

Tworzenie i testowanie aplikacji Internetowych przy użyciu ASP.NET Core MVC

Autor : Paweł Bubień

pbubien@pl.sii.eu

Wrocław, 20.01.2018

ASP.NET Core MVC

<https://github.com/pawel87/SiiTrainingCodeSample>

Program szkolenia

CZĘŚĆ 1

**Zaawansowane zagadnienia
(+ testowanie)**

- 1) ASP.NET Core MVC Pipeline,
- 2) Filtry,
- 3) Model Binders,
- 4) Routes,
- 5) Tag Helpers,
- 6) Cache,
- 7) Walidatory
- 8) View Components
- 9) Globalizacja

CZĘŚĆ 2

**Programowanie asynchroniczne &
Web API**

- 1) tworzenie asynchronicznych akcji, cancellation token w kontrolerach,
- 2) tworzenie Web API w ASP.NET CORE (w tym: formatowanie danych, ApiExplorer, EF Core In-Memory Data Provider)

CZĘŚĆ 3

**Testy integracyjne w
ASP.NET Core MVC**

Część 1

ZAAWANSOWANE ZAGADNIENIA

ASP.NET CORE MVC

0. Początek

Zadanie

Utwórz solucję w .NET Core zawierającą:

A) Projekt z aplikacją webową
ASP.NET Core MVC

B) Projekt z testami jednostkowymi w
.NET Core *

C) Projekt z testami integracyjnymi w
.NET Core **

D) Stwórz obiekty domenowe dla bloga,
minimum to:

- Post,
- PostCategory (np. Enum),
- PostComment

E) Stwórz repozytorium (interfejs + implementacja)
do następujących operacji w kontekście bloga
,czyli np:

- GetPost(int postId),
- GetAll(),
- GetByCategory(eCategory category),
- AddPost(Post post),
- GetLatestPosts()

F) Wstrzyknij implementację swojego repozytorium
w konfiguracji aplikacji, aby móc korzystać z jej
implementacji

* - zainstaluj pakiety z NuGet:

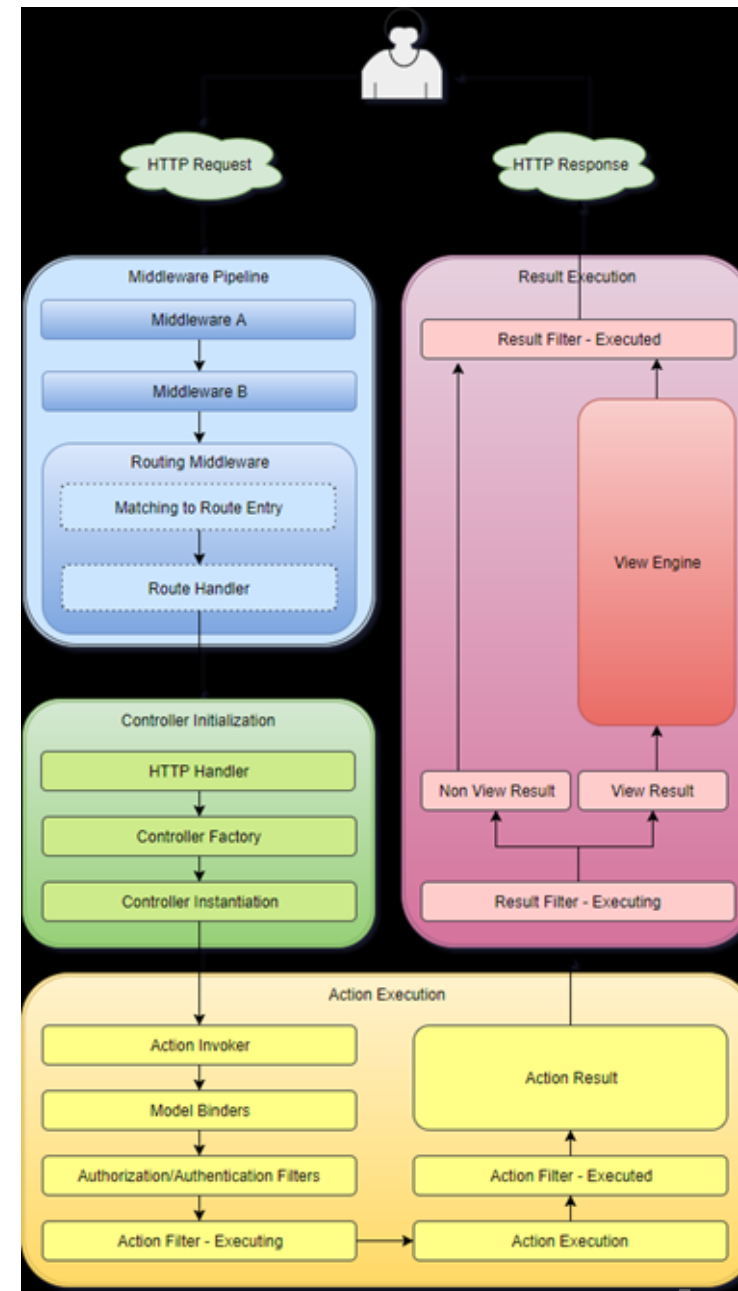
- Moq (testy jednostkowe)

** - zainstaluj pakiety z NuGet:

- Microsoft.AspNetCore.TestHost

1. ASP.NET Core MVC Pipeline

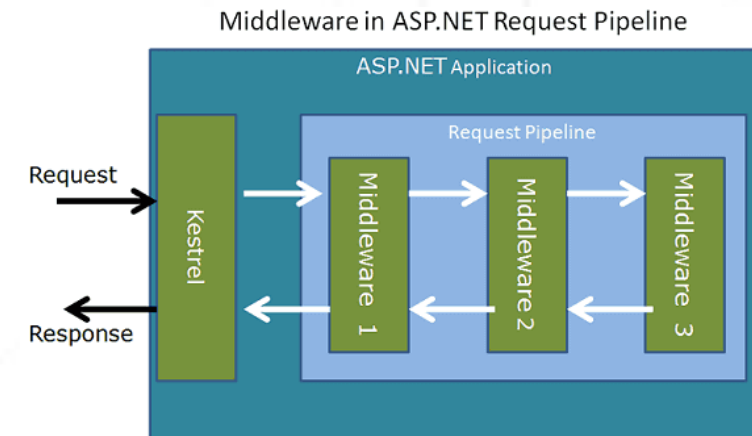
- Middleware Pipeline
- Routing Middleware
- Controller Initialization
- Action Execution
- Model Binding
- Filters
- Action Result
- Result Exception



CZYM JEST MIDDLEWARE?

W ASP.NET Core middleware kontroluje jak nasza aplikacja odpowiada na żądania HTTP. Może także kontrolować jak aplikacja wygląda kiedy wystąpi wyjątek, obsługuje autentykację i autoryzację użytkowników etc.

- middleware są komponentami oprogramowania, które są osadzone w pipeline aplikacji aby obsługiwać requesty i response,
- każdy komponent może określić czy przekazać request do następnego komponentu w pipeline i może wykonać różne akcje przed i po wywołaniu następnego komponentu,
- każdy element middleware w ASP.NET Core jest obiektem i ma bardzo specyficzną i ograniczoną rolę.



KONFIGURACJA MIDDLEWARE

Aby zacząć używać jakiegokolwiek Middleware, musimy dodać go do pipeline'a.

W tym celu wykorzystujemy metodę "Configure" z klasy "Startup.cs"

```
app.Run(async (context) =>
{
    //middleware logic here
});
```

Metoda "Run" ma dostęp do instancji HttpContext

Extension methods "Use" oraz "Run" umożliwiają nam zarejestrowanie "Inline Middleware" w pipeline'ie.

Metoda Run dodaje 'kończący' middleware, metoda Use dodaje middleware, który może wywołać kolejny

```
app.Use(async (context, next) =>
{
    var text = "Hello World from inline middleware 1!";
    var bytes = Encoding.ASCII.GetBytes(text);
    await context.Response.Body.WriteAsync(bytes, 0, bytes.Length);
    await next.Invoke();
});
```


WŁASNY MIDDLEWARE

Innym sposobem na stworzenie middleware jest użycie klasy.

Klasa zawierająca middleware nie musi implementować żadnego interfejsu, ani po niczym dziedziczyć.

1. Klasa musi deklarować niestatyczny publiczny konstruktor z co najmniej 1 parametrem typu `RequestDelegate`
2. Klasa musi definiować publiczną metodę "Invoke" która przyjmuje `HttpContext` i zwraca `Task`

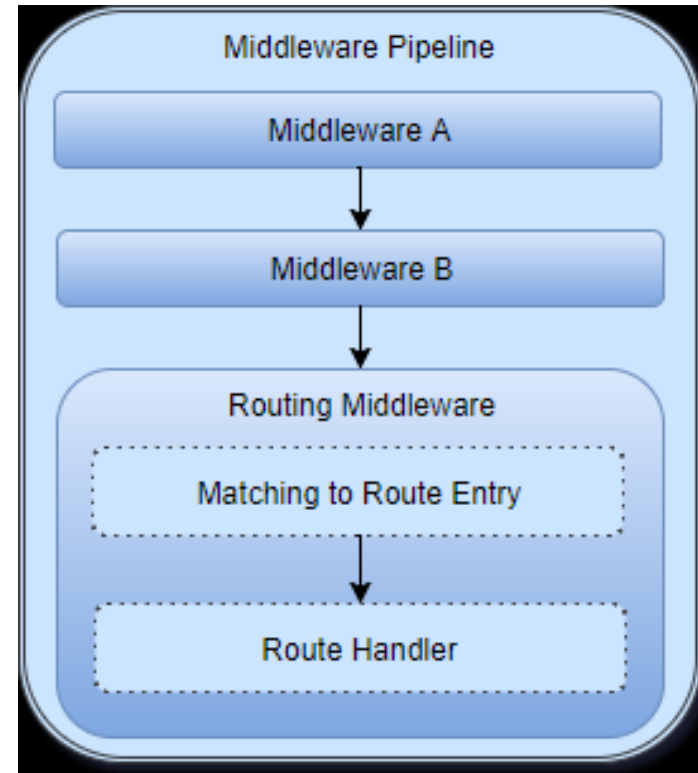
```
public class SampleMiddleware
{
    private readonly RequestDelegate _next;

    public SampleMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        await context.Response.WriteAsync("<div> Hello from");
        await _next(context);
        await context.Response.WriteAsync("<div> Bye from Mi");
    }
}
```

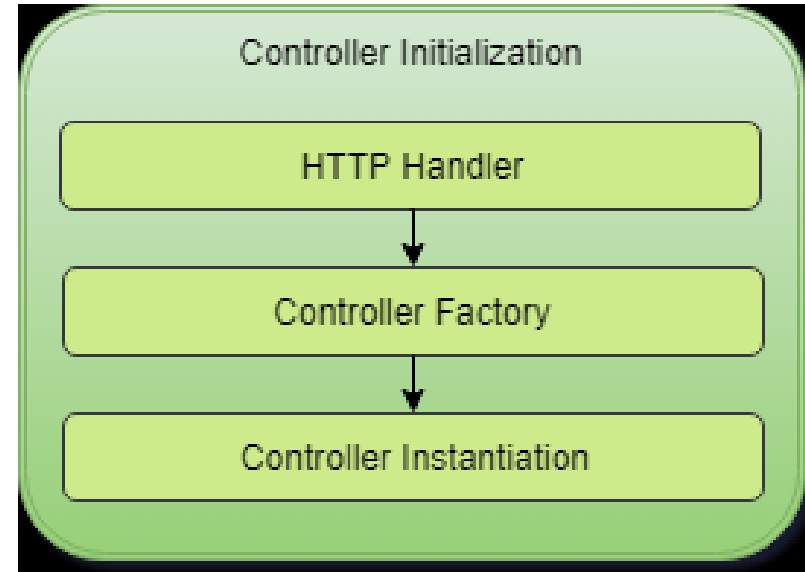

ROUTING MIDDLEWARE

- Jeden z wielu dostępnych middleware w ASP.NET Core, ale jeden z najważniejszych w pipeline'ie Core MVC.
- Odpowiada za uruchomienie pipeline'a dla MVC. Jeśli adres URL requesta nie zostanie dopasowany do żadnego zdefiniowanego route'a, wówczas request jest przekazywany dalej do kolejnego middleware pomijając w tym przypadku MVC pipeline.



CONTROLLER INITIALIZATION

- W momencie gdy znaleziony zostaje route przez middleware Routing, kolejnym etapem jest znalezienie kontrolera który odpowiada routingowi i jego inicjalizacja,
- Proces szukania najbardziej pasującego kontrolera i akcji dla danego Route również może być customizowany.

