Projektowanie aplikacji ASP.NET

Wykład 02/15
Infrastruktura ASP.NET w .NET Framework

Wiktor Zychla 2024/2025

Spis treści

1 Ogólne wymagania do technologii 2
1.1 Routing i mapowanie żądań 2
1.2 Radzenie sobie z tzw. bezstanowością HTTP 3
1.3 Zestaw formantów (kontrolek) 3
1.4 Ochrona przed zagrożeniami 4
1.5 Rożne rodzaje odpowiedzi 4
2 Architektura klasycznego ASP.NET 5
2.1 Przykład handlera http 7
2.2 Przykład modułu http 8
3 Anatomia protokołu HTTP 11
4 Architektura WebForms 14

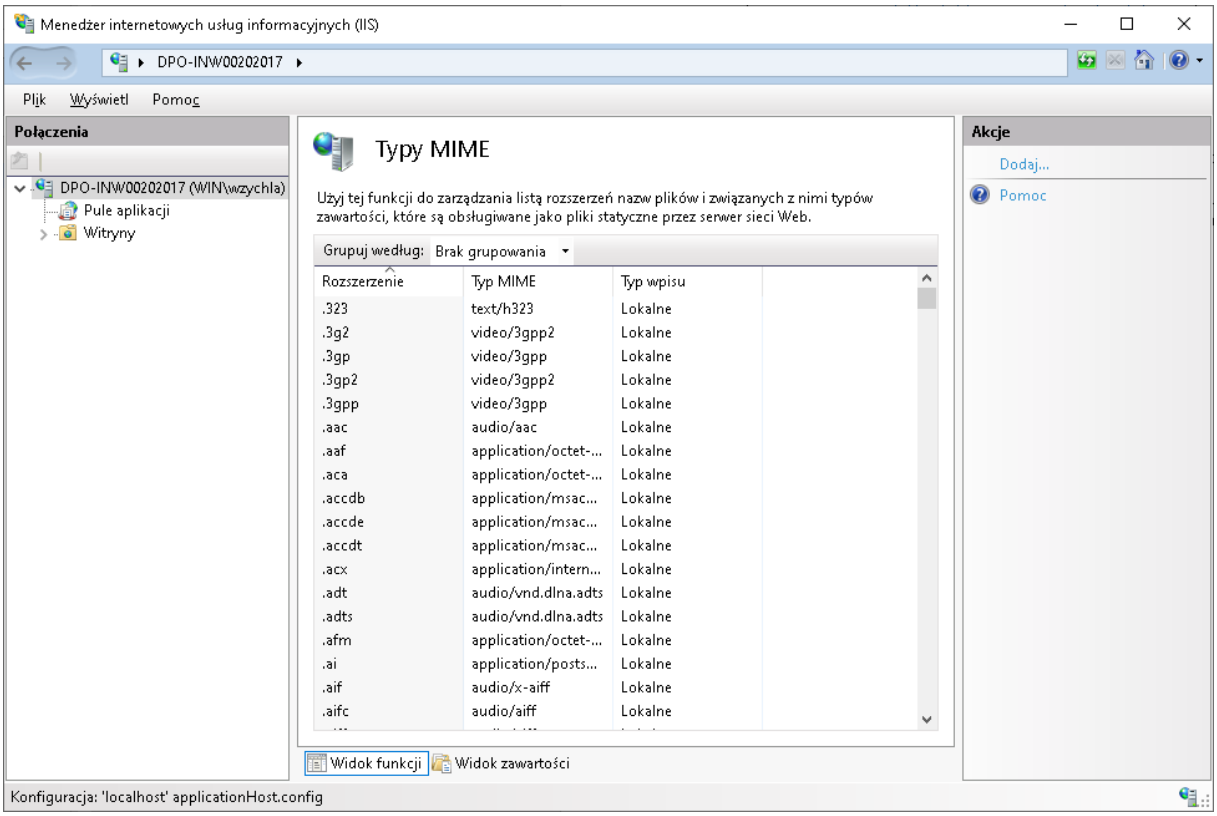
1 Ogólne wymagania do technologii

Rozpoczynając naukę dowolnej technologii tworzenia aplikacji webowych, należy zwracać uwagę na następujące elementy architektury takich aplikacji

