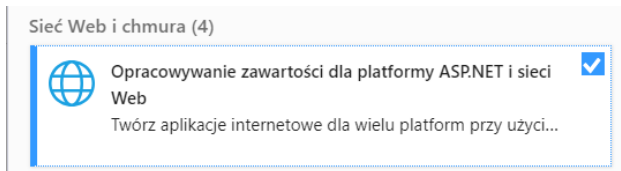


Tutorial – Instalacja i konfiguracja

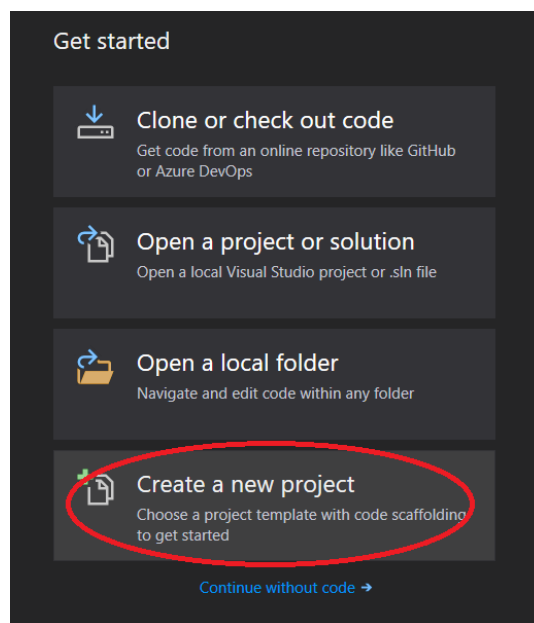
Instalacja Visual Studio 2019.

Jeśli nie korzystałeś z programów Microsoft w ramach swojego konta studenckiego podążaj za tą instrukcją. Po zrobieniu tych kroków pobierz Visual Studio Enterprise 2019 (ew Visual Studio for Mac) <https://instrukcje.put.poznan.pl/tworzenie-konta-w-portalu-microsoft-azure/>

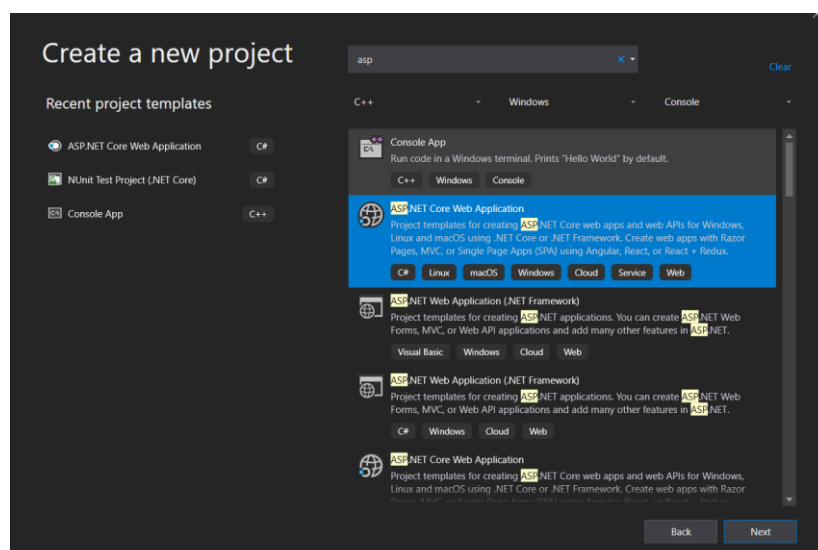
Podczas instalacji zainstaluj następujący pakiet:



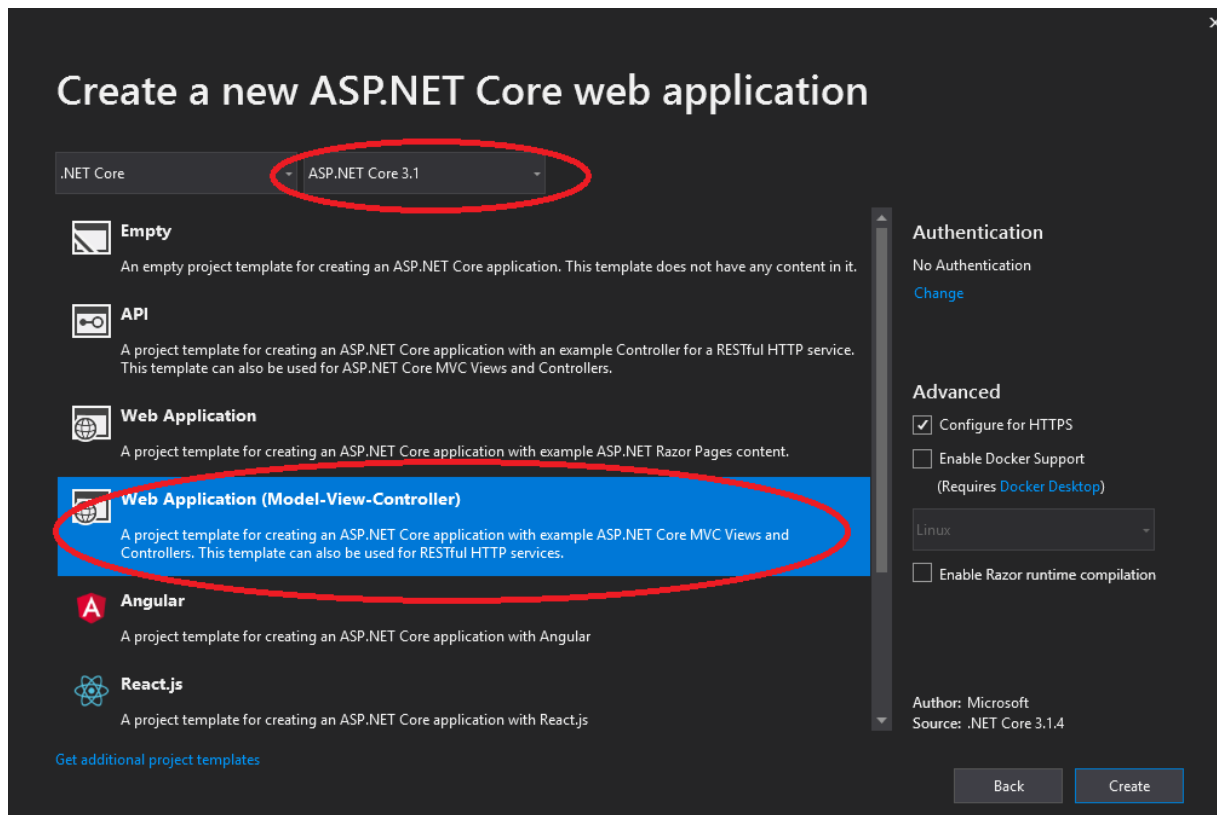
Po instalacji, aby stworzyć projekt należy kliknąć:



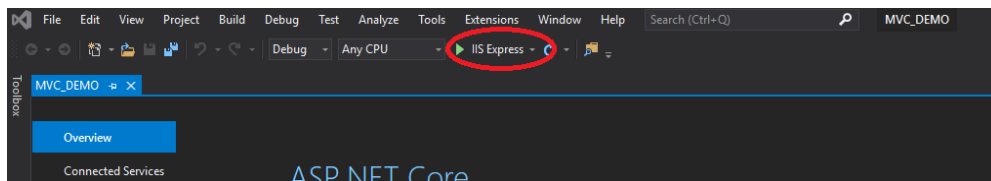
Wybrać ASP.NET Core Web Application:



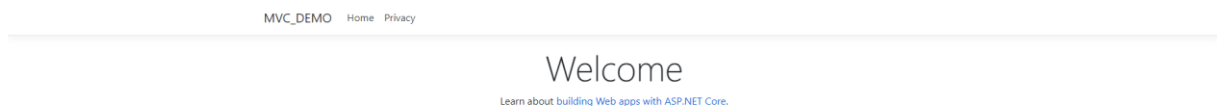
Nadać tytuł projektu i kliknąć Create, następnie wybrać Web Application (Model-View-Controller) i ASP.NET Core 3.1. Jeśli nie byłoby na liście ASP.NET Core 3.1 (lub 3.0) to można pobrać go z <https://dotnet.microsoft.com/download>: (pobieramy SDK i Runtime)