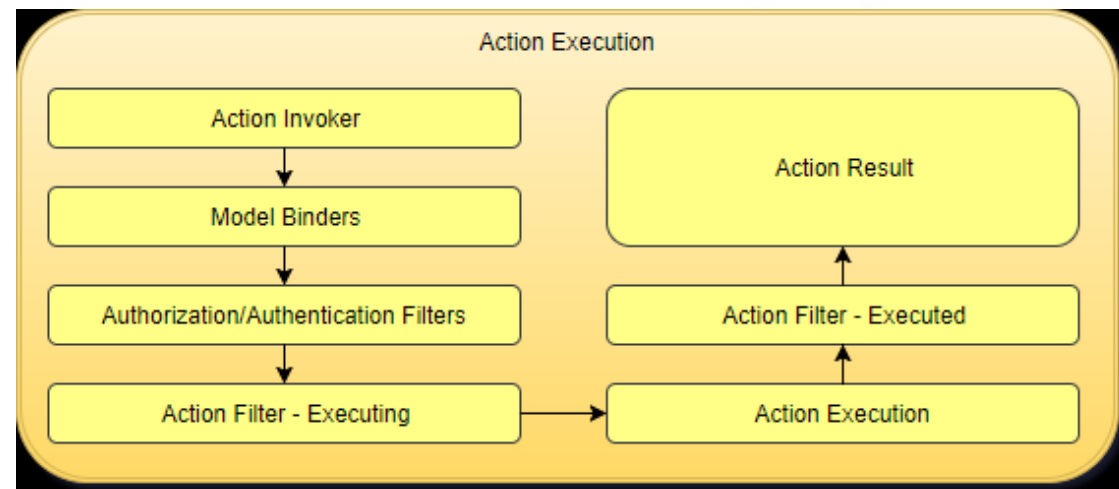


1. ASP.NET Core MVC Pipeline

Action Execution

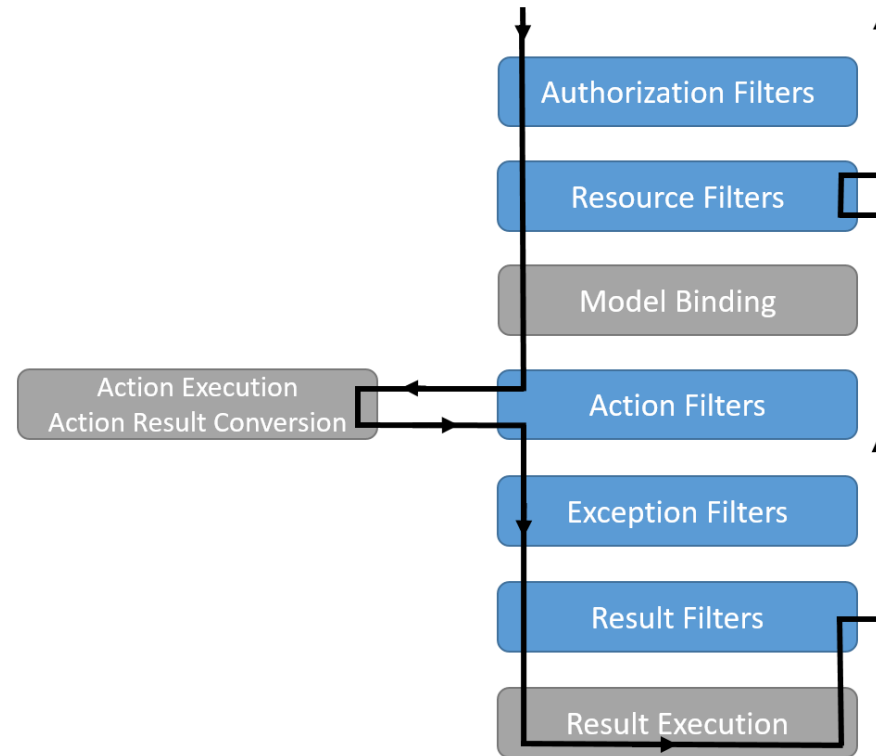
Serce wzorca MVC. Proces Action Execution przetwarza nadchodzące requesty, a na końcu generuje wynik (Result).

W tym procesie możemy dodać zdecydowaną większość naszych komponentów.



FILTER PIPELINE

- Authorization filter - określa czy użytkownik może wykonać aktualny request,
- Resource Filter - można ich używać do implementacji cache'a lub innych powodów wydajnościowych. Jako, że odpalany jest przed bindingiem modelu, może także wpływać na ten proces,
- Action Filters – odpalany przed i po wykonaniu akcji,
- Exception Filters - obsługa wyjątków zanim nastąpi wyświetlenie response,
- Result filters – wykonywane przed i po wyniku akcji. Odpalane są wyłącznie jeśli akcja się powiedzie.



1. ASP.NET Core MVC Pipeline

Model Binding

- Funkcjonalność pozwalająca frameworkowi MVC na bindowanie danych z requestu do obiektów POCO.

Filters

- Filtry są to komponenty, które mogą być wstrzyknięte na wielu etapach wykonywania akcji i pozwalają nam określać zachowanie, którego oczekujemy w naszych akcjach.

Action Result

- Ostatnim etapem wykonywania akcji jest Action Result, tworzący typ wyniku, który będzie następnie przetworzony przez Result Execution.

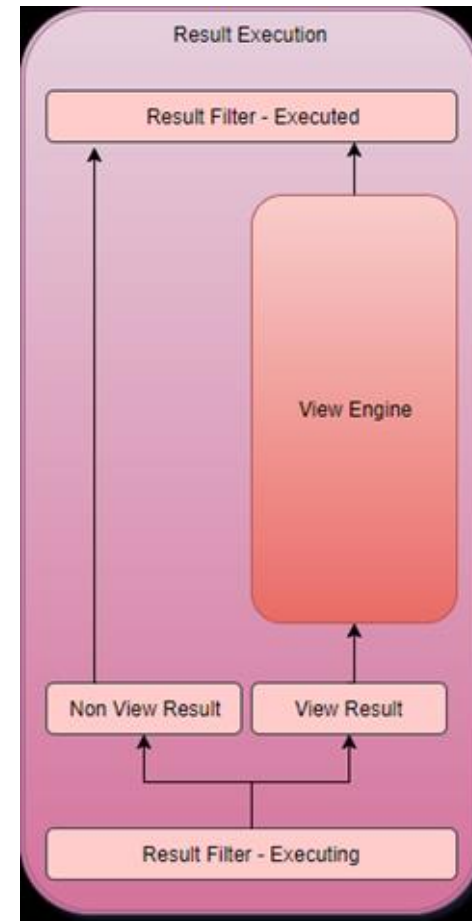
1. ASP.NET Core MVC Pipeline

Result Execution

Ostatni etap w ASP.NET Core Pipeline.

Tutaj nasz 'wynik' będzie przeanalizowany i odpowiednio przeprocesowany.

Również w tym miejscu można wstrzyknąć różne filtry, które będą wykonane przed i po procesowaniu, umożliwiając nam zmianę sposobu w jaki wynik będzie przetworzony.



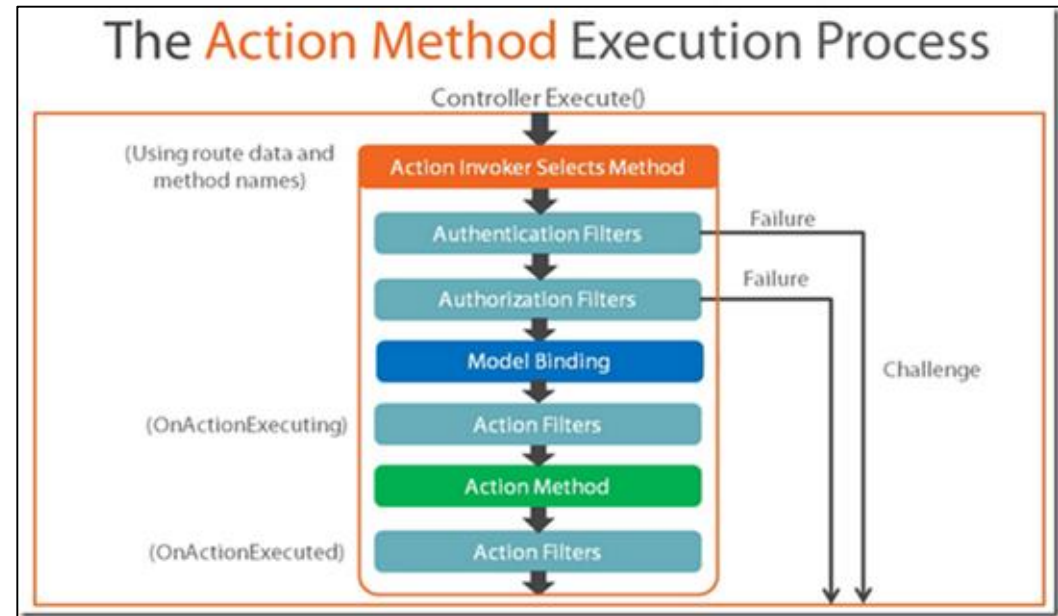
1. ASP.NET Core MVC Pipeline

Zadanie

1. **Utwórz własny middleware, który:**
 - będzie logował bieżący URL requesta i godzinę,
 - w przypadku gdy jeden z przesłanych nagłówków będzie zawierał klucz „sii-training” zwróć status 401 (unauthorized) i przerwij dalsze przetwarzanie requesta. (sprawdź za pomocą narzędzia np. Postman)
2. **Napisz extension method umożliwiającą dodanie Twojego middleware do application buildera.**
3. **Utwórz dwie akcje o takiej samej nazwie, jedna z nich powinna obsługiwać wyłącznie połączenia HTTPS, natomiast druga HTTP. Wykorzystaj do tego własny atrybut.**

2. FILTERS

- Filtry umożliwiają nam wykonanie fragmentu kodu przed lub po wykonaniu metody (Action Method) kontrolera,
- Służą do wykonywania często powtarzalnych zadań na naszych metodach,
- Filtry są wywoływane na konkretnie określonych etapach wykonywania całego requestu,
- ASP.NET Core MVC udostępnia wiele wbudowanych filtrów, jak również umożliwia nam tworzenie własnych.



2. FILTERS

Problem	Rozwiązanie
„Wstrzyknięcie” dodatkowej logiki w trakcie przetwarzania requestu	Użycie filtra na poziomie kontrolera lub akcji.
Ograniczenie dostępu do akcji kontrolera	Użycie Authorization Filter
Zbadanie lub modyfikacja rezultatu zwróconego przez akcję	Użycie Result Filter
Użycie własnych/customowych serwisów w filtrze	Deklaracja zależności w konstruktorze filtra, rejestracja serwisu w Startup.cs i użycie filtra za pomocą atrybutu TypeFilter
Użycie filtra przy każdej dostępnej akcji w ramach aplikacji	Użycie Global Filter
Zmiana kolejności w jakiej filtr jest wykonywany	Użycie Order Parameter

2. FILTERS

W jaki sposób tworzyć filtry?

- a) Prosty Action Filter z użyciem atrybutu [**ActionFilterAttribute**],
- b) Implementacja interfejsu **IActionFilter**,
- c) Użycie **ServiceFilterAttribute**,
- d) Użycie **TypeFilterAttribute**,
- e) Implementacja interfejsu **IFilterFactor**.

2. FILTERS

ActionFilterAttribute

ASP.NET Core dostarcza puste implementacje interfejsów filtrów, gdzie jedną z nich jest **ActionFilterAttribute** implementujący **IActionFilter** i dziedziczący po **Attribute**.

* Są tu również dostępne metody asynchroniczne

```
public class SampleActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Hello");
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        base.OnActionExecuted(context);
        Console.WriteLine("Bye bye");
    }
}
```

```
public class SampleController
{
    [SampleActionFilter]
    public string Method()
    {
        return "Welcome";
    }
}
```

2. FILTERS

IActionFilter

Implementacja interfejsu **IActionFilter**. Jedyna różnica polega na tym, że chcąc zaimplementować wyłącznie 1 operację (np. **OnActionExecuting**), kolejna musi być zaimplementowana jako pusta.

```
public class SampleActionFilter : Attribute, IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Hello");
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
    }
}
```

2. FILTERS

ServiceFilterAttribute

Przydatny, kiedy chcemy wstrzyknąć zależność do naszego filtra poprzez konstruktor.

- 1) stworzyć filtr
- 2) zarejestrować filtr :
`services.AddScoped<SampleActionFilter>();`

```
public class SampleController
{
    [ServiceFilter(typeof(SampleActionFilter))]
    public string Method()
    {
        return "Welcome";
    }
}
```

```
public class SampleActionFilter : IActionFilter
{
    private readonly IRepository repository;

    public SampleActionFilter(IRepository repository)
    {
        this.repository = repository;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        var data = repository.GetAllProducts();
        Console.WriteLine("Hello");
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
    }
}
```

2. FILTERS

TypeFilterAttribute

Przydatny, jeżeli chcemy przekazać do filtru argumenty, które nie mogą zostać rozwiązane za pomocą Dependency Injection.

```
public class SampleController
{
    [TypeFilter(typeof(SampleActionFilter), Arguments = new[] { "abc" })]
    public string Method()
    {
        return "Welcome";
    }
}
```

```
public class SampleActionFilter : IActionFilter
{
    private readonly IRepository repository;
    private string text;
    private int value;

    public SampleActionFilter(IRepository repository, string text, int value)
    {
        this.repository = repository;
        this.text = text;
        this.value = value;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        var data = repository.GetAllProducts();
        Console.WriteLine($"Hello {text}");
    }
}
```


2. FILTERS

IFilterFactory

W przypadku `TypeFilterAttribute` nie możemy przekazać jako argumentów zwykłych klas. Jeśli chcemy mieć pełną kontrolę nad inicjalizacją filtra, możemy stworzyć własną fabrykę za pomocą `IFilterFactory`.

```
public class SampleActionFilterImpl : ActionFilterAttribute
{
    private readonly IRepository repository;
    private string text;
    private readonly SomeOptions options;

    public SampleActionFilterImpl(IRepository repository, string text, SomeOptions options)
    {
        this.repository = repository;
        this.text = text;
        this.options = options;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var data = repository.GetAllProducts();
        Console.WriteLine($"Hello {text}");
    }
}
```

```
public class SampleActionFilterAttribute : Attribute, IFilterFactory
{
    private string text;

    // Indicates if the filter created can be reused accross requests.
    public bool IsReusable => false;

    public SampleActionFilterAttribute(string text)
    {
        this.text = text;
    }

    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new SampleActionFilterImpl(
            serviceProvider.GetRequiredService<IRepository>(),
            text,
            new SomeOptions { Value = 12345 });
    }
}
```

2. FILTERS

Result Filters

- Filtry implementujące interfejs `IResultFilter` lub `IAsyncResultFilter`
- Są wykonywane wyłącznie dla poprawnych wyników - wtedy, kiedy akcja generuje wynik.
- W przypadku złapania wyjątku przez `ExceptionHandler` filtr nie jest uruchamiany.
- Jeśli przerywamy proces wykonywania akcji, wówczas powinniśmy dodać do wyniku response'a treść, aby uniknąć zwracania pustych response'ów.

```
public class AddHeaderFilterWithDi : IResultFilter
{
    private ILogger _logger;
    public AddHeaderFilterWithDi(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation($"Header added: {headerName}");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has already begun.
    }
}
```

2. FILTERS

Kolejność wykonywania filtrów
(patrz: tabelka obok)

Jak nadpisać kolejność wykonania
filtra?

