

```
Html.EditorFor(model => model.pole)
```

wstawia odpowiedni element pozwalający na edycję pola `pole` modelu; element zależy od typu pola (`intput:text` dla `string`, `intput:checkbox` dla `bool` itp.),

```
Html.LabelFor(model => model.pole)
```

wstawia opis pola `pole` modelu; opis jest brany z parametru `Name` atrybutu `Display`, bądź jest to nazwa pola,

```
Html.ActionLink(string opis, string akcja[, string kontroler]
                [object parametryTrasy, object atrybutyHtml])
```
wstawia link (tag A) prowadzący do podanej akcji w bieżącym lub podanym kontrolerze; `parametryTrasy` pozwalają na ustawienie parametrów dla akcji; `atrybutyHtml` pozwalają ustawić dodatkowe parametry tagu,

```
Html.BeginForm(string akcja, string kontroler,
               FormMethod metoda, object atrybutyHtml)
```
tworzy formularz (element `FORM`) prowadzący do podanej akcji we wskazanym kontrolerze; `metoda` opisuje, czy formularz wysyłany jest zapytaniem POST czy GET; `atrybutyHtml` pozwalają ustawić dodatkowe parametry tagu; przykład użycia:

```
@using(Html.BeginForm("Login", "User", FormMethod.Post,
                      new { id="loginform"})) {
     Html.TextBox("login");
     Html.TextBox("passwd");
     }
```

```
Html.AntiForgeryToken()
```
wraz z atrybutem `ValidateAntiForgeryToken` akcji odbierającej dane z formularza służy jako zabezpieczenie przed atakami typu CSRF; omówienie można znaleźć pod adresem http://www.devcurry.com/2013/01/what-is-antiforgerytoken-and-why-do-i.html

Asynchroniczne akcje

Do tworzenia akcji możemy skorzystać z metod asynchronicznych języka C#. Wszystko co musimy zrobić, to dodać słowo kluczowe `async` do sygnatury akcji i zwrócić z niej `Task<IActionResult>`.

```
using System.Threading.Tasks;
...
public async Task<IActionResult> Akcja() {
     ... await ...
     return View();
     }
```

Lokalizacja

Aby utworzyć różne wersje językowe naszej aplikacji, możemy skorzystać z zasobów (add -> new item -> Visual C# / General -> resources file). Dodajemy do projektu pliki zasobów o nazwach w formacie:

`Views.`*`kontroler.akcja.kultura`*`.resx` oraz

1. `Models.`*`nazwa_modelu.kultura`*`.resx`

kontroler i *akcja* określają widok, którego dotyczy zasób, *nazwa_modelu* określa model, zaś *kultura* to kod opisujący język (np. pl, en, en-US itp.). Każdy zasób jest zbiorem par typu klucz (nazwa), wartość (napis).

Dodatkowo musimy włączyć usługi lokalizacji. W pliku `Startup.cs` w metodzie `ConfigureServices` dodajemy

```
using Microsoft.AspNetCore.Localization;
...
services.AddLocalization();
services.AddMvc()
   .AddDataAnnotationsLocalization()
   .AddViewLocalization(options
                          => options.ResourcesPath = “Resources”);
```

zaś w metodzie `Configure` podajemy obsługiwane kultury oraz ustawiamy domyślną:

```
using System.Globalization;
...
var supportedCultures = new[] {
   new CultureInfo(“pl-PL”), new CultureInfo(“pl”),
   new CultureInfo(“en-US”), new CultureInfo(“en”) };
app.UseRequestLocalization(new RequestLocalizationOptions {
   DefaultRequestCulture = new RequestCulture(“pl”),
   SupportedCultures = supportedCultures,
   SupportedUICultures = supportedCultures
   });
```

W widoku który ma mieć różne wersje językowe umieszczamy dyrektywy

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
```

a odpowiednie napisy wstawiamy poprzez

```
@Localizer[“napis”]
```

Zamiast wstawiać wyżej wymienione dyrektywy w każdym widoku, możemy je dołączyć do każdego z nich umieszczając je w pliku `Views \ Shared \ _ViewImports.cshtml`.

Komunikaty o błędach modelu lokalizujemy podając w `ErrorMessage` klucz, np.

```
[StringLength(9, MinimumLength=2, ErrorMessage=“klucz”)]
```

Więcej o lokalizacji:
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization

Błędy i wyjątki

MVC domyślnie zwraca pustą stronę w przypadku błędu HTTP (400, 404, 500 itp.). Jeżeli chcemy zmodyfikować to zachowanie, musimy dodać dodatkową usługę przed innymi usługami wpływającymi na zawartość odpowiedzi (jak `UseStaticFiles`, `UseMvc`). Mamy tu do wyboru kilka opcji:

```
app.UseStatusCodePages();
```
zwraca przedefiniowaną treść, np. `Status Code: 404; Not Found`;