Po stworzeniu klikamy:

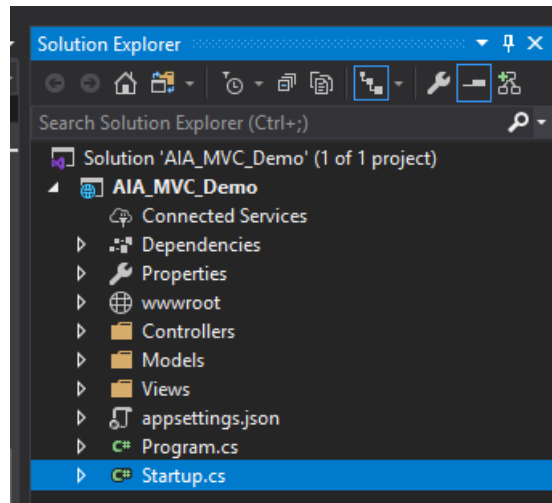


Naszym oczom powinien ukazać się taki widok:



Tutorial – Tworzenie aplikacji

Opis hierarchii:



Program.cs: Tutaj wszystko się zaczyna w funkcji Main. Można tutaj definiować różne ustawienia np. dotyczące logowania w aplikacji. Ważnym miejscem jest linia `webBuilder.UseStartup<Startup>()`; gdzie wskazujemy klasę Startup gdzie jest zdefiniowane większość ustawień aplikacji.

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

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

Startup.cs: W tej klasie mamy dwie główne metody `ConfigureServices` i `Configure`.

`ConfigureServices` – służy głównie do dodawania nowych serwisów. Jest tam wbudowane `Dependency Injection`. (o tym będzie później)

`Configure` - tutaj możemy zdefiniować takie rzeczy jak podstawowy routing jaki ma być w naszej aplikacji, czy ma być używana Autentykacja lub jakieś middleware (rodzaj funkcji przez którą musi przejść request przed dotarciem do controllera)

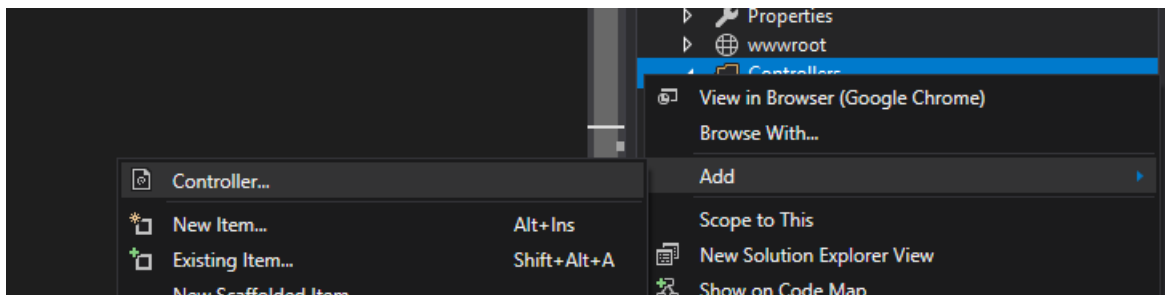
_View imports – tutaj definiujemy główne importy klas z C# do naszych widoków

_View starts - tutaj definiujemy nasz główny startowy Layout

_Layout – Widok początkowy w nim możemy umieścić różne importy bibliotek js. Umieścić stopkę czy też pasek do nawigacji (navbar)

Wszystkie widoki będą renderowane w miejscu gdzie jest div z klasą container, w środku jest funkcja @RenderBody(). Tam pojawiają się wszystkie zdefiniowane później widoki

Dodajmy teraz controller z którego będziemy później korzystać (Klikamy prawym na folder Controller -> Add -> Controller wybieramy MVC Controller – Empty i wprowadzamy nazwę *GithubController*):