Implementując interfejs
`IOrderedFilter`. Interfejs udostępnia
mam właściwość `"Order"` który
zostanie użyty do określenia
kolejności wykonania.

Filtr z najniższą wartością zostanie
wykonany przed uruchomieniem
filtra o wyższej wartości

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

Sequence	Filter scope	Filter method
1	Global	OnActionExecuting
2	Controller	OnActionExecuting
3	Method	OnActionExecuting
4	Method	OnActionExecuted
5	Controller	OnActionExecuted
6	Global	OnActionExecuted

2. FILTERS

Zadanie

Zaimplementuj filtry za pomocą wszystkich przedstawionych metod.

A) Implementacja [ActionFilterAttribute]:

Napisz filtr (wersja synchroniczna i asynchroniczna), który:
- wyliczy czas wywołania akcji i wyświetli te dane na stronie

B) Implementacja IActionFilter przy użyciu [ServiceFilterAttribute]

Napisz filtr, który będzie korzystał z interfejsu `IRepository` i sprawdzi, czy dany post nie został usunięty (np. flaga `IsDeleted` w obiekcie `Post`). Jeśli dany post jest usunięty, zwróć rezultat 404 (`NotFound`).

C) Implementacja IActionFilter przy użyciu [TypeFilterAttribute]

Napisz filtr taki jak w pkt B z tą różnicą, że przyjmuje on dodatkowy parametr `showDeleted`, który w przypadku wartości `true` wyświetli także usunięte posty.

D) IFilterFactory – filtr jako parametry używa interfejsu `ILogger`, flagi (jak w pkt 3) oraz klasy, która będzie symulować instancję bloga (np. `id`, `nazwa` etc). Filtr powinien zapisywać do logów czas wykonania akcji, aktualny czas oraz informacje o aktualnym blogu.

Napisz testy jednostkowe, które sprawdzą poprawność logiki filtrów z pkt B i C w kontekście wywołania action filter.

3. MODEL BINDING

- **Czym jest model binding?**

Model binding mapuje dane z requestu HTTP do parametrów Action Method. Parametry mogą być typami prostymi (string, int, float), jak również złożonymi.

- **Jak działa model binding?**

Routing middleware „rozkłada” żądanie HTTP do klasy RouteData, a następnie przypisuje jego wartości do parametrów akcji wg nazwy. Parametry akcji mogą być typami prostymi lub złożonymi. Dla typów złożonych (klasy) model binding używa refleksji. Typ musi posiadać domyślny konstruktor i publiczne właściwości.

3. MODEL BINDING

Jak działa model binding?

Przykładowy URL:

<http://localhost/blog/post/1>

Założenia:

Route template:

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

1. Routing kieruje nas do kontrolera "Blog" i akcji "Post"

2. Przekazujemy parametr 1 jako "id"

3. Sygnatura metody:

```
public IActionResult Post(int id) { }
```

3. MODEL BINDING

Atrybuty, które nadpisują domyślne zachowanie bindingu :

Atrybut	Opis
[BindRequired]	Dodaje błąd do ModelState jeśli binding się nie powiedzie
[BindNever]	Ignoruje bindowanie parametru
[FromHeader]	Używa nagłówka HTTP
[FromQuery]	Używa wartości z Query String
[FromRoute]	Używa wartości z routingu
[FromForm]	Używa wartości z formularza (HTTP POST)
[FromServices]	Wstrzykuje wartość za pomocą dependency injection
[FromBody]	Używa HTTP request body w oparciu o skonfigurowany formatter (np. JSON, XML). Tylko 1 parametr akcji może używać tego atrybutu
[ModelBinder]	Dostarcza użycie własnego model bindera

3. MODEL BINDING

Na co zwrócić
uwagę...

- nie każde źródło bindingu jest domyślnie sprawdzane. Przykładowo, jeśli chcemy wykorzystać binding z Body musimy skorzystać z [FromBody], w przeciwnym wypadku BodyModelBinder nie zostanie użyty (tak samo dla Header, File),
- JSON formatted jest dodany domyślnie, jeśli chcemy skorzystać z XML formatter'a musimy go dodać sami,

3. MODEL BINDING

- **FromForm Attribute**
`public IActionResult Detail([FromForm] ProductViewModel productViewModel) => View(productViewModel);`
- **FromRoute Attribute**
`public IActionResult Detail([FromRoute] int id) => View();`
Atrybut sprawia, że bindujemy wyłącznie dane, które zawierają się w naszym route data
- **FromQuery Attribute**
`public IActionResult Detail([FromQuery] int id) => View();`
Pobieranie danych wyłącznie z query stringa. Przykładowo dla requesta <http://Product/Detail/2?id=5> wartość id wynosi 5.
- **FromHeader Attribute**
Używamy wtedy, kiedy chcemy zbindować nagłówki requesta. Dla pobrania konkretnej wartości nagłówka używamy parametru Name, np. `:[FromHeader(Name = "Accept-Language")]`
- **From Body Attribute**
Używamy, aby zbindować zawartość body z requesta, np. wykonując AJAX'owe requesty i wysyłając obiekty JSON jako dane `$.get("/Product/Detail", { productViewModel: productViewModel }, function (data, textStatus, jqXHR) { }, "json");`
`public IActionResult Detail([FromBody] ProductViewModel productViewModel) => View(productViewModel);`

3. MODEL BINDING

Własny model binder – kiedy się przydaje?

- gdy pobieramy kilka wartości osobno, ale chcemy je połączyć w „1 całość” (np. data + godzina w DateTime);
- gdy model przesyłany z frontendu jest inny niż model backendowy,
- gdy musimy do naszego modelu 'dorzucić' pewne dane, które nie znajdują się w payload requesta (np. wartości z bazy danych, konfiguracji, ciasteczka etc)

3. MODEL BINDING

Własny model binder – jak zacząć?

Przykład:

<http://www.domena.pl/getposts?year=2018&month=2>

1. Tworzymy ViewModel, który będzie zawierał pole typu DateTime, do którego będziemy chcieli zbindować wartości z requesta (osobno rok i miesiąc)
2. Tworzymy ModelBinder implementujący **IModelBinder**
3. Tworzymy swojego providera, który dla naszego ViewModel'u użyje stworzonego ModelBindera
4. Dołączamy naszego providera do listy dostępnych w pliku Startup.cs (ConfigureServices)

3. MODEL BINDING

Model Binder z FallbackBinder'em

Kiedy warto stosować?

Jeśli np. pewne wartości nie zostały przekazane, a chcemy w dalszym ciągu spróbować obsłużyć wartości które przysły z requesta.

```
public class DateAndTimeModelBinderWithFallback : IModelBinder
{
    private readonly IModelBinder fallbackBinder;
    public DateAndTimeModelBinderWithFallback(IModelBinder fallbackBinder)
    {
        this.fallbackBinder = fallbackBinder;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var datePartName = $"{bindingContext.ModelName}.Date";
        var timePartName = $"{bindingContext.ModelName}.Time";
        var datePartValues = bindingContext.ValueProvider.GetValue(datePartName);
        var timePartValues = bindingContext.ValueProvider.GetValue(timePartName);

        // Fallback to the default binder when a part is missing
        if (datePartValues.Length == 0 || timePartValues.Length == 0) return fallbackBinder.BindModelAsync(bindingContext);
    }
}
```

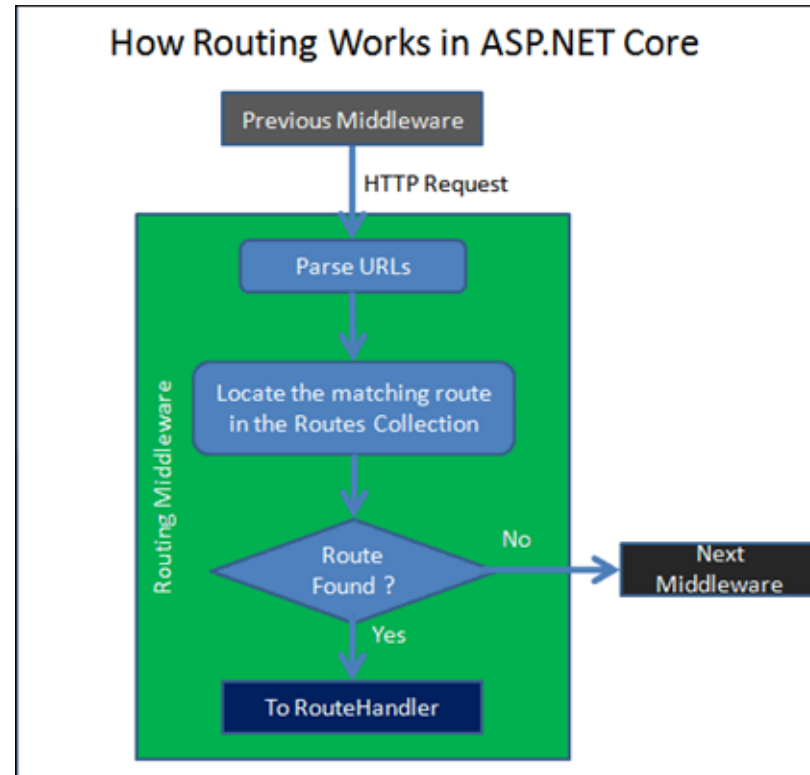
3. MODEL BINDING

ĆWICZENIE

- A) Stwórz model i akcje kontrolerów które wykorzystują wszystkie przedstawione atrybuty do bindingu. Sprawdź zachowanie wykorzystując do tego Postman'a.
- B) Napisz własny model binder, którego zadaniem będzie zmapowanie danych z formularza, który zawiera kilka elementów, w tym między innymi miesiąc, dzień i rok. Na podstawie tych 3 wartości model binder powinien ustawić wartość pola typu DateTime w ViewModelu, aby była ona zgodna z tym co przekazał użytkownik.
- C) Napisz wersję z fallback'iem do domyślnego model bindera dla typu DateTime (new SimpleTypeModelBinder(typeof(DateTime)))

4. ROUTING

1. Czym jest Routing?
2. Routing – kilka definicji
3. Definiowanie ograniczeń



<https://www.tektutorialshub.com/routing-in-asp-net-core/>

4. ROUTING

Czym jest Routing?

Routing w ASP.NET Core MVC jest mechanizmem, dzięki któremu request jest mapowany do odpowiedniego kontrolera i jego akcji.

Dzieje się to dzięki dodaniu Routing Middleware do naszego pipeline'a oraz użyciu `IRouteBuilder` 'a do mapowania wzorca URL (template) do kontrolera i akcji.

Po co nam Routing?

- SEO Friendly,
- URL nie musi odpowiadać fizycznej lokalizacji pliku,
- Długie adresy oraz rozszerzenia mogą być wyeliminowane,
- etc...

4. ROUTING

Route Templates

Szablony routingu używają literałów oraz tokenów. Literały są dopasowywane dokładnie do tekstu zawartego w adresie URL, natomiast tokeny są zamieniane w momencie znalezienia odpowiedniego Route'a.

Aby dopasować szablon, musi on zawierać przede wszystkim kontroler oraz akcję. Pozostałe tokeny z URL'a są mapowane do parametrów akcji używając mechanizmu Model Binding.

template: "{controller=Home}/{action=Index}/{id?}"

Segmenty

Adres URL może być podzielony na segmenty (części adresu URL – pomijając hostname i query string – rozdzielone znakiem „/”).

Przykładowo „<http://localhost/admin/orders>” - zawiera 2 segmenty.

Nie każdy segment musi być zmienną. Możemy definiować statyczne segmenty.

```
routes.MapRoute(  
    name: "default",  
    template: "klienci/{controller=Home}/{action=Index}/{id?}");
```

4. ROUTING

- **Routes**

W ASP.NET Core używamy Route, aby trafić do odpowiedniej akcji kontrolera. Każdy Route zawiera: *Name*, *Route Template*, *Defaults* oraz *Constraints*.

- **Route Collection**

Zbiór wszystkich Routes w naszej aplikacji.

- **Route Handler**

Komponent, który decyduje co zrobić z danym Route'm.

Kiedy mechanizm routingu dopasuje route dla nadchodzącego requesta, wywoływany jest powiązany RouteHandler i przekazuje tego route'a do dalszego przetwarzania.

Route Handler jest klasą, która implementuje *IRouteHandler*.

4. ROUTING

Data Tokens

Są dość mało znanym feature'm infrastruktury routingu. Są to dodatkowe wartości, które mogą być połączone z konkretnym route, ale nie wpływają na proces dopasowywania URL'a ani jego tworzenia.

```
routes.MapRoute(  
    name: "BindingModel",  
    template: "{controller=Binding}/{action=BindExample}/{category}",  
    defaults: null,  
    constraints: null,  
    dataTokens: new { Name = "binding_model" }  
);
```

Do czego używać data tokens?