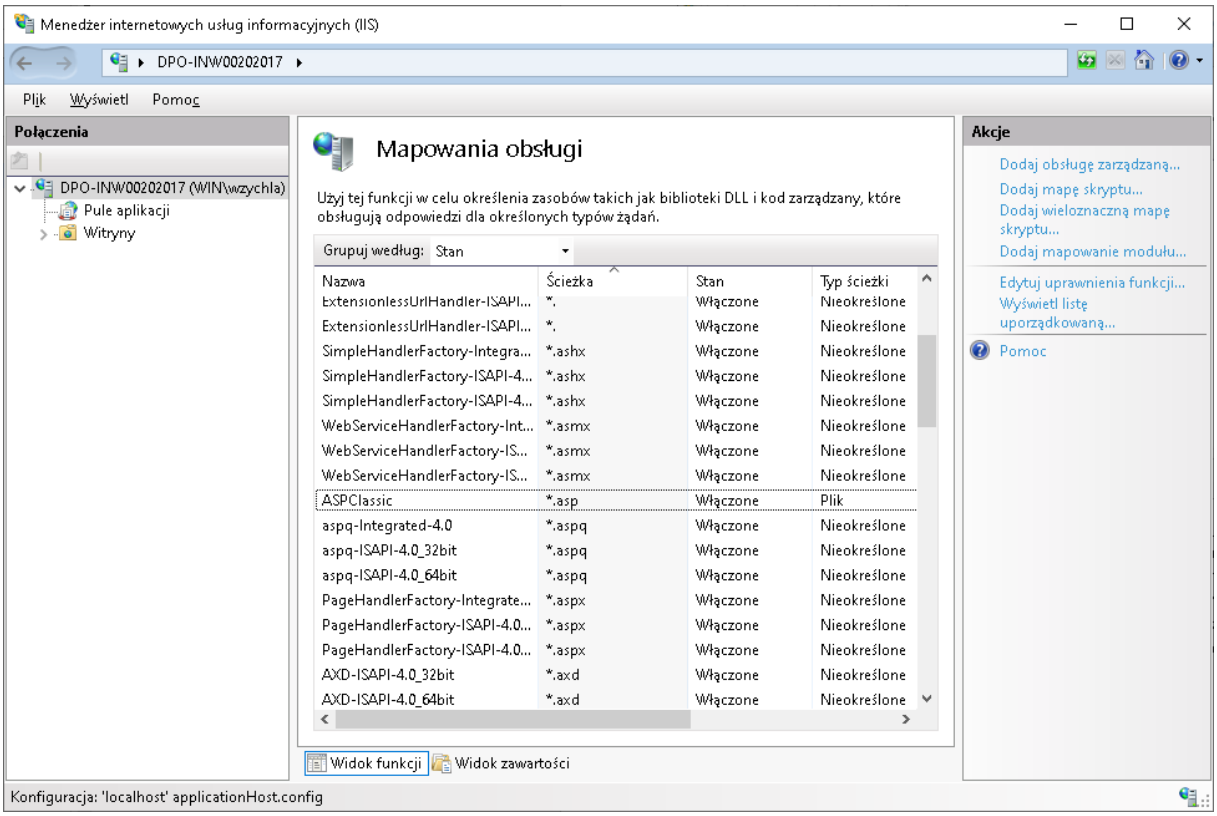
1.1 Routing i mapowanie żądań

Za tym pojęciem kryje się sposób w jaki technologia webowa przyporządkowuje przychodzące z przeglądarki adresy zasobów do tych elementów infrastruktury na serwerze, które będą odpowiedzialne za przetwarzanie żądań. Mówiąc od strony serwera - jeżeli użytkownik w pasku adresowym widzi adres http://myapp.server.com/foo/bar/qux to jaki element infrastruktury na serwerze powinien odpowiedzieć na takie żądanie?