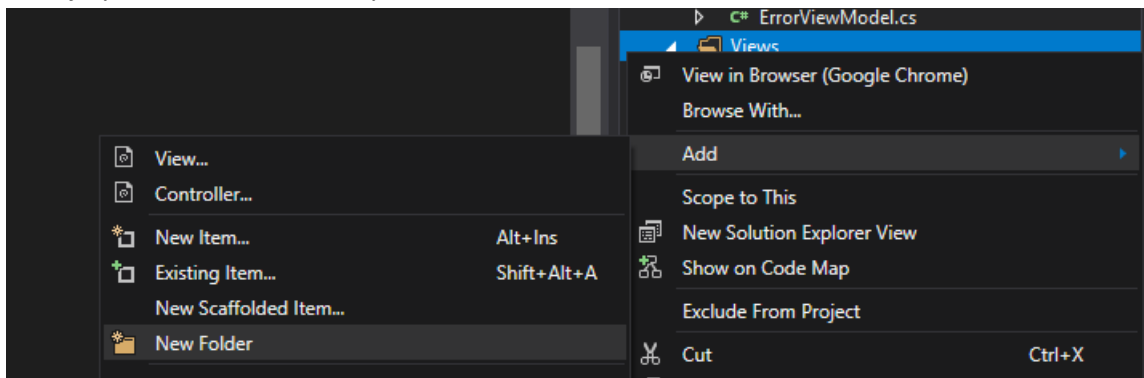
W Startup.cs zmieniamy w funkcji Configure wywołanie app.UseEndpoints:

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

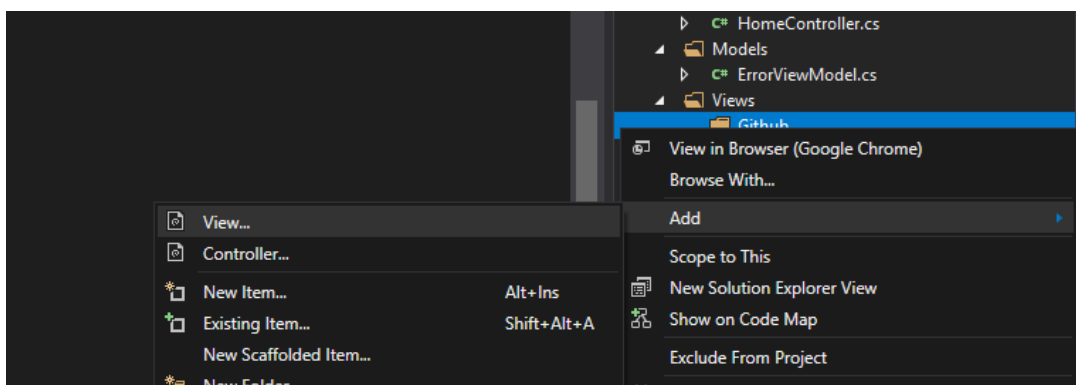
na

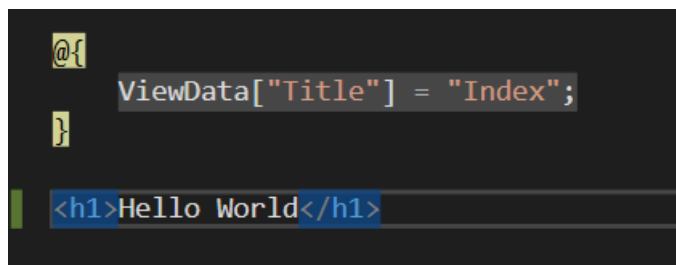
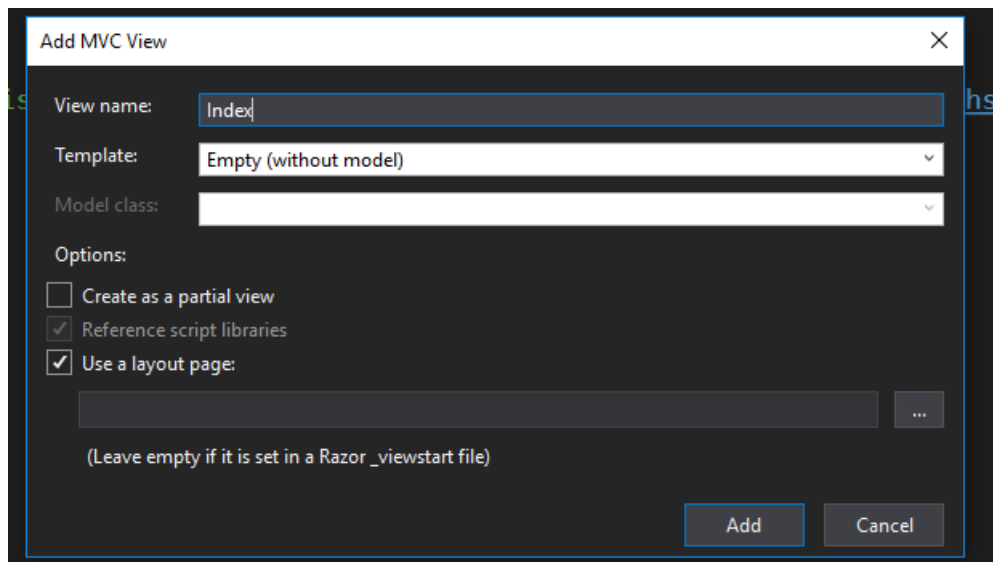
pattern: "{controller=Github}/{action=Index}/{id?}";

Dodajmy do folderu View nowy folder o nazwie Github:



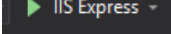
Następnie w tym folderze utwórzmy View:





```
@{
    ViewData["Title"] = "Index";
}

<h1>Hello World</h1>
```

Możemy kliknąć w IIS Express, włączy to aplikację:  w przypadku VS for mac przycisk będzie się inaczej nazywał

Powinno pojawić się coś podobnego:



Będziemy chcieli stworzyć aplikację do obsługi api githuba. W tym celu pierwszą rzeczą będzie stworzenie prostego formularza gdzie użytkownik będzie mógł wprowadzić dane odnośnie nazwy użytkownika i repozytorium github.

W tym celu dodajmy do naszego widoku index.cshtml prosty kod

```

@model AIA_MVC_Demo.Models.GithubInputModel
@{
    ViewData["Title"] = "Index";
}

<h1>Hello World</h1>

@using (Html.BeginForm("Index", "Github", FormMethod.Post))
{
    <table class="table">
        <thead>
            <tr>
                <th>
                    User Name: <input type="text" name="UserName" id="UserName" />
                </th>
                <th>
                    Repository Name: <input type="text" name="RepositoryName"
id="RepositoryName" />
                </th>
                <th>
                    <input type="submit" value="Submit" />
                </th>
            </tr>
        </thead>
    </table>
}

@if (Model != null)
{
    <table class="table">
        <thead>
            <tr>
                <th>
                    Model
                </th>
                <th>
                    Wartość
                </th>
            </tr>
            <tr>
                <td>@Html.LabelFor(x => x.UserName)</td>
                <td>@Model.UserName</td>
            </tr>
            <tr>
                <td>@Html.LabelFor(x => x.RepositoryName)</td>
                <td>@Model.RepositoryName</td>
            </tr>
        </thead>
    </table>
}

```

Formularz będzie po kliknięciu przycisku submit wysyłał Post request do controllera **Github** co zostało sprecyzowane w przekazywanych do `Html.BeginForm` parametrach.