- zostały stworzone, aby pomóc nam połączyć dane ('state data') z konkretnym route'm.
- nie są one wartościami dynamicznymi, więc nie zmieniają się w zależności od URL'a

Dzięki temu, możemy określić który route został wybrany do wykonania określonej akcji. Może być to przydatne w sytuacji, kiedy mamy kilka routes mapujących do tej samej akcji i musimy wiedzieć, który route został użyty.

4. ROUTING

Definiowanie ograniczeń :

1) ograniczenia są odseparowane od segmentów za pomocą znaku ":",
np. "{controller}/{action}/{id:int?}"

2) za pomocą argumentu **"constraints"** przekazywanego do metody "MapRoute"

```
routes.MapRoute(  
    name: "CustomRouteConstraint",  
    template: "{controller}/{action}/{day?}",  
    defaults: new { controller = "Routes", action = "CustomConstraint" },  
    constraints: new { day = new WeekDayConstraint() }  
);
```

3) za pomocą wyrażeń regularnych (REGEX)

```
routes.MapRoute(  
    name: "regexRoute",  
    template: "{controller:regex(^H.*)=Home}/{action:regex(^Index$|^About$)=Index}/{id?}";
```

4. ROUTING

Definiowanie ograniczeń :

4) Ograniczenia typu i wartości:

```
routes.MapRoute(  
    name: "myRoute",  
    template: "{controller=Home}/{action=Index}/{id:range(10,20)?}");
```

5) Łączenie wielu ograniczeń dla pojedynczego segmentu (za pomocą „:”)

```
routes.MapRoute(  
    name: "myRoute",  
    template: "{controller=Home}/{action=Index}/{id:alpha:minlength(6)?}");
```

lub za pomocą klasy **CompositeRouteConstraint**

```
routes.MapRoute(  
    name: "myRoute",  
    template: "{controller}/{action}/{id?}",  
    defaults: new { controller = "Home", action = "Index" },  
    constraints: new  
    {  
        id = new CompositeRouteConstraint(new IRouteConstraint[]  
        {  
            new AlphaRouteConstraint(),  
            new MinLengthRouteConstraint(6)  
        })  
    }  
);
```

4. ROUTING

```
private static IDictionary<string, Type> GetDefaultConstraintMap()
{
    return new Dictionary<string, Type>(StringComparer.OrdinalIgnoreCase)
    {
        // Type-specific constraints
        { "int", typeof(IntRouteConstraint) },
        { "bool", typeof(BoolRouteConstraint) },
        { "datetime", typeof(DateTimeRouteConstraint) },
        { "decimal", typeof(DecimalRouteConstraint) },
        { "double", typeof(DoubleRouteConstraint) },
        { "float", typeof(FloatRouteConstraint) },
        { "guid", typeof(GuidRouteConstraint) },
        { "long", typeof(LongRouteConstraint) },

        // Length constraints
        { "minlength", typeof(MinLengthRouteConstraint) },
        { "maxlength", typeof(MaxLengthRouteConstraint) },
        { "length", typeof(LengthRouteConstraint) },

        // Min/Max value constraints
        { "min", typeof(MinRouteConstraint) },
        { "max", typeof(MaxRouteConstraint) },
        { "range", typeof(RangeRouteConstraint) },

        // Regex-based constraints
        { "alpha", typeof(AlphaRouteConstraint) },
        { "regex", typeof(RegexInlineRouteConstraint) },

        { "required", typeof(RequiredRouteConstraint) },
    };
}
```


4. ROUTING

- **RESTful Routes**

Aby zadeklarować REST'owy kontroler, możemy użyć następującej konfiguracji routingu:

```
[Route("api/[controller]")]
public class ValuesController : Controller
{
    //GET api/values
    [HttpGet]
    public IEnumerable<string> GetValues()
    {
        return new string[] { "value1", "value2" };
    }
}
```

Dzięki takiej konfiguracji, nasz RESTowy webservice będzie akceptował żądania pod adresem „api/values”. Zamiast używać atrybutu Route na poziomie akcji, dekorujemy je odpowiednimi typami żądań ([HttpGet], [HttpPost], [HttpPut], [HttpDelete]).

4. ROUTING

ZADANIE

1. Utwórz klasę „inline route constraint” (wartość parametru powinna akceptować wyłącznie poprawne nazwy miesiący (january, february,...)).
2. Zdefiniuj route, który będzie przyjmował adresy url w formacie: „**blog/posts/{year}/{month}**”, gdzie {year} będzie liczbą z zakresu 2000-2100, natomiast {month} będzie korzystać z ograniczenia z pkt 1.
3. Napisz unit testy sprawdzające route constraint z pkt 1.
4. Sprawdź działanie route'a tworząc kontroler z akcją pasującą do zdefiniowanego szablonu route'a.

5. TAG HELPERS

Wprowadzenie

Tag Helpers są nowym feature'm wprowadzonym w ASP.NET Core MVC i są one klasami C# które:

- manipulują elementami HTML (np. poprzez dołączanie do nich dodatkowej treści lub podmieniając ich zawartość),
- umożliwiają generowanie lub transformowanie treści widoku za pomocą logiki w C#

```
<form asp-action="SubmitPost" asp-controller="Home" method="post">  
    ....  
    ....  
</form>
```



```
<form action="/home/SubmitPost" method="post">  
    ....  
    ....  
</form>
```

5. TAG HELPERS

- **Wprowadzenie cd.**

- Dzięki TagHelpers nie musimy już korzystać z Razor Helpers do tworzenia np. formularzy.

```
@Html.ActionLink("See blog", "Posts", "Blog")  
  
<a asp-action="Posts" asp-controller="Blog">See blog</a>
```

```
// Before - HTML Helpers  
@Html.ActionLink("Click me", "MyController", "MyAction", { @class="my-css-classname", data_my_attr="my-attribute"})  
  
// After - Tag Helpers  
<a asp-controller="MyController" asp-action="MyAction" class="my-css-classname" my-attr="my-attribute">Click me</a>
```

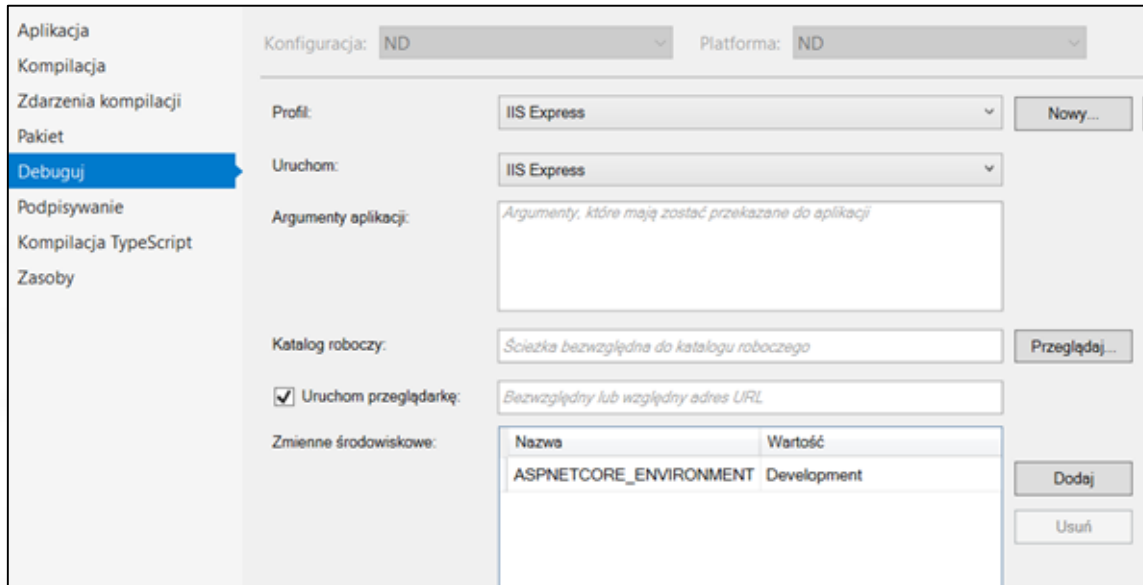
- Dzięki nowemu podejściu, nasze widoki wyglądają bardziej jak czysty HTML, a nie połączenie HTML'a z C#.

- ASP.NET Core MVC dostarcza nam wiele standardowych TagHelpers jak np.: Anchor Tag Helper, Form Tag Helper, Label Tag Helper, Input Tag Helper, Validation Tag Helper i inne...

5. TAG HELPERS

- **Environment Tag Helper**

Dzięki temu helperowi, zawartość jest renderowana wyłącznie gdy aplikacja jest wdrożona na dane środowisko. Dzieje się to za pomocą sprawdzenia zmiennej środowiskowej „ASPNETCORE_ENVIRONMENT”.



Konfiguracja: ND Platforma: ND

Profil: IIS Express [Nowy...]

Uruchom: IIS Express

Argumenty aplikacji: *Argumenty, które mają zostać przekazane do aplikacji*

Katalog roboczy: *Ścieżka bezwzględna do katalogu roboczego* [Przeglądaj...]

☒ Uruchom przeglądarkę: *Bezwzględny lub względny adres URL*

Zmienne środowiskowe:

Nazwa	Wartość
ASPNETCORE_ENVIRONMENT	Development

[Dodaj] [Usuń]

```
<environment exclude="Development">  
    ...  
</environment>  
<environment include="Staging,Production">  
    ...  
</environment>
```

5. TAG HELPERS

- Environment Tag Helper

Dostępne nazwy środowisk: Development, Staging, Production.
Możemy również stworzyć własne nazwy środowisk w zależności od potrzeb.

```
<environment exclude="Development">  
    ...  
</environment>  
<environment include="Staging,Production">  
    ...  
</environment>
```

Atrybuty:

- include: lista środowisk, dla których dany blok ma się wykonać,
- exclude: przeciwieństwo „include”,
- names: pozostałość po .NET Core 1.0 (nie zalecane do używania)

<https://blogs.msdn.microsoft.com/mvpawardprogram/2017/04/25/custom-environment-asp-net-core/>

5. TAG HELPERS

ENVIRONMENT TAG HELPER

- Jeżeli zmienna środowiskowa nie została ustawiona, wówczas domyślną wartością jest "Production"
- Zmienne środowiskowe ustawione w pliku "Properties\launchSettings.json" nadpisują wartości ustawione w zmiennych systemowych

```
"profiles": {  
  "IIS Express": {  
    "commandName": "IISExpress",  
    "launchBrowser": true,  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Development"  
    }  
  },  
  "WebApp1": {  
    "commandName": "Project",  
    "launchBrowser": true,  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Staging"  
    },  
    "applicationUrl": "http://localhost:54340/"  
  },  
}
```

Kiedy aplikacja uruchamiana jest za pomocą "dotnet run", zostanie użyty pierwszy profil z "commandName": "Project".

Wartość commandName określa jaki web server zostanie uruchomiony.

- IIS Express,
- IIS,
- Project (odpalany Kestrel)

5. TAG HELPERS

- Link & Script Tag Helpers

```
<environment exclude="Development">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
        asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
        asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

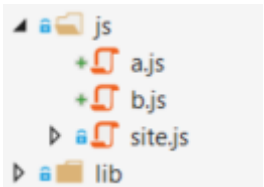
- Atrybut „**asp-fallback-src**” określa drugą lokalizację, z której można pobrać dany plik jeśli pierwsza lokalizacja jest niedostępna,
- Atrybut „**asp-append-version**” umożliwia nam wersjonowanie adresów URL plików (cache buster). „Version” jest wyliczany dla pliku tylko raz na podstawie algorytmu SHA256 i trzymany w ImemoryCache. Dzięki temu URL pliku zmieni się dopiero gdy jego wartość ulegnie zmianie.

5. TAG HELPERS

Referencje do wielu plików:

- asp-src-include,
- asp-src-exclude

Dzięki temu tagowi możemy za pomocą jednej linijki dołączyć zbiór plików z danej lokalizacji, np.:



```
<script src="/js/a.js"></script>
<script src="/js/b.js"></script>
<script src="/js/site.js"></script>
```

```
<script asp-src-include="/js/*.js"></script>
```

5. TAG HELPERS

- **Cache Tag Helper**

Cache Tag Helper cache'uje zawartość w pamięci „server-side” w zależności od ustawionego expiration date.

```
<cache expires-after="@TimeZone.FromMinutes(5)" vary-by-user="true">  
    @Html.Partial("MyPartial")  
</cache>
```

W powyższym przykładzie cache'ujemy PartialView na okres 5 minut i jest to rozróżniane na poziomie użytkownika.

Zasada działania jest taka, że ASP.NET generuje ID unikalne do kontekstu cache'owanej treści, dzięki temu możemy mieć wiele tagów <cache> na pojedynczej stronie i żaden nie ma wpływu na pozostałe.

(Więcej na temat Cache Tag Helper w sekcji 6)

5. TAG HELPERS

- **Wsparcie dla TagHelpers w widokach**

Plik „_ViewImports.cshtml” znajdujący się w strukturze projektu odpowiada za dostarczanie namespace'ów używanych przez wszystkie widoki.

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, SiiTraining
```

Wildcard (*) oznacza, że dołączamy wszystkie tag helpers dostępne w assembly „SiiTraining”.

Plik „_ViewImports.cshtml” można również dodać do każdego katalogu z widokami i na jego poziomie określać do których tag helpers chcemy mieć dostęp.