Z punktu widzenia serwera najprostsza sytuacja to taka w której żądanie dotyczy pliku statycznego – serwer po prostu odczyta plik i wyśle jego zawartość do przeglądarki. Czy serwer ma obowiązek odsyłać wszystkie pliki jakie znajdują się w folderze aplikacji? **Nie**. W konfiguracji serwera znajduje się tzw. lista typów MIME, która określa które rozszerzenia plików statycznych są obsługiwane. Jeżeli jakiegoś rozszerzenia na liście nie ma – serwer nie wyda takiego pliku przeglądarce.



Bardziej złożona sytuacja to taka, w której plik nadal istnieje w fizycznej formie na dysku, ale określa tzw. **zasób dynamiczny**, czyli taki który wymaga dodatkowego wykonania jakiegoś kodu i dopiero odesłania zawartości. Często bywa tak, że plik jest po prostu **dynamiczną stroną webową** (czyli „mieszanką” HTML i kodu jakiegoś języka programowania), od strony statycznej taka dynamiczna strona różni się właśnie tym że dodatkowo wykonuje kod, który w jakiś sposób wpływa na odpowiedź serwera. W konfiguracji serwera znajduje się przyporządkowanie rozszerzeń zasobów dynamicznych do elementów infrastruktury, które są odpowiedzialne za przejęcie kontroli nad obsługą takich zasobów.



Najbardziej ogólna sytuacja to taka, w której adres żądania **nie odpowiada żadnemu fizycznemu plikowi** w strukturze folderów aplikacji. W takim przypadku infrastruktura posiada konfigurację przyporządkowującą przychodzącym żądaniom jakiś kod (na przykład klasę a w niej metodę) który musi się wykonać żeby obsłużyć żądanie.

1.2 Radzenie sobie z tzw. bezstanowością HTTP

Protokół HTTP jest tzw. **bezstanowy**. Oznacza to że serwer nie zapamiętuje sekwencji żądań użytkownika i obsługując żądanie numer *n* nie pamięta i nie daje dostępu do żądań *n-1, n-2*, …

To bardzo utrudnia pisanie aplikacji webowych, bo z kolei wrażenie użytkownika powinno być „ciągłe” – jeżeli do strony wpisuje jakieś dane i odsyła strony na serwer, to po otrzymaniu odpowiedzi z serwera użytkownik chce ciągłości pracy, a nie otrzymywania za każdym razem tej samej, domyślnej postaci odpowiedzi.

Przykładowo – jeżeli użytkownik loguje się, na formularzu logowania wpisuje login i hasło, serwer waliduje tę parę, ale hasło jest niepoprawne. Użytkownikowi zostanie zwrócona strona z informacją o błędzie logowania, to oczywiste. Oczywiste jest też że pole hasła zostanie wyczyszczone.

Ale pole z nazwą użytkownika powinno być już wstępnie wypełnione tą wartością, którą użytkownik dopiero co chwilę temu wpisał w to pole.

Sam protokół HTTP nie ma żadnych mechanizmów dedykowanych takiemu „podtrzymywaniu stanu”, to dopiero konkretna technologia programowania aplikacji może w jakimś zakresie wspierać programistę.

1.3 Zestaw formantów (kontrolek)

Wytwarzając aplikacje webowe szczególną uwagę zwrócimy na dostępny zestaw „kontrolek”. O ile nie ma wątpliwości że podstawowe formanty (pole tekstowe, przycisk) są przewidziane w standardzie HTML i przeglądarki na pewno je obsługują, o tyle bardziej złożone formanty (drzewka, gridy) nie są

przewidziane w standardzie. To konkretna technologia wytwarzania aplikacji dostarcza konkretnego pakietu formantów.