Przed obsłużeniem tego zapytania dodajmy do folderu Models klasę **GithubInputModel** i dodajmy tam następujące właściwości:

```
1 reference | 0 changes | 0 authors, 0 changes
public class GithubInputModel
{
    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string UserName { get; set; }
    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string RepositoryName { get; set; }
}
```

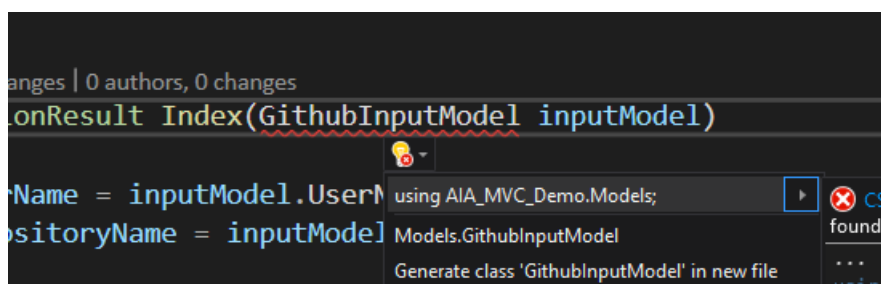
Ważne, aby nazwy pól były takie same jak wartości atrybutów **name** w widoku.

Teraz do Github controllera możemy dodać metodę:

```
[HttpPost]
public IActionResult Index(GithubInputModel inputModel)
{
    var userName = inputModel.UserName;
    var repositoryName = inputModel.RepositoryName;

    return View(new GithubInputModel { UserName = userName, RepositoryName = repositoryName });
}
```

Jeśli kompilator zaznacza nam na czerwono klasy, możemy kliknąć alt + enter aby dodać odpowiednie referencje



Po włączeniu aplikacji powinny wyświetlać się takie widoki:

AIA_MVC_Demo Home Privacy

Hello World

User Name: Repository Name:

Po wpisaniu wartości i kliknięciu przycisku:

Hello World

User Name: Repository Name:

Model	Wartość
UserName	user
RepositoryName	repos

Gdybyśmy chcieli zmienić jakieś pliki serwerowe jak style i pliki js, wszystkie znajdują się w folderze wwwroot.

Obecnie wyświetlamy to co użytkownik wprowadził, a dążymy do tego aby wyświetlać prawdziwe dane z platformy github.

Stwórzmy model odpowiedzi: GithubResultModel:

```
public class GithubResultModel
{
    [JsonProperty("full_name")]
    public string FullName { get; set; }
    [JsonProperty("description")]
    public string Description { get; set; }
    [JsonProperty("clone_url")]
    public string CloneUrl { get; set; }
    [JsonProperty("language")]
    public string Language { get; set; }
    [JsonProperty("open_issues_count")]
    public string OpenIssuesCount { get; set; }
    [JsonProperty("watchers")]
    public string Watchers { get; set; }
}
```

Uaktualnijmy kod w naszym controllerze:

```
var webRequest =
(HttpWebRequest)WebRequest.Create($"https://api.github.com/repos/{userName}/{repositoryName}");
webRequest.UserAgent = "userAgent";
var webResp = webRequest.GetResponse();

GithubResultModel githubResult;

using (var reader = new StreamReader(webResp.GetResponseStream()))
{
    string result = reader.ReadToEnd();
    githubResult = JsonConvert.DeserializeObject<GithubResultModel>(result);
}

return View(githubResult);
```


W widoku musimy zmienić model z jakiego korzystamy, znajduje się w pierwszej linii kodu:

```
@model AIA_MVC_Demo.Models.GithubResultModel
```

Zamiast wpisywać do tabelki wszystkie wiersze w taki sposób:

```
<tr>
<td>@Html.LabelFor(x => x.UserName)</td>
<td>@Model.UserName</td>
</tr>
```

Skorzystamy z refleksji:

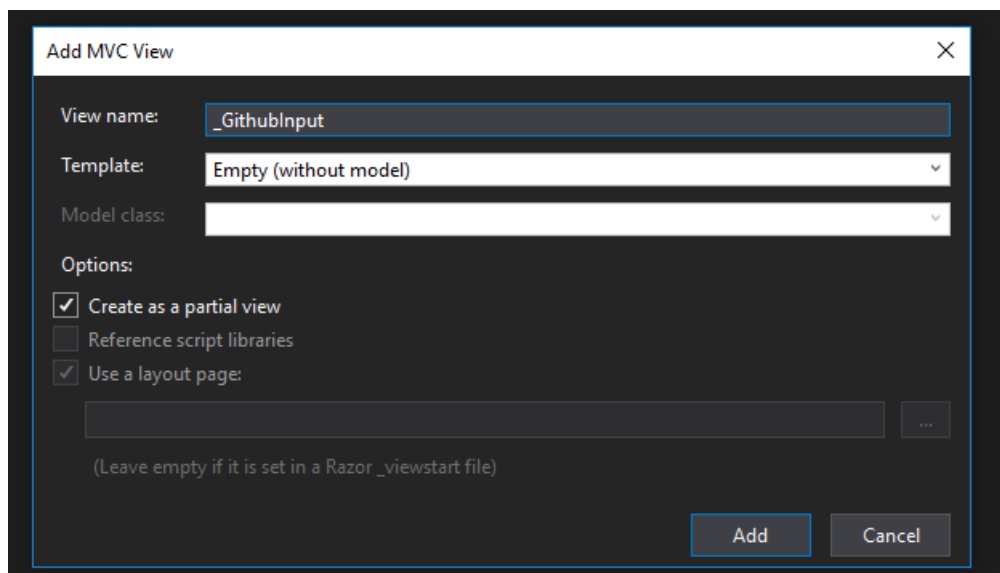
```
@foreach (var property in ViewData.ModelMetadata.Properties)
{
    <tr>
        <td>@(property.PropertyName)</td>
        <td>@Html.Display(property.PropertyName)</td>
    </tr>
}
```

Refleksja jest to mechanizm, który pozwala na dostęp do metadanych klas lub obiektów.

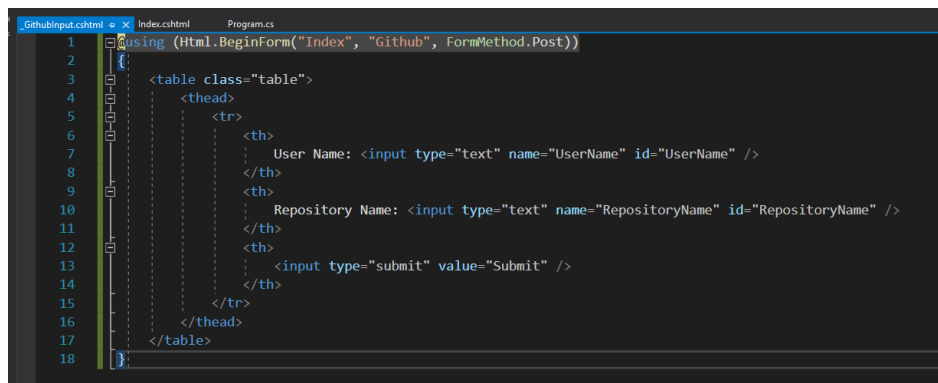
Metadane to informacje przechowywane w plikach wykonywalnych czyli .dll lub .exe opisujące pola, właściwości, metody znajdujące się w tych plikach. Do tworzenia metadanych służą atrybuty.

Mówiąc o ASP.NET Core MVC na pewno należy wspomnieć o partialach. Są to częściowe widoki które służą do dwóch rzeczy, lepszej strukturyzacji naszych widoków i możliwości wykorzystania ich później w innym miejscu.

Klikamy prawym przyciskiem na folder Github (w View) i Add -> View. Zaznaczamy Create as a partial view:



Do partiala przenosimy cały kod formularza z Index.cshtml



```
1 @using (Html.BeginForm("Index", "Github", FormMethod.Post))
2 {
3     <table class="table">
4     <thead>
5     <tr>
6     <th>
7         User Name: <input type="text" name="UserName" id="UserName" />
8     </th>
9     <th>
10        Repository Name: <input type="text" name="RepositoryName" id="RepositoryName" />
11    </th>
12    <th>
13        <input type="submit" value="Submit" />
14    </th>
15    </tr>
16    </thead>
17    </table>
18 }
```

Do Index.cshtml dodajemy linię:

```
<partial name="_GithubInput" />
```

W miejscu gdzie wcześniej był formularz:

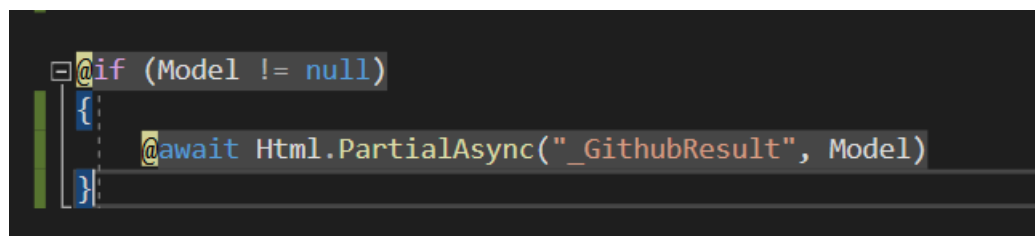


```
2 @if (Model != null)
3 {
4     ViewData["Title"] = "Index";
5 }
6 <h1>Hello World</h1>
7
8 <partial name="_GithubInput" />
9
10 @if (Model != null)
11 {
12
13
14     <table class="table">
```

Włączamy i działanie aplikacji powinno być takie samo jak poprzednio.

Stwórzmy teraz partial _GithubResult. Przenieśmy do niego kod z wnętrza IF'a.

Ostatecznie kod w Index.cshtml powinien wyglądać tak. Jeśli chcielibyśmy w PartialView korzystać z innego modelu niż w macierzystym widoku możemy go podać w taki sposób:



```
@if (Model != null)
{
    @await Html.PartialAsync("_GithubResult", Model)
}
```

W ASP.NET Core MVC jest bardzo wiele sposobów, metod a nawet bibliotek do obsługi błędów. Rozważymy prosty scenariusz, chcemy aby pokazano stronę Shared/Error.cshtml w momencie kiedy github zwróci nam kod 404. Do kodu controllera wystarczy dodać odpowiedni try catch:

```
WebResponse webResp;

    try
    {
        webResp = webRequest.GetResponse();
    }
    catch (WebException ex)
    {
        if (ex.Status == WebExceptionStatus.ProtocolError)
        {
            var response = ex.Response as HttpWebResponse;
            if (response != null)
            {
                if ((int)response.StatusCode == 404)
                {
                    return View("Error", new ErrorViewModel { RequestId = Activity.Current?.Id
?? HttpContext.TraceIdentifier });
                }
            }
        }

        throw ex;
    }
```

Oczywiście tą logikę powinno umieścić w jakimś osobnym serwisie i wywoływać ten serwis tylko z controllera. Taki serwis moglibyśmy wstrzykiwać do controllera przy pomocy wbudowanego dependency injection. Załóżmy że serwis nazywa się GithubService, a interface z jego metodami IGithubService. Moglibyśmy go wstrzyknąć w następujący sposób. Do klasy Startup.cs w metodzie ConfigureServices dodać następującą linię:

```
services.AddTransient<IGithubService, GithubService>();
```

Podana linia sprawi że za każdym wstrzyknięciem będzie tworzona nowa instancja GithubService. Używając AddScoped, byłby tworzony jeden obiekt na każdy request.

Wstrzykiwanie do samego controllera jest bardzo proste, wystarczy dodać tam konstruktor z wstrzykniętym interface'em:

```
private readonly IGithubService _githubService;

public GithubController(IGithubService githubService)
{
    _githubService = githubService;
}
```

Z takiego serwisu możemy potem korzystać w controllerze. Taki serwis umożliwi też proste pisanie testów do niego, dzięki czemu możemy bardzo dużą część kodu pokryć testami jednostkowymi i integracyjnymi

Możliwe rozszerzenia:

- Dodanie walidacji na modele
- Dodanie bazy danych + entity framework (sql lite żeby nie trzeba było pobierać nic innego), jakaś historia wyszukiwania + prosty widok
- Dodanie automappera
- Dodanie Unit testów
- Dodanie logowania (nlog) to raczej takie generyczne co w każdej aplikacji to będzie dodane ale to nic ciekawego w sumie
- Dodanie autentykacji (pewnie basic niż barer w tak prostej aplikacji ale w sumie w asp.net to praktycznie żadna różnica)