5. TAG HELPERS

- **Własny Tag Helper**

1. Tworzymy klasę dziedziczącą po **TagHelper** oraz właściwości klasy, które będą używane jako atrybuty dla naszego taga:

```
[HtmlTargetElement("img", Attributes = "gravatar-email")]
public class GravatarTagHelper : TagHelper
{
    [HtmlAttributeName("gravatar-email")]
    public string Email { get; set; }

    [HtmlAttributeName("gravatar-size")]
    public int Size { get; set; } = 50;
}
```

Aby ograniczyć użycie naszego helpera wyłącznie do tagów , używamy do tego atrybutu **HtmlTargetElement**

5. Tag Helpers

- Użycie atrybutu [**HtmlTargetElement**] umożliwia nam wpięcie naszej logiki w proces renderowania normalnych tagów HTML (jak w przykładzie: dla)
- Dzięki temu możemy zmieniać rezultat naszego HTML'a bez konieczności 'głębszych' zmian.
- Modyfikacja wszystkich tagów danego typu w całej aplikacji może zostać wykonana za pomocą stworzonego tag helpera (np. Dodanie title do obrazków, które ich jeszcze nie mają)

Minusy korzystanie z Tag Helpers?

- wyglądają jak zwykły HTML, więc niektórzy uważają, że ta składnia jest zbyt myląca,
- edytory tekstu nie wspierają syntaxu Tag Helpers (w przeciwieństwie do Visual Studio)

Rozwiązanie?

Prefixowanie używanych tag helpers za pomocą "@tagHelperPrefix":

- jeśli zadelkarujemy prefix na widoku, każdy tag helper który nie będzie z niego korzystał zostanie zignorowany przez Razor engine

```
@tagHelperPrefix "helper:"
```

```
<helper:no-follow href="http://www.wp.pl">wp.pl</helper:no-follow>
```


5. TAG HELPERS

2. Aby zdefiniować zachowanie naszego tag helpera, musimy przeciążyć metodę „Process” i dodać tam pożądaną logikę.

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    using (var md5 = MD5.Create())
    {
        var result = md5.ComputeHash(Encoding.ASCII.GetBytes(Email));
        var hash = BitConverter.ToString(result).Replace("-", "").ToLower();
        var url = $"http://gravatar.com/avatar/{hash}";
        var queryBuilder = new QueryBuilder();
        queryBuilder.Add("s", Size.ToString());
        url = url + queryBuilder.ToQueryString();
        output.Attributes.SetAttribute("src", url);
    }
}
```

W powyższym przykładzie, rezultatem będzie wygenerowanie atrybutu „src” z odpowiednią wartością, którą wygenerowaliśmy za pomocą logiki w C#.

5. TAG HELPERS

Jak dodać treść pomiędzy taga?

```
output.Content.SetContent(DateTime.ToString("MMM d, yyyy h:mm tt"));
```

Jak dodać zawartość jeśli tag nie definiuje żadnej treści „w środku”?

```
var childContent = await output.GetChildContentAsync();  
if (childContent.IsEmptyOrWhiteSpace)  
{
```

5. TAG HELPERS

ĆWICZENIE

a) Stworzyć własny Tag Helper, którego wywołanie będzie następujące:

```
@{  
    var exampleDate = new DateTime(2018, 01, 02, 13, 50, 11, DateTimeKind.Utc);  
}  
  
<time asp-date-time="@exampleDate" />
```

a rezultat wyrenderowanego rezultatu:

```
@*<time datetime="2018-01-02T13:50:11Z" title="Sunday, January 2, 2018 01:50 PM UTC">January 2, 2018 1:50 PM</time>*@
```

b) Napisz testy jednostkowe sprawdzające poprawność działania stworzonego tag helpera

6. CACHE

Cachowanie danych różni się w zależności od tego, czy strona postawiona jest tylko na 1 serwerze, czy też uruchomiona jest na serwerach z load balancer'em.

Dwie wysokopoziomowe opcje do wyboru:

- obsługiwać cache na każdym serwerze osobno;
- używać scentralizowanego cache'a, do którego każdy serwer będzie mieć dostęp. (distributed cache)



ASP.NET Caching

6. CACHE

Wady utrzymywania cache'a na każdym serwerze osobno:

- różnica pomiędzy tym co trzymamy w cache na każdym z serwerów,
- trudno jest inwalidować zawartość cache'a na X serwerach,
- niepotrzebne zużycie pamięci RAM na zduplikowane dane.



6. CACHE

IMemoryCache oraz IDistributedCache

- Oba interfejsy są wbudowanymi mechanizmami w .NET Core do cache'owania danych.

```
public class HomeController : Controller
{
    private readonly IMemoryCache _memoryCache;

    public HomeController(IMemoryCache memoryCache)
    {
        _memoryCache = memoryCache;
    }

    [HttpGet]
    public string Get()
    {
        var cacheKey = "TheTime";
        DateTime existingTime;
        if (_memoryCache.TryGetValue(cacheKey, out existingTime))
        {
            return "Fetched from cache : " + existingTime.ToString();
        }
        else
        {
            existingTime = DateTime.UtcNow;
            _memoryCache.Set(cacheKey, existingTime);
            return "Added to cache : " + existingTime;
        }
    }
}
```

```
public class DistributedController : Controller
{
    private readonly IDistributedCache _distributedCache;

    public DistributedController(IDistributedCache distributedCache)
    {
        _distributedCache = distributedCache;
    }

    [HttpGet]
    public async Task<string> Get()
    {
        var cacheKey = "TheTime";
        var existingTime = _distributedCache.GetString(cacheKey);
        if (!string.IsNullOrEmpty(existingTime))
        {
            return "Fetched from cache : " + existingTime;
        }
        else
        {
            existingTime = DateTime.UtcNow.ToString();
            _distributedCache.SetString(cacheKey, existingTime);
            return "Added to cache : " + existingTime;
        }
    }
}
```

6. CACHE

IMemoryCache

Jak korzystać z IMemoryCache ?

1) Wywołanie metody „**AddMemoryCache**” w ConfigurationServices:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddMemoryCache();
}
```

2) Wstrzyknięcie do konstruktora kontrolera interfejsu IMemoryCache i posługiwanie się dostępnymi metodami

```
private readonly IMemoryCache memoryCache;
private readonly IRepository repository;

private readonly IDistributedCache distributedCache;

public CachingController(IMemoryCache memoryCache, IRepository repository)
{
    this.memoryCache = memoryCache;
    this.repository = repository;
}
```

6. CACHE

IMemoryCache

Methods

<code>CreateEntry(Object)</code>	Create or overwrite an entry in the cache.
<code>Remove(Object)</code>	Removes the object associated with the given key.
<code>TryGetValue(Object, Object)</code>	Gets the item associated with this key if present.

Extension Methods

<code>Get(IMemoryCache, Object)</code>
<code>Get<TItem>(IMemoryCache, Object)</code>
<code>GetOrCreate<TItem>(IMemoryCache, Object, Func<ICacheEntry, TItem>)</code>
<code>GetOrCreateAsync<TItem>(IMemoryCache, Object, Func<ICacheEntry, Task<TItem>>)</code>
<code>Set<TItem>(IMemoryCache, Object, TItem)</code>
<code>Set<TItem>(IMemoryCache, Object, TItem, MemoryCacheEntryOptions)</code>
<code>Set<TItem>(IMemoryCache, Object, TItem, IChangeToken)</code>
<code>Set<TItem>(IMemoryCache, Object, TItem, DateTimeOffset)</code>
<code>Set<TItem>(IMemoryCache, Object, TItem, TimeSpan)</code>
<code>TryGetValue<TItem>(IMemoryCache, Object, TItem)</code>

6. CACHE

- **PostEvictionCallback**

Delegat umożliwiający nam rejestrację akcji, która zostanie wywołana za każdym razem, kiedy dane z cache'a się „przedawnią”.

```
memoryCache.Set("key", "test",  
    new MemoryCacheEntryOptions()  
        .RegisterPostEvictionCallback((key, value, reason, state) => { /*Do Something Here */ })  
    );
```

- **CancellationToken**

MemoryCache wspiera także obsługę CancellationToken. Mogą być przydatne np. do inwalidowania danych w cache'u.

```
var cts = new CancellationTokensource();  
memoryCache.Set("key", "test",  
    new MemoryCacheEntryOptions()  
        .AddExpirationToken(new CancellationChangeToken(cts.Token))  
    );
```

6. CACHE

- **IDistributedCache**

Dla aplikacji korzystających z web farms lepiej skorzystać z rozwiązań oferowanych przez IDistributedCache.

Różnice w porównaniu do IMemoryCache:

- dodatkowe asynchroniczne metody,
- metody odświeżania (refresh): resetuje sliding expiration bez pobierania danych,
- zwraca bajty zamiast obiektów (w związku z tym musimy serializować i deserializować obiekty, które trafiają do cache'a oraz są z niego pobierane,
- DistributedCacheEntryOptions oferują absolute oraz sliding expiration (jak dla IMemoryCache), ale nie ma dostępu do "token based expiration".

6. CACHE

Dostępne 3 implementacje IDistributedCache:

In memory, Redis, SQL Server

- **Implementacja „In memory” :**
 - Jest dostępna w pakiecie MVC (element z Microsoft.Extensions.Caching.Memory),
 - Można dodać ją też ręcznie
- **Implementacja Redis version:**
 - Dostępny w pakiecie NuGet "Microsoft.Extensions.Caching.Redis.Core"

```
services.AddDistributedMemoryCache();

services.AddDistributedRedisCache((x) => x.Configuration = ...);

services.AddDistributedSqlServerCache((x) => { x.ConnectionString = .... })
```

6. CACHE

- **Partial Pages**

Oprócz cache'owania danych po stronie serwera, możemy cachować wyrenderowany wynik strony (output). Partial page caching jest dostępny dzięki wbudowanemu mechanizmowi Caching Tag Helpers.

- Użycie tagów <cache>...</cache> spowoduje, że treść będąca wewnątrz zostanie dodana do cache'a na okres domyślny (20 minut).
- Dobrym przykładem do cache'owania są View Components, których renderowanie może być kosztowne (np. Poprzez ciągłe odpytywanie bazy danych)

```
<cache expires-after="@TimeSpan.FromSeconds(30)">...content to be cached...</cache>  
  
<distributed-cache name="cache" expires-after="@TimeSpan.FromSeconds(60)" vary-by-user="true">...cache something...</distributed-cache>
```

Więcej informacji: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/cache-tag-helper>

6. CACHE

Cache Attributes

- "Vary-by-user" - używamy tego atrybutu do cachowania różnych treści dla każdego zalogowanego użytkownika. Nazwa zalogowanego użytkownika zostanie dodana do cache'a.
- "Vary-by-route" - cachowanie treści w oparciu o zbiór parametrów z route data. Atrybut oczekuje listy (oddzielonych przecinkami) route data (nazw parametrów).

```
<cache vary-by-route="id">  
  <!--View Component or something that gets data from the database-->  
  *last updated @DateTime.Now.ToLongTimeString()  
</cache>
```

W zależności od parametru "id" cache'ujemy różną treść.

- "Vary-by-query" - cache'uje dane w zależności od parametrów query dla danego requestu.
- "Vary-by-cookie" - cache'uje dane w zależności od wartości trzymanej w cookies.
- "Vary-by-header" - cache'uje dane w zależności od wartości dla danego nagłówka headera (np. "User-Agent")
- "Vary-by" - cache'uje dane w zależności od wartości jakiegokolwiek wartości którą prześlemy (string)

Complex Keys

Możemy zdefiniować dowolną ilość parametrów "vary-by" tworząc w ten sposób klucz kompozytowy.

```
<cache vary-by-user="true" vary-by-route="id">  
  <!--View Component or something that gets data from the database-->  
  *last updated @DateTime.Now.ToLongTimeString()  
</cache>
```

6. CACHE

Najwyższy poziom cache'owania, którego możemy użyć jest cache'owanie całej strony. Możemy cache'ować całe strony po stronie przeglądarki, jak również po stronie serwera.

W .NET Core mamy dostępne:

- **ResponseCache attribute** do ustawiania nagłówków cache'a (browser side)
- **ResponseCaching middleware** (server side)

Full pages caching :

```
[ResponseCache(Duration = 60)]  
public IActionResult Contact()  
{  
    ViewData["Message"] = "Your contact page.";  
    return View();  
}
```

Produces the following headers:

```
Cache-Control: public,max-age=60
```

6. CACHE

- Atrybut **[ResponseCache]** określa parametry niezbędne do ustawiania odpowiednich nagłówków w response caching. Zastępuje on atrybut "[OutputCache]" z wcześniejszych wersji ASP.NET MVC

Właściwość	Opis
CacheProfileName	Nazwa profilu cache'owania
Duration	Czas cache'owania (w sekundach). Ustawia „max-age” w nagłówku „cache-control”.
Location	Miejsce gdzie dane z URL'a mają być cache'owane.
NoStore	Czy dane mają być przechowywane
VaryByHeader	Wartość dla „Vary” nagłówka odpowiedzi
VaryByQueryKeys	Query Keys dla „Vary by”

6. CACHE

VaryByQueryKeys string[]

Response Caching middleware rozróżnia zapisane odpowiedzi na podstawie listy query keys.

Aby móc korzystać z **VaryByQueryKeys**, middleware musi być uruchomiony.

Aby zwrócić cachowany rezultat zarówno query string oraz jego wartość muszą zgadzać się z poprzednim requestem.