1.4 Ochrona przed zagrożeniami

Aplikacja webowa podlega rozlicznym zagrożeniom, wśród których wymienić warto **Query String Tampering**, **Cross-site Request Forgery** czy **Cross-site Scripting**. Technologia wytwarzania aplikacji może być tak skonstruowana, żeby zapewniać jakąś formę ochrony przed pewnymi typami zagrożeń. W trakcie naszego poznawania kolejnych podsystemów zwrócimy więc uwagę na to jak to wygląda w ASP.NET.

1.5 Rożne rodzaje odpowiedzi

Przeglądarka internetowa to tylko jeden z możliwych klientów aplikacji webowej. Równie dobrymi klientami są aplikacje mobilne czy nawet zwykłe aplikacje desktopowe. W związku z tym, odpowiedź w formacie HTML, wygodna dla przeglądarki jeśli ta ma po prostu wyrenderować stronę może nie być wygodna dla aplikacji mobilnej, która potrzebuje danych, a nie widoku. Technologia wytwarzania aplikacji powinna więc zapewniać wachlarz podsystemów wspierających obsługę rożnych architektur.

Od strony serwerowej potrzeba jest więc nie tylko wsparcia dla przeglądarek i architektury żądanie – odpowiedź HTML, ale też m.in.

- Architektury żądanie – odpowiedź JSON lub XML (usługi aplikacyjne typu REST lub WSDL)
- Komunikacja dwukierunkowa (protokół WebSockets)

2 Architektura ASP.NET dla .NET Framework

Jako technologia dynamicznego WWW, ASP.NET musi dostarczać jakiegoś sposobu organizacji przetwarzania żądań po stronie serwera.

Architektura klasycznego ASP.NET opiera się na dwóch najbardziej fundamentalnych rodzajach obiektów – to tzw. handlery i moduły.

Handlery to obiekty odpowiedzialne za wybudowanie odpowiedzi na podstawie parametrów żądania. Tylko **jeden** handler odpowie na żądanie (wytworzy odpowiedź), natomiast w jednej aplikacji może być wiele handlerów, a każdy będzie odpowiadał na pewne żądania. Na przykład – jeżeli żądanie ma adres http://myapp.server.com/..../..../WebPage.aspx a w strukturze plików na serwerze jest plik WebPage.aspx – ASP.NET obsłuży to żądanie za pomocą wbudowanego handlera stron ASP.NET WebForms. Jeżeli jednak do tej samej aplikacji trafi żądanie postaci http://myapp.server.com/foo/bar to serwer może uznać to za żądanie do podsystemu MVC, do kontrolera Foo i metody Bar w tym kontrolerze i obsłuży to żądanie handlerem podsystemu MVC.

Mieszanie rożnych podsystemów w jednej aplikacji ASP.NET jest zawsze dozwolone, bo tak naprawdę ścieżka żądania zawsze wskazuje na handler który ma je obsłużyć, a w przypadku niejednoznaczności – żądanie spróbuje obsłużyć pierwszy handler, który uzna je za „swoje”.

O tym jak uogólnić routing, czyli w pełni kontrolować przyporządkowanie ścieżek żądań do kodu który je wykonuje, porozmawiamy na jednym z kolejnych wykładów.

Moduły to obiekty implementujące dodatkowe, opcjonalne zdarzenia potoku przetwarzania. Modułów w potoku przetwarzania może być wiele.