```
app.UseStatusCodePages(content_type, body);
```
zwraca treść typu mime *`content_type`* o zawartości *`body`*; w body ciąg znaków `{0}` zostanie zastąpiony przez kod błędu (404 itp.);

`app.UseStatusCodePages(async context => { ... });`
zwraca treść wygenerowaną przez podaną funkcję; korzystając z `context.HttpContext.Response` możemy ustawić typ zwracanej zawartości (`.ContentType`) czy też ustawić zawartość wołając `.WriteAsync`.

`app.UseStatusCodePagesWithRedirects(`*`adres`*`);`
zwraca przekierowanie pod wskazany *`adres`*; ponieważ jest to przekierowanie (kod HTTP 302), oryginalny kod odpowiedzi jest tracony; ciąg znaków `{0}` w adresie zostanie zastąpiony przez oryginalny kod HTTP.

`app.UseStatusCodePagesWithReExecute(`*`adres, parametry_zapytania`*`);`
przekazanie do akcji pod adresem *`adres`*; akcja jest wyznaczona przez routing ale nie jest to przekierowanie HTTP więc oryginalny kod zostaje zachowany; ciąg znaków `{0}` w adresie lub parametrach zapytania zostanie zastąpiony przez kod HTTP; *`adres`* powinien zaczynać się od znaku / zaś parametry zapytania od znaku ?, jak w poniższym przykładzie

```
app.UseStatusCodePagesWithReExecute("/Home/Status","?code={0}");
...
public IActionResult Status(int code) {
   ViewBag.code = code;
   return View();
   }
```

Wyjątki nie złapane przez logikę naszej aplikacji są obsługiwane przez kod umieszczony w wygenerowanej z szablonu aplikacji. Robi to usługa, w zależności od środowiska:

```
app.UseDeveloperExceptionPage();
```

w przypadku środowiska developerskiego lub

```
app.UseExceptionHandler(...);
```

w przypadku środowiska produkcyjnego. W drugim przypadku podajemy adres akcji obsługującej wyjątek. Implementacja z szablonu prowadzi do akcji `Error` kontrolera `Home` i prezentuje widok `Shared \ Error.cshtml`. W treści akcji mamy dostęp do rzuconego wyjątku i adresu akcji która rzuciła wyjątkek poprzez następujący fragment:

```
var exceptionHandlerPathFeature =
    HttpContext.Features.Get<IExceptionHandlerPathFeature>();
exceptionHandlerPathFeature?.Error;     // wyjątek
exceptionHandlerPathFeature?.Path;      // adres akcji
```

Przegląd innych metod obsługi wyjątków i błędów można znaleźć pod adresem:
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-2.1

Filtry akcji

Jeżeli musimy wykonać jakiś wspólny kod przed wykonaniem każdej z kilku akcji, możemy skorzystać z filtrów akcji. Są to klasy dziedziczące po klasie
`Microsoft.AspNetCore.Mvc.Filters.ActionFilterAttribute`
i implementujące metodę `OnActionExecuting`. Tak zaimplementowaną klasę wykorzystujemy

jako atrybut akcji, do której chcemy dołączyć nasz filtr. Metoda ta przyjmuje jeden parametr: `context`. Zawiera on wszystkie potrzebne nam dane. Dane trasy otrzymujemy w `context.RouteData`, opis zapytania wraz z sesją w `context.HttpContext`, opis akcji w `context.ActionDescriptor` a jej parametry w `context.ActionParameters`. Jeżeli chcemy zmienić wynik akcji, możemy to zrobić ustawiając `context.Result` na odpowiedni wynik. Kod odpowiedzi HTTP możemy ustawić w `context.HttpContext.Response.StatusCode`. Mamy także dostęp do kontrolera w polu `context.Controller`.

W poniższym przykładzie do akcji `Contact` dodajemy filtr `Test`, który pozwala wykonać tę akcję jedynie, gdy ustawiony jest parametr `p` o wartości `pass` (np. w zapytaniu). Jeżeli parametru nie ma, lub ma niepoprawną wartość, użytkownik przekierowywany jest do widoku `Index`.

```
public class TestAttribute: ActionFilterAttribute {
   public override void OnActionExecuting(ActionExecutingContext
                                          context) {
      if(context.HttpContext.Request.Params["p"] != "pass")
         context.Result = new ViewResult() { ViewName = "Index" };
      }
   }
...
public class HomeController : Controller {
   public IActionResult Index() { return View(); }
   [Test]
   public IActionResult Contact() { return View(); }
   }
```

Proszę zwrócić uwagę, że jest to przekierowanie od razu do widoku, z pominięciem wykonania akcji. Jeżeli chcielibyśmy ustawić jakieś parametry w `ViewBag`, musimy to zrobić w filtrze, np. tworząc obiekt typu `ViewDataDictionary` wypełniony potrzebnymi danymi, który następnie przekażemy do `ViewResult` w polu `ViewData`.

Jeżeli chcemy wykonać dodatkowy kod po zakończeniu wykonywania akcji, powinniśmy nadpisać metodę `OnActionExecuted(ActionExecutedContext context)`. W tym przypadku `context` zawiera dodatkowe pola: `Canceled` informuje o tym, czy akcja była pominięta (gdy jakiś filtr nadpisał wynik akcji), `Exception` zawiera wyjątek, który został rzucony przez akcję (jeżeli nie było wyjątku pole będzie miało wartość null), zaś w `ExceptionHandled` możemy poinformować MVC, że filtr obsłużył ten wyjątek i nie ma potrzeby dalszej jego obsługi (na przykład przez wyświetlenie strony z błędem). W takiej sytuacji powinniśmy też ustawić odpowiednio pole `Result`.

Poza filtrami akcji mamy również do dyspozycji filtry autoryzacyjne, filtry wyników i filtry wyjątków. Więcej pod adresami:
https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1