Response Cache

Request	Result
<code>http://example.com?key1=value1</code>	Returned from server
<code>http://example.com?key1=value1</code>	Returned from middleware
<code>http://example.com?key1=value2</code>	Returned from server

6. CACHE

Zamiast duplikowania ustawień **[ResponseCache]** na wielu atrybutach akcji kontrolerów, można skonfigurować "cache profiles" (w ConfigureServices)

- **Cache Profile**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default",
            new CacheProfile()
            {
                Duration = 60
            });
        options.CacheProfiles.Add("Never",
            new CacheProfile()
            {
                Location = ResponseCacheLocation.None,
                NoStore = true
            });
    });
}
```

```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
}
```

6. CACHE

Response Caching Middleware

Middleware określający kiedy odpowiedzi są cache'owane, zapisuje odpowiedzi i dostarcza je jako odpowiedzi z cache'a.

Wymagane pakiety:

Aby dołączyć middleware do projektu, potrzebujemy pakietu:

- Microsoft.AspNetCore.ResponseCaching
- lub Microsoft.AspNetCore.All

```
services.AddResponseCaching();
```

Konfiguracja:

- Dodanie middleware do kolekcji serwisów w ConfigureServices
- Wywołanie metody "UseResponseCaching()" w "Configure"

6. CACHE

Response Caching Middleware

- middleware cache'uje wyłącznie odpowiedzi ze statusem 200 (OK)
- Odpowiedzi, które zawierają treść dla zautentykowanych klientów muszą być oznaczone jako 'non cacheable'

Middleware oferuje 3 opcje do kontrolowania cache'owania odpowiedzi:

Opcja	Opis
UseCaseSensitivePaths	Określa czy odpowiedzi są cache'owane dla ścieżek „case-sensitive”. (domyślnie: false)
MaximumBodySize	Największy rozmiar cache'owanej odpowiedzi w bajtach (domyślnie 64 MB)
SizeLimit	Limit dla cache'owanych odpowiedzi w middleware w bajtach (domyślnie 100 MB)

6. CACHE

- Przykład:

```
services.AddResponseCaching(options =>
{
    options.UseCaseSensitivePaths = true;
    options.MaximumBodySize = 1024;
});
```

Powyższa konfiguracja ustawia middleware w ten sposób, aby cache'ował odpowiedzi o maksymalnym rozmiarze 1024 bajtów używając ścieżek „case-sensitive”, więc odpowiedzi dla adresów „/strona1”, „Strona2” będą przechowywane osobno.

6. CACHE

```
[ResponseCache(Duration = 30)]  
public IActionResult Article()  
{  
  
    return View();  
}
```

Headers	Cookies	Params	Response	Timings	Preview
Request URL: /article/914182					
Request method: GET					
Remote address: 23.101.67.245:80					
Status code: 200 OK				Edit and Resend	Raw headers
Version: HTTP/1.1					
Filter headers					
Response headers (0,249 KB)					
Cache-Control: "public,max-age=30"					
Content-Encoding: "gzip"					
Content-Type: "text/html; charset=utf-8"					
Date: "Wed, 08 Mar 2017 14:30:45 GMT"					
Server: "Microsoft-IIS/8.0"					
Transfer-Encoding: "chunked"					
Vary: "Accept-Encoding"					

6. CACHE

Wymagania konieczne, aby cache został użyty:

- odpowiedź musi mieć status 200 (OK),
- request musi być GET lub HEAD,
- Middleware taki jak np. "Static File Middleware" nie może przetworzyć odpowiedzi przed Cache Middleware,
- Nie może występować Authorization Header,
- Nie może występować nagłówek "Set-Cookie",
- I inne...
<https://docs.microsoft.com/en-us/aspnet/core/performance/caching/middleware?tabs=aspnetcore2x>

Problemy z ciasteczkami:

Obecnie nie da się używać ciasteczek z response caching. Tak długo jak nagłówek cookie jest obecny, wówczas nie da się wykorzystać response cache'ingu.

Usługi takie jak Application Insights oraz system Antiforgery w ASP.NET Core nie działają z response cache.

6. CACHE

ĆWICZENIE

1. Włącz w aplikacji webowej korzystanie z MemoryCache,
2. Wykorzystaj możliwość MemoryCache na poziomie kontrolera (przykład dodania/czytania z cache/usuwanie/reagowanie na zdarzenie usunięcia),
3. Zaimplementuj różne rodzaje cacheExpirations i sprawdź ich poprawność działania w praktyce,
4. Napisz test jednostkowy, sprawdzający poprawność działania akcji kontrolera w połączeniu z MemoryCache i np. Repozytorium (przykładowo: jeśli dane nie znajdują się aktualnie w cache'u – pobierz dane z repozytorium, w przeciwnym razie wykorzystaj cache'a).

7. WALIDATORY

- **Walidacja modelu – w jakim celu ?**

Zanim nasza aplikacja zapisze dane np. w bazie danych, muszą one zostać zwalidowane. Sprawdzamy różne kryteria, jak np.:

- kwestie bezpieczeństwa (przemykanie kodu javascript, kodu HTML, etc, SQL Injection),
- typ i rozmiar danych,
- własne wymagania dotyczące danego pola.

W aplikacji MVC walidacja odbywa się po 2 stronach – zarówno u klienta, jak i na serwerze.

W .NET wykorzystujemy atrybuty walidacyjne w celu walidacji danych.

7. WALIDATORY

- **Atrybuty walidacyjne**

Wykorzystania atrybutów walidacyjnych pozwala nam skonfigurować walidację na poziomie modelu. Walidujemy zarówno wymagany typ danych, jak również to czy pole jest wymagane.

Popularne wbudowane atrybuty walidacji to np.:

[CreditCard], [Compare], [EmailAddress], [Phone], [Range], [RegularExpression], [Required], [StringLength], [Url], etc...

7. WALIDATORY

- **Własne walidatory**

Jeśli chcemy stworzyć swój własny walidator mamy do wyboru 2 podejścia:

- stworzenie własnego atrybutu walidacyjnego który dziedziczy po **ValidationAttribute**,
- zmiana swojej klasy modelu, aby implementowała interfejs **IValidatableObject**

```
public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator
{
    private int _year;

    public ClassicMovieAttribute(int Year)
    {
        _year = Year;
    }

    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        Movie movie = (Movie)validationContext.ObjectInstance;

        if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
    {
        yield return new ValidationResult(
            $"Classic movies must have a release year earlier than {_classicYear}.",
            new[] { "ReleaseDate" });
    }
}
```

7. WALIDATORY

- **Remote validation**

Zdalne walidatory są przydatne w sytuacji, jeśli chcemy zwalidować dane po stronie klienta z danymi na serwerze (np. unikalność nazwy użytkownika).

```
[Remote(action: "VerifyEmail", controller: "Users")]  
public string Email { get; set; }
```

```
[AcceptVerbs("Get", "Post")]  
public IActionResult VerifyEmail(string email)  
{  
    if (!_userRepository.VerifyEmail(email))  
    {  
        return Json($"Email {email} is already in use.");  
    }  
  
    return Json(true);  
}
```

7. WALIDATORY

- **Remote validation**

Odpowiedź serwera musi być w formacie JSON, który zwraca „true” dla poprawnych elementów, natomiast dla nieprawidłowych może zwracać „false”, undefined, null lub zawierać własny komunikat błędu, który zostanie wyświetlony zamiast domyślnego.

Atrybut [Remote] może zawierać dodatkową właściwość „AdditionalFields”, której używamy jeśli walidujemy kombinację pól z modelu.

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName))]  
public string FirstName { get; set; }  
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName))]  
public string LastName { get; set; }
```

W takim przypadku metoda, która odpowiada za walidację musi przyjmować 2 argumenty odpowiadające nazwami.

7. WALIDATORY

- **Model State**

Model state zawiera błędy walidacji w wartościach z przesłanego formularza HTML. MVC waliduje kolejne pola tak długo, aż osiągnie maksymalną ilość błędów (domyślnie 200). Można tę liczbę zmienić w następujący sposób:

```
services.AddMvc(options => options.MaxModelValidationErrors = 50);
```

- **Obsługa błędów z Model State**

Walidacja modelu odbywa się na każdej akcji i to akcja kontrolera musi sprawdzić jaki jest aktualny stan ModelState.IsValid aby odpowiednio na niego zareagować.

Możemy również wymusić ponowną walidację modelu z poziomu ActionMethod wywołując metodę: **TryValidateModel**

```
TryValidateModel(movie);
```


7. WALIDATORY

ĆWICZENIE

1. Stwórz walidator który będzie implementował logikę walidacji na 2 przedstawione sposoby:

a) **ValidationAttribute**

b) **IValidatableObject**

Walidator powinien sprawdzać komentarz do bloga pod kątem występowania w treści adresów email oraz stron Internetowych. Jeśli któryś z tych elementów zostanie wykryty, powinniśmy zwrócić użytkownikowi komunikat błędu.

2. Stwórz akcję kontrolera (POST), która przyjmuje jako parametr model, który używa stworzonego walidatora i przetestuj jego działanie za pomocą np. POSTMAN'a.

3. Napisz metodę, która będzie sprawdzać unikalność wprowadzonego tytułu (możesz zamockować) i użyć jej w Remote Validator'ze. Sprawdź działanie rozwiązania.

8. VIEW COMPONENTS

Kilka słów wstępu

W standardowym ASP.NET MVC często używanymi feature'ami są Partial Views oraz Child Actions.

Partial Views umożliwiają nam tworzenie reużywalnych komponentów, których możemy używać w wielu widokach.

Są też akcje kontrolera oznaczone jako ChildAction i nie można ich wywołać z poziomu URL'a, natomiast można wewnątrz widoków lub partial views.

W ASP.NET Core Child Actions nie są już dostępne.
Nowym sposobem implementacji jest feature o nazwie **View Components**.

8. VIEW COMPONENTS

- **Czym są View Components?**

Jeden z fajniejszych nowych features zaprezentowanych w ASP.NET Core. Różnica pomiędzy View Components a Partial Views jest taka, że nowe podejście umożliwia nam między innymi: **asynchroniczność, proste używanie dependency injection, rozdzielanie zależności.**

Dzięki temu tworzenie takich komponentów sprawia, że **są prostsze w utrzymaniu i są testowalne.**

View Components mogą przyjmować parametry i zawierać w sobie logikę biznesową. Komponent składa się z 2 elementów:

- klasy zawierającej logikę komponentu,
- widoku, który zwraca widok (w większości przypadków)

8. VIEW COMPONENTS

- **Jak tworzyć View Components?**

Class name ends
with
ViewComponent

Use attribute
[ViewComponent]

Derive from
ViewComponent
class

1. Klasa z suffixem w nazwie „ViewComponent”.

ASP.NET Core rozpoznaje suffix i renderuje taką klasę jako View Component. Nie mamy tutaj dostępu do takich danych jak np. kontekst MVC itp.

```
public class ProductsSummaryViewComponent
{
    public string Invoke()
    {
        return $"We have {repository.GetAllProducts().Count} products in our store";
    }
}
```

8. VIEW COMPONENTS

2. Za pomocą atrybutu [ViewComponent]

Wystarczy użyć atrybutu [ViewComponent] nad dowolną klasą.

```
[ViewComponent]
public class SimpleClass
{
    public void InvokeAsync()
    {
        //some logic
    }
}
```

3. Dziedziczenie po klasie ViewComponent (zalecane)

Najprostszy i najbardziej efektywny sposób na tworzenie View Component z dostępem do wielu informacji jakie daje nam framework MVC jak HttpContext, TempData, ModelState etc. To rozwiązanie umożliwia nam również na używanie widoków podobnie jak w przypadku standardowego kontrolera MVC.

```
public class ProductSummaryWithContextData : ViewComponent
{
    public IActionResult Invoke()
    {
        var manufacturer = RouteData.Values["id"] as string;
        //some logic here...

        return View(new ProductSummaryViewModel()
        {
            CategoriesCount = 2,
            ProductsCount = 5,
            Manufacturer = manufacturer
        });
    }
}
```

8. VIEW COMPONENTS

Jak tworzymy UI (widoki) dla View Components?

Są to klasyczne widoki razor'owe. Muszą one spełniać 2 wymagania:

- nazwa pliku z widokiem musi brzmieć: „Default.cshtml”,
- lokalizacja pliku musi znajdować się w jednej z lokalizacji:

a) `~/Views/Shared/Components/[ViewComponentName]/Default.cshtml`

b) `~/Views/[ControllerName]/Components/[ViewComponentName]/Default.cshtml`

W przypadku używania komponentu w wielu kontrolerach, powinniśmy użyć lokalizacji z „Shared”, natomiast jeśli chcemy ograniczyć dostępność do konkretnego kontrolera używamy opcji b)

Nazwa pliku nie musi być koniecznie „Default.cshtml”, wówczas zwracając widok należy podać nazwę pliku „return View(„Test”, model)”

8. VIEW COMPONENTS

- Jak używać (wywoływać) View Components?

Najprostszym sposobem jest wywołanie w dowolnym miejscu:

```
@await Component.InvokeAsync("ProductSummary")
```

Pierwszym parametrem jest nazwa komponentu, a następnie parametry, których użyliśmy przy implementacji metody `InvokeAsync`. Jako, że używamy asynchronicznych metod, musimy użyć słowa „await” aby komponent został wyrenderowany na ekranie.