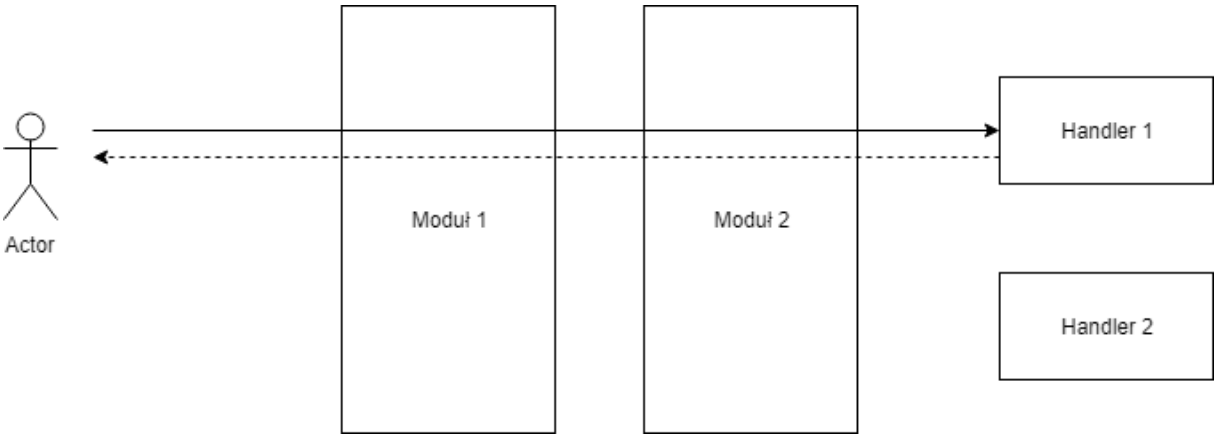
Cykl życia pojedynczego żądania to nie tylko bowiem wyprodukowanie odpowiedzi, ale również szereg innych zdarzeń, które dla wygody przetwarzania uporządkowano w powtarzalną sekwencję, w której każde pojedyncze zdarzenie może być oprogramowane własną, dodatkową logiką (jeśli jest taka potrzeba).

The following tasks are performed by the HttpApplication class while the request is being processed. The events are useful for page developers who want to run code when key request pipeline events are raised. They are also useful if you are developing a custom module and you want the module to be invoked for all requests to the pipeline. Custom modules implement the IHttpModule interface. In Integrated mode in IIS 7.0, you must register event handlers in a module's Init method.

1. Validate the request, which examines the information sent by the browser and determines whether it contains potentially malicious markup. For more information, see ValidateRequest and Script Exploits Overview.
2. Perform URL mapping, if any URLs have been configured in the UrlMappingsSection section of the Web.config file.
3. Raise the BeginRequest event.
4. Raise the AuthenticateRequest event.
5. Raise the PostAuthenticateRequest event.
6. Raise the AuthorizeRequest event.
7. Raise the PostAuthorizeRequest event.
8. Raise the ResolveRequestCache event.
9. Raise the PostResolveRequestCache event.
10. Raise the MapRequestHandler event. An appropriate handler is selected based on the file-name extension of the requested resource. The handler can be a native-code module such

as the IIS 7.0 **StaticFileModule** or a managed-code module such as the PageHandlerFactory class (which handles .aspx files).
11. Raise the PostMapRequestHandler event.
12. Raise the AcquireRequestState event.
13. Raise the PostAcquireRequestState event.
14. Raise the PreRequestHandlerExecute event.
15. Call the ProcessRequest method (or the asynchronous version IHttpAsyncHandler.BeginProcessRequest) of the appropriate IHttpHandler class for the request. For example, if the request is for a page, the current page instance handles the request.
16. Raise the PostRequestHandlerExecute event.
17. Raise the ReleaseRequestState event.
18. Raise the PostReleaseRequestState event.
19. Perform response filtering if the Filter property is defined.
20. Raise the UpdateRequestCache event.
21. Raise the PostUpdateRequestCache event.
22. Raise the LogRequest event.
23. Raise the PostLogRequest event.
24. Raise the EndRequest event.
25. Raise the PreSendRequestHeaders event.
26. Raise the PreSendRequestContent event.

Zależność między modułami a handlerami ilustruje poniższy diagram:



- Każdemu żądaniu może być przypisana **dowolna** liczba modułów, które implementować mogą różne zdarzenia potoku przetwarzania
- Każdemu żądaniu przypisany jest **jeden** handler, który odpowiedzialny jest za wyprodukowanie odpowiedzi
- Istnieją moduły i handlery wbudowane w środowisko uruchomieniowe, np.
    - Moduły do obsługi sesji po stronie serwera
    - Moduły do obsługi uwierzytelniania
    - Handler obsługi plików statycznych
    - Handler obsługi stron ASP.NET (rozszerzenie *.aspx)
    - Handler obsługi żądań MVC
    - Handler obsługi WCF
    - Itd.