View Component może być również używany na poziomie kontrolera:

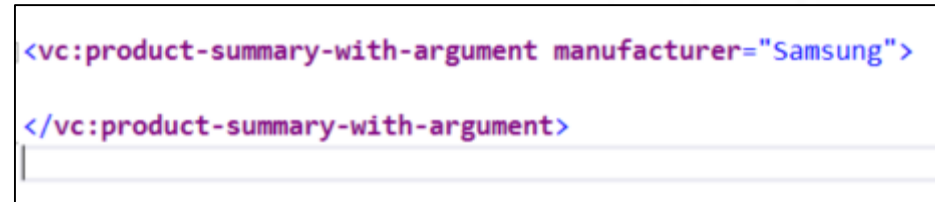
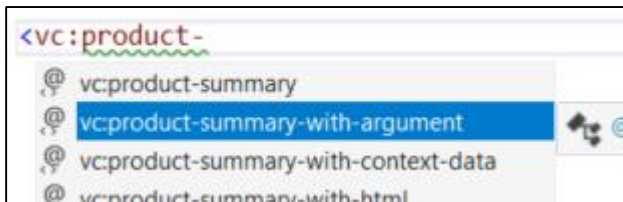
```
public IActionResult IndexWithVC()
{
    return ViewComponent("PageSize");
}
```

▲ 3 z 4 ▼ `ViewComponentResult Controller.ViewComponent(string componentName, object arguments)`
Creates a `ViewComponentResult` by specifying the name of a view component to render.
arguments: An object with properties representing arguments to be passed to the invoked view comp

8. VIEW COMPONENTS

- **Używanie View Components jako TagHelpers**

ASP.NET Core umożliwia nam jeszcze wywoływanie naszych komponentów za pomocą tag helper'ów.



Proste i czyste użycie jeszcze bardziej przypominające standardowe wywołanie znaczników HTML.

Aby używać komponentu za pomocą tag helpera, tag musi rozpoczynać się od „vc”, natomiast nazwa komponentu i parametry są przetransformowane na małe znaki (kebab case – tzn nazwa typu ProductSummary jest zmieniana na „product-summary”).

W celu używania View Components jako Tag Helpers musimy użyć dyrektywy

8. VIEW COMPONENTS

ĆWICZENIE

1. Utwórz View Component na 3 sposoby:
 - nazwa klasy kończąca się na "ViewComponent" implementująca metodę Invoke(),
 - przy użyciu atrybutu [ViewComponent],
 - dziedzicząca po klasie ViewComponent
2. Komponent powinien wyświetlać ostatnio dodane posty do bloga za pomocą wstrzykniętego interfejsu repozytorium.
3. Napisz test jednostkowy sprawdzające poprawność wygenerowanego modelu przez komponent.

9. GLOBALIZACJA

Internacjonalizacja aplikacji Internetowej zawiera w sobie **Globalizację** oraz **Lokalizację**.

Globalizacja

Proces projektowania aplikacji, która wspiera różne kultury. Globalizacja dodaje wsparcie dla takich elementów jak "input", "display" oraz "output" dla zbioru zdefiniowanych języków.

Lokalizacja

Proces adoptowania już zglobalizowanej aplikacji do danej kultury (culture/locale).

- a) dostosowanie treści aplikacji do lokalizacji,
- b) dostarczenie zlokalizowanych resource'ów dla języków i kultur które ma wspierać aplikacja,
- c) implementacja strategii wyboru danego języka/kultury dla każdego requestu

Dostosowanie treści aplikacji do lokalizacji

- ASP.NET Core dostarcza 2 interfejsy wspierające proces tworzenia zlokalizowanych aplikacji: **IStringLocalizer** oraz **IStringLocalizer<T>**,
 - Używają one **ResourceManager**'a oraz **ResourceReader**'a do dostarczania "culture-specific" resources w run-time'ie,
 - W przypadku jeśli localizer nie znajdzie wartości dla podanego klucza, zwróci nam domyślną wartość, czyli klucz. Dzięki temu podejściu możemy skupić się na tworzeniu aplikacji, bez konieczności tworzenia defaultowego pliku z resource'ami.

```
public class AboutController : Controller
{
    private readonly IStringLocalizer<AboutController> _localizer;

    public AboutController(IStringLocalizer<AboutController> localizer)
    {
        _localizer = localizer;
    }

    [HttpGet]
    public string Get()
    {
        return _localizer["About Title"];
    }
}
```

View localization

- **ViewLocalizer** dostarcza zlokalizowane stringi dla widoków. Klasa **ViewLocalizer** implementuje ten interfejs i szuka ścieżki z resource'ami na podstawie ścieżki pliku z widokiem.

```
@using Microsoft.AspNetCore.Mvc.Localization

@Inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.StarterWeb.Services

@Inject IViewLocalizer Localizer
@Inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

Treści błędów dla DataAnnotation lokalizuje się z pomocą **IStringLocalizer<T>**. Definiując opcję `ResourcesPath = "Resources"`, treści wiadomości dla przykładowego modelu "RegisterViewModel" mogą być przechowywane w następujących lokalizacjach:

- `Resources/ViewModels.Account.RegisterViewModel.fr.resx`
`Resources/ViewModels/Account/RegisterViewModel.fr.resx`

Jak używać jednego źródła treści dla atrybutów walidacyjnych? :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```


➤ Implementacja strategii wyboru języka/kultury dla każdego requesta

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

✓ **"AddLocalization"-**

dodaje usługę lokalizacji do kontenera oraz ustawia ścieżkę do resource'ów

✓ **"AddViewLocalization" –**

dodaje wsparcie dla zlokalizowanych plików widoków (np. "Index.pl.cshtml" - w przypadku opcji "Suffix")

✓ **"AddDataAnnotationsLocalization"**

dodaje wsparcie dla lokalizacji wiadomości walidacyjnych z DataAnnotations

Localization middleware

- **UseRequestLocalization** inicjalizuje obiekt **RequestLocalizationOptions**.
Przy każdym request'cie sprawdzana jest lista dostępnych **RequestCultureProvider**,
a następnie pierwszy provider, który może skutecznie określić kulturę jest użyty.

```
app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture(enUSCulture),
    // Formatting numbers, dates, etc.
    SupportedCultures = supportedCultures,
    // UI strings that we have localized.
    SupportedUICultures = supportedCultures
});
```

QueryStringRequestCultureProvider

CookieRequestCultureProvider

AcceptLanguageHeaderRequestCultureProvider

QUERY STRING REQUEST CULTURE PROVIDER

- Niektóre aplikacje używają query string'a do ustawiania kultury. Ten provider ustawiony jest jako domyślny.
- Jako query stringi przekazujemy "culture" lub/oraz "ui-culture"
- Jeśli prześlemy wyłącznie jedną z powyższych wartości w query stringu, provider będzie używał jej do ustawienia zarówno "culture" oraz "ui-culture".

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

COOKIE REQUEST CULTURE PROVIDER

- Aplikacje działające produkcyjnie, bardzo często dostarczają mechanizm, który pozwala ustawić obecną kulturę w ciasteczkach,
- Metoda "MakeCookieValue" służy do tworzenia ciasteczek,
- `CookieRequestCultureProvider.DefaultCookieName` zwraca nam domyślną nazwę ciasteczka używaną do przenoszenia informacji o kulturze użytkownika. Domyślna nazwa to `".AspNetCore.Culture"`

- Format ciasteczka to:

```
c=%LANGCODE%|uic=%LANGCODE%
```

gdzie

`c` = culture, `uic` = `UICulture`

THE ACCEPT-LANGUAGE HTTP HEADER

Nagłówek "Accept-Language" jest możliwy do ustawienia w większości przeglądarek.



WŁASNY CULTURE PROVIDER

- Po co nam własny Culture Provider?

Przykładowo, jeśli chcemy zapisywać ustawienia języka naszych użytkowników w bazie danych.

- Jak dodać własnego culture provider'a?

```
options.DefaultRequestCulture = new RequestCulture(culture: enusCulture, uiculture: enusCulture);
options.SupportedCultures = supportedCultures;
options.SupportedUICultures = supportedCultures;

options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
{
    // My custom request culture logic
    return new ProviderCultureResult("en");
}));
```

9. GLOBALIZACJA

ĆWICZENIE

- a) Skonfiguruj aplikację MVC, aby wspierała minimum 2 języki (np. Polski/Angielski)
- b) Utwórz osobny kontroler, w którym będziesz testować działanie (IStringLocalizer),
- c) Utwórz widok, który będziesz lokalizować (IViewLocalizer),
- c) Stwórz pliki Resources dla wszystkich języków zarówno dla kontrolera oraz widoku,
- d) Stwórz partial view, który umożliwi wybór jednego ze wspieranych języków,
- e) Stwórz metodę, która ustawi w Cookies wybrany język na podstawie wartości przekazanej do metody (wywołujemy tą metodę z poziomu formularza w partial view z pkt d)

Część 2

PROGRAMOWANIE ASYNCHRONICZNE

ASP.NET CORE MVC

1. PROGRAMOWANIE ASYNCHRONICZNE

- **Task Parallel Language (TPL)** - programowanie asynchroniczne dostępne od .NET 4.5,
- Słowa kluczowe: **async**, **await**,
- Tworzenie asynchronicznych aplikacji Internetowych oraz Web API za pomocą **async/await**. Action methods w kontrolerach powinny używać **async** w sygnaturze metody, natomiast metoda powinna zwracać **Task** zawierający **ActionResult**.

```
public async Task<BlogPost> GetPostAsync(int id)
{
    var post = await context.Posts.FindAsync(id);
    return post;
}
```

1. PROGRAMOWANIE ASYNCHRONICZNE

Kiedy stosować metody asynchroniczne?

- gdy aplikacja będzie szybciej działać przy wykorzystaniu metod asynchronicznych,
- gdy operacja opiera się w głównej mierze na operacjach typu I/O (zapis/odczyt) na dysku lub też operacjach sieciowych, których wolne działanie może powodować blokadę wątku,
- aby móc zaimplementować mechanizm zatrzymywania/przerywania długich żądań

1. PROGRAMOWANIE ASYNCHRONICZNE

Użycie **CancellationToken** w akcjach kontrolera

Problem: Jak przerwać wykonywanie długo trwającego żądania w przypadku kiedy użytkownik przerwie wykonywanie requesta z poziomu swojej przeglądarki (np. wstrzymując ładowanie strony, odświeżając ją etc.)?

Rozwiązanie: użycie **CancellationToken** w akcji kontrolera, który zawiera "long-running task".

1. PROGRAMOWANIE ASYNCHRONICZNE

Użycie **CancellationToken** w akcjach kontrolera

ASP.NET Core dostarcza mechanizm dla web server'a (np. **Kestrel**) do sygnalizowania kiedy request został anulowany przy wykorzystaniu **CancellationToken**.

(przykład – [AsyncSampleController.cs](#))

Uwaga! : Rozwiązanie nie zachowuje się tak samo w przypadku uruchamiania aplikacji z poziomu IIS/IIS Express - działa dobrze na Kestrel webserver.

1. PROGRAMOWANIE ASYNCHRONICZNE

Sprawdzanie **cancellation state**

- `cancellationToken.ThrowIfCancellationRequested();`
- `cancellationToken.IsCancellationRequested`

Obsługa cancellation za pomocą **ExceptionHandler** :

W przypadku korzystania z `CancellationToken`'ów do obsługi anulowanych requestów, warto obsłużyć wyjątki, żeby np. nie zapisywać ich do bazy danych. Warto skorzystać z pomocy **ExceptionHandler** i dodać go np. do listy globalnych filtrów (plik `Startup.cs`).

(przykład: `OperationCancelledExceptionHandler.cs`)

2. ASP.NET Core Web API

- **Czym jest ASP.NET Core Web Api?**

Framework, który umożliwia nam proste budowanie serwisów HTTP, które mogą być używane przez wiele różnych klientów, w tym przeglądarki oraz urządzenia mobilne.

Platforma do budowania RESTowych aplikacji w .NET Framework'u.

2. ASP.NET Core Web API

- **[Route(„dummyapi/[controller]”)]** - atrybut routingu definiujący adres url dla Web Api:
„<http://localhost/dummyapi/values>”,
- **Get()** - prosta metoda **HttpGet** zwracająca wszystkie wyniki,
- **Get(int id)** – metoda **HttpGet** z użyciem template'a, który określa parametr z jakim ją wywołamy.

```
[Route("dummyapi/[controller]")]
public class ValuesController : Controller
{
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new[] { "Hello", "World" };
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {
        //find by id...
        return new ObjectResult("Hello");
    }

    [HttpPut]
    public IActionResult Put(string value)
    {
        if (string.IsNullOrEmpty(value))
        {
            return new BadRequestResult();
        }

        return Ok();
    }
}
```

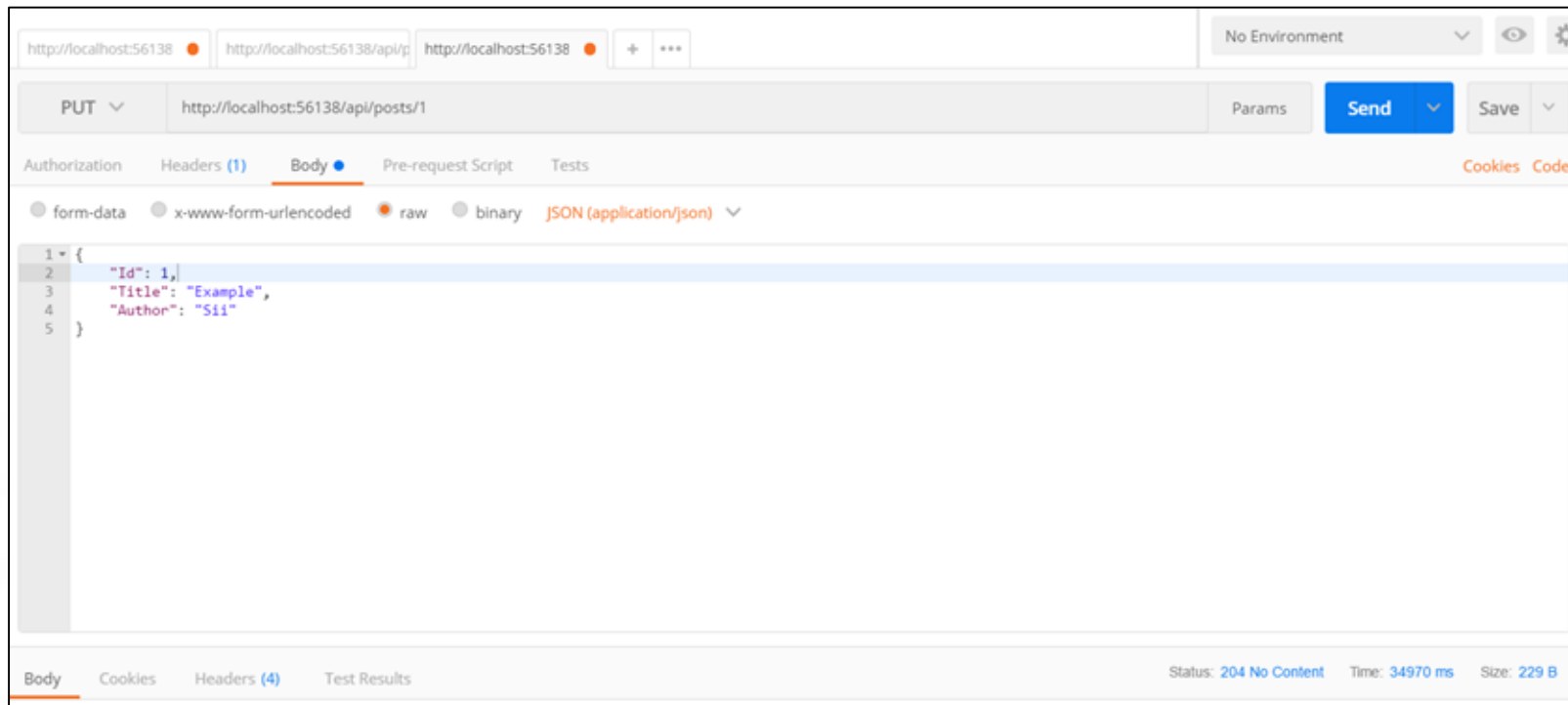

2. ASP.NET Core Web API

- Rodzaje ActionResult zwracane w WebApi

BadRequestResult	return BadRequest();	empty 400 response
BadRequestObjectResult	return BadRequest(modelState);	returns 400 with an object containing error detail as object or as Model State Dictionary
NotFoundResult	return NotFound();	returns and empty 404 response
NotFoundObjectResult	return NotFound(new { Id = 2, error = "xxx" });	returns 404 with an object containing pertinent info
ObjectResult	return new ObjectResult(new { Name = "TomDickHarry" });	response with an object but a null status code
OkObjectResult	return new OkObjectResult(new { Name = "TomDickHarry" });	200 with an object if formatting succed
OkResult	return Ok();	empty 200 without object and formatting
StatusCodeResult	return StatusCode(404);	returns a response with specified status code
	return Created(new Uri("/Home/Index", UriKind.Relative), new { Name = "Hamid" });	return 201 created status code along with the path of the created resource and the actual object

2. ASP.NET Core Web API

Testowanie kontraktów stworzonego WebApi - **POSTMAN**
(<https://www.getpostman.com/>)



2. ASP.NET Core Web API

Formatowanie wyniku odpowiedzi jako XML/JSON w zależności od adresu URL requestu

- **[FormatFilterAttribute]** - do obsługi formatu response'a na podstawie adresu URL requestu.
- **Metoda AddXmlSerializerFormatters()** - dzięki temu można serializować obiekty do i z XML'a.
- **XmlSerializerOutputFormatter()** - jeśli tylko chcemy mieć możliwość zwracania wyniku w formie XML'a.

Przykład:

AsyncSampleController.cs ([HttpGet("list.{format}"), FormatFilter])

2. ASP.NET Core Web API

Aby aplikacja mogła zwrócić odpowiedni format danych na podstawie suffixu (np. ".xml"), musimy odpowiednio skonfigurować `FormatFilter`, aby mapował dany suffix do wybranego MIME type'a.

```
services.AddMvc(options =>
{
    options.FormatterMappings.SetMediaTypeMappingForFormat
        ("xml", "application/xml");
    options.FormatterMappings.SetMediaTypeMappingForFormat
        ("js", "application/json");
})
.AddXmlSerializerFormatters();
```

2. ASP.NET Core Web API

ApiExplorer – czym jest?

- Zawiera funkcjonalność umożliwiającą wydobywanie i wyświetlanie metadanych na temat naszej aplikacji MVC.
- Dostarcza takich informacji jak np. dostępne kontrolery i akcje, ich adresy URL, dostępne metody HTTP, rodzaje parametrów i odpowiedzi.

Po co nam ApiExplorer?

Możemy generować automatyczną dokumentację lub strony z pomocą dla naszej aplikacji. Narzędzia takie jak np. **Swagger** używają mechanizmów ApiExplorer do dostarczania dokumentacji aplikacji.

2. ASP.NET Core Web API

ApiExplorer – jak zacząć?

ApiExplorer jest częścią pakietu Microsoft.AspNetCore.Mvc.ApiExplorer. Ten pakiet jest automatycznie dołączony jako referencja gdy nasza aplikacja korzysta z pakietu Microsoft.AspNetCore.Mvc.

W przeciwnym wypadku wystarczy w metodzie ConfigureServices dodać `services.AddApiExplorer()`

```
public class DocumentationController : Controller
{
    private readonly IApiDescriptionGroupCollectionProvider _apiExplorer;
    public DocumentationController(IApiDescriptionGroupCollectionProvider apiExplorer)
    {
        _apiExplorer = apiExplorer;
    }

    public IActionResult Index()
    {
        return View(_apiExplorer);
    }
}
```

2. ASP.NET Core Web API

Entity Framework In-Memory Data Provider

Aplikacje ASP.NET Core mogą być testowane z różnymi frameworkami. Entity Framework Core sprawia, że testowanie aplikacji wykorzystujących EF jest niezwykle proste dzięki wykorzystaniu providera 'In-Memory'.

- Testowanie kontrolerów opartych o pobieranie danych przy użyciu EF
- Tworzenie encji, kontekstu bazy danych jest standardowe, natomiast jakiej bazy użyjemy decydujemy na poziomie klasy Startup,
- Aby uniknąć mock'owanie i fake'owania, EF Core dostarcza nam "In-Memory" provider'a, który nie opiera się na bazie danych i jego głównym celem jest wspieranie scenariuszy testowych gdzie realny data-storage nie jest potrzebny, czyli np. Unit testy.

```
// This method gets called by the runtime. Use this method to add services to the container.  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationContext>(options =>  
        options.UseInMemoryDatabase("BlogDb"));  
  
    services.AddTransient<IRepository, InMemoryRepository>();  
}
```


2. ASP.NET Core Web API

ZADANIE

- a) Utwórz kontekst bazodanowy (Entity Framework) oraz wykorzystaj "InMemoryDatabase" (w klasie Startup) w celu zamockowania dostępu do bazy danych.
- b) Utwórz RESTful Web API, który będzie zawierał 4 różne rodzaje requestów HTTP w odniesieniu do repozytorium (GET/POST/PUT/DELETE - CRUD Operations)
- c) Sprawdź rozwiązanie za pomocą narzędzia POSTMAN
- d) Zaimplementuj nową asynchroniczną akcję w kontrolerze, która wykonuje „długie zadanie” z możliwością jej przerwania za pomocą CancellationToken oraz obsłuż wyjątek implementując odpowiedni ExceptionFilter.
- e) Sprawdź działanie rozwiązania z punktu d), uruchamiając aplikację z poziomu konsoli (dotnet...), a następnie wywołując/zatrzymując odpowiednią akcję.
- f) Napisz testy jednostkowe kontrolera, w którym zaimplementowałeś operacje CRUD i przetestuj jednostkowo każdą z metod.

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-vsc>

Część 3

TESTOWANIE APLIKACJI

ASP.NET CORE MVC

1. TESTOWANIE KONTROLERÓW

Typowe odpowiedzialności kontrolerów:

- Sprawdzanie stanu **ModelState.IsValid**,
- Zwracanie wyniku błędu jeśli ModelState jest nieprawidłowy,
- Pobieranie encji biznesowej ze źródła danych,
- Wykonywanie akcji na encji biznesowej,
- Zapis encji biznesowej do źródła danych,
- Zwracanie odpowiedniego **ActionResult**.

2. TESTY INTEGRACYJNE

Kilka słów o testach integracyjnych

Testy integracyjne sprawdzają czy różne części systemu działają poprawnie wspólnie. W przeciwieństwie do testów jednostkowych, testy integracyjne często zawierają w sobie elementy infrastrukturalne, takie jak: dostęp do bazy danych, plików, zasobów sieciowych etc..

Testowanie jak aplikacja działa przy użyciu frameworka, np. ASP.NET Core lub w połączeniu z bazą danych jest przykładem, kiedy powinniśmy pomyśleć o pisaniu testów integracyjnych.

2. TESTY INTEGRACYJNE

Testy Integracyjne w ASP.NET Core

Jak zacząć?

- stworzyć osobny projekt testowy dla testów integracyjnych,
- podpiąć do niego referencję do naszej aplikacji webowej,
- zainstalować test runnera (MSTest, Nunit, etc...).

Test Host

ASP.NET Core zawiera test host'a, którego można podpiąć pod nasz projekt z testami i używać go do hostowania aplikacji ASP.NET Core, bez konieczności posiadania prawdziwego web host'a. Testowy host jest dostępny po pobraniu paczki Nugetowej: **Microsoft.AspNetCore.TestHost**

2. TESTY INTEGRACYJNE

Używanie in-memory web host'a umożliwia nam szybką i prostą konfigurację oraz uruchamianie naszych testów integracyjnych. **Kestrel** jest cross platformowym web serwerem developerskim, którego używają aplikacje napisane w .NET core.

Kestrel is a cross-platform web server for ASP.NET Core based on libuv, a cross-platform asynchronous I/O library. Kestrel is the web server that is included by default in ASP.NET Core project templates. Kestrel supports the following features:

- HTTPS
- Opaque upgrade used to enable WebSockets
- Unix sockets for high performance behind Nginx

Kestrel is supported on all platforms and versions that .NET Core supports.

2. TESTY INTEGRACYJNE

Konfiguracja serwera i klienta

```
public class TestContext
{
    private TestServer _server;
    public HttpClient Client { get; private set; }

    public TestContext()
    {
        SetUpClient();
    }

    private void SetUpClient()
    {
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());

        Client = _server.CreateClient();
    }
}
```


2. TESTY INTEGRACYJNE

Konfiguracja TestServer'a

Klasę TestServer konfigurujemy przekazując IWebHostBuilder do jego konstruktora.

Jest kilka sposobów na konfigurację IWebHostBuilder, między innymi:

- bezpośrednia konfiguracja w kodzie,
- korzystanie z plików Startup

```
var policy = new SecurityHeadersPolicyBuilder()
    .AddDefaultSecurePolicy();

var hostBuilder = new WebHostBuilder()
    .ConfigureServices(services => services.AddSecurityHeaders())
    .Configure(app =>
    {
        app.UseSecurityHeadersMiddleware(policy);
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Test response");
        })
    });
```

```
public class TestFixture<TStartup> : IDisposable where TStartup : class
{
    private readonly TestServer _server;

    public TestFixture()
    {
        var builder = new WebHostBuilder().UseStartup<TStartup>();
        _server = new TestServer(builder);

        Client = _server.CreateClient();
        Client.BaseAddress = new Uri("http://localhost:5000");
    }

    public HttpClient Client { get; }

    public void Dispose()
    {
        Client.Dispose();
        _server.Dispose();
    }
}
```

2. TESTY INTEGRACYJNE

Używanie WebHostBuilder:

- testowanie naszej aplikacji oraz middleware'ów w różnych scenariuszach,

Używanie klasy Startup

- testowanie systemu, który będzie taki sam jak produkcyjny

2. TESTY INTEGRACYJNE

ĆWICZENIE

- A) Skonfiguruj WebHostBuilder'a w ten sposób, aby przy uruchamianiu testów tworzył instancję serwera przy użyciu klasy Startup z projektu aplikacji webowej.
- B) Uaktualnij csproj testów integracyjnych i dodaj do niego wymagany Target "CopyDepsFiles"
- C) Napisz smoke testy dla 2 wybranych endpointów w WebApi oraz dla "zwykłej" akcji MVC
- D) Zaimplementuj testy integracyjne, które sprawdzą poprawność działania globalizacji aplikacji, w tym celu stwórz 2 akcje kontrolera które będą zwracać określony (zlokalizowane) tekst.
Test powinien sprawdzić ustawianie kultury na min 2 sposoby (np. Poprzez query string oraz nagłówek "Accept-Language")



Dziękuję za uwagę :)

BIBLIOGRAFIA

PRO ASP.NET Core MVC – Adam Freeman (wyd. APRESS)

<https://www.linkedin.com/pulse/aspnet-core-mvc-pipeline-lucas-araujo/>

<https://kimsereyblog.blogspot.com/2017/04/filters-in-asp-net-core-what-are-they.html>

<http://www.dotnetcurry.com/aspnet-mvc/1368/aspnet-core-mvc-custom-model-binding>

<https://tahirnaushad.com/2017/08/22/asp-net-core-2-0-mvc-model-binding/>

<https://dotnetcoretutorials.com/2016/12/28/custom-model-binders-asp-net-core/>

<https://blog.mariusschulz.com/2015/12/14/tag-helpers-in-asp-net-mvc-6>

<https://codewala.net/2017/03/09/exploring-asp-net-core-view-component/>

<https://docs.microsoft.com/en-US/aspnet/core/index>