Z uwagi na możliwość tworzenia własnych modułów i handlerów, istnieje możliwość rozszerzenia środowiska uruchomieniowego o obsługę dowolnej logiki.

2.1 Przykład handlera http

Najprostszy handler http mógłby wyglądać tak:

Tabela 1 Kod przykładowego handlera

```
namespace WebApplication2
{
    /// <summary>
    /// Najprostszy handler http
    /// </summary>
    public class CustomHttpHandler : IHttpHandler
    {
        public bool IsReusable
        {
            get
            {
                return false;
            }
        }

        public void ProcessRequest(HttpContext context)
        {
            context.Response.AppendHeader("Content-type", "text/html");
            context.Response.Write("handler obsługuje " + context.Request.Url);
            context.Response.End();
        }
    }
}
```

To co istotne, handler otrzymuje dostęp do elementów infrastruktury, związanych z żądaniem – tu jest to obiekt **HttpContext**, który zawiera m.in.:

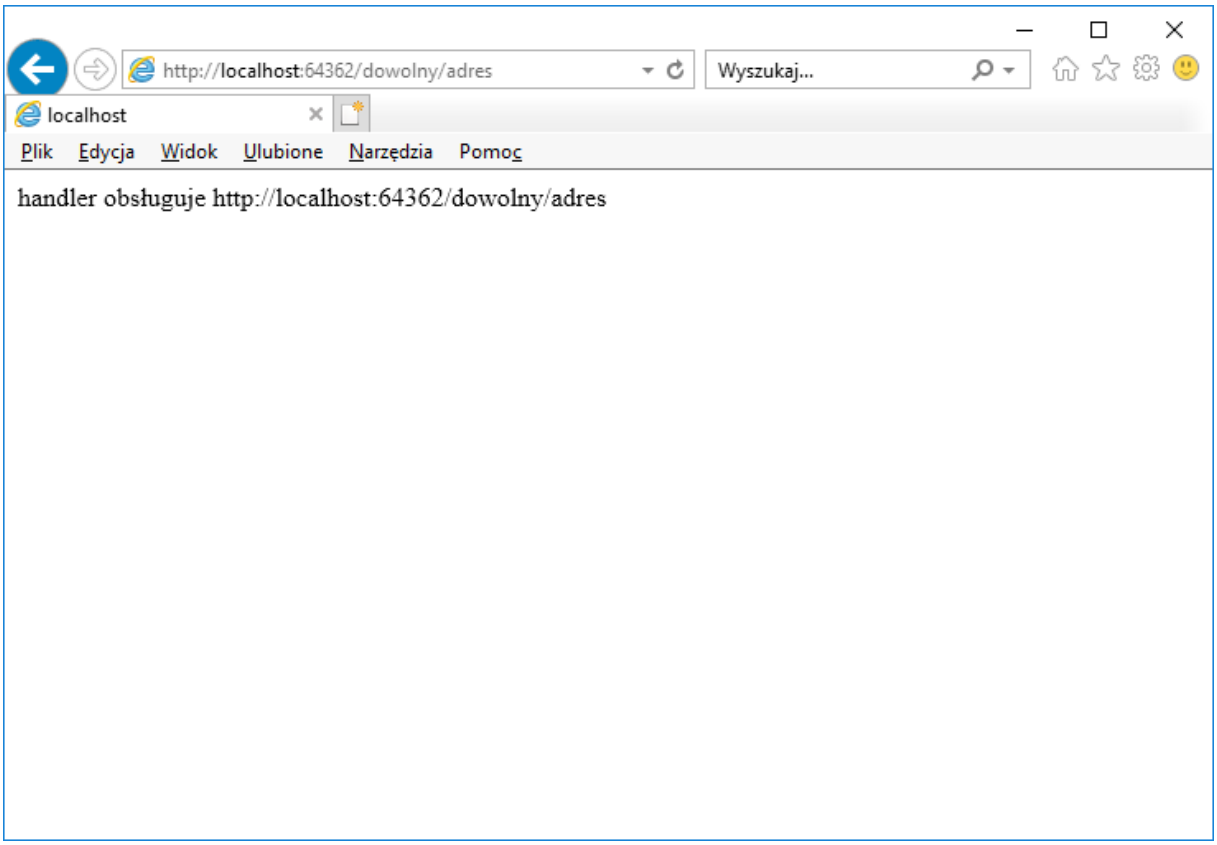
- Obiekt reprezentujący żądanie – HttpRequest
- Obiekt reprezentujący odpowiedź – HttpResponse
- Obiekt reprezentujący kontekst serwera - HttpServerUtility

Samo dodanie kodu handlera do aplikacji nie sprawi że będzie on używany – do tego potrzebna jest jego rejestracja. Możliwa jest na kilka sposobów, jednym z nich jest skonfigurowanie węzła handlerów w pliku **web.config**, który jest plikiem globalnej konfiguracji aplikacji.

Rejestracja tego konkretnego handlera wyglądałaby tak

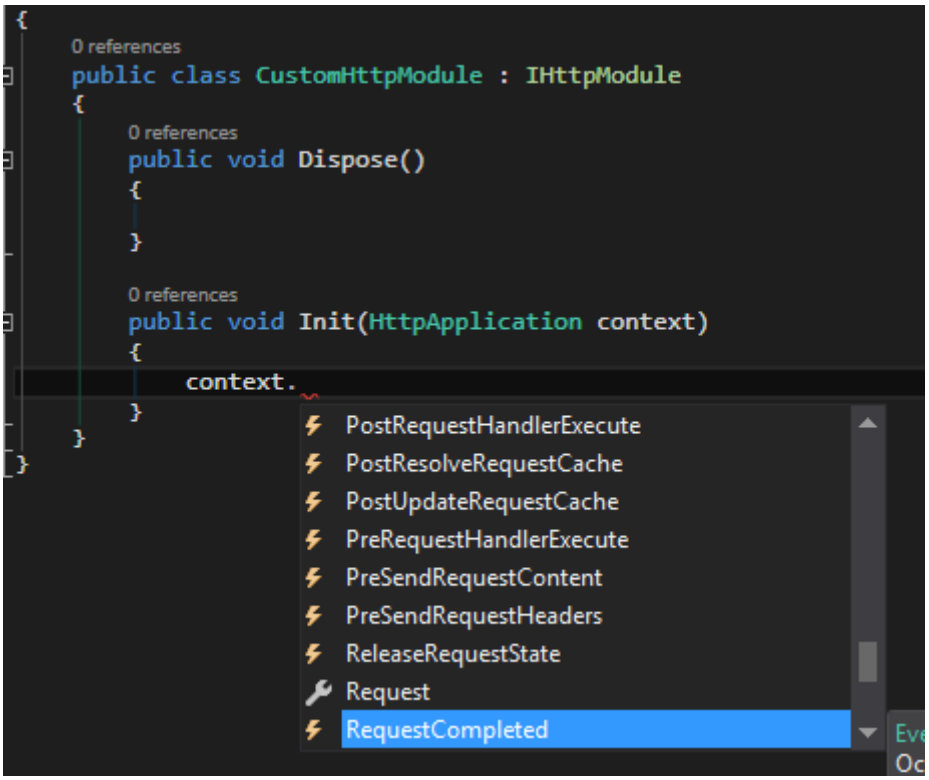
```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.6.2"/>
    <httpRuntime targetFramework="4.6.2"/>
  </system.web>
  <system.webServer>
    <handlers>
      <add name="CustomHttpHandler" type="WebApplication2.CustomHttpHandler"
           path="*" verb="*"/>
    </handlers>
  </system.webServer>
</configuration>
```

Po zarejestrowaniu, handler przejmuje kontrolę nad żądaniami, dla których środowisko nie wskaże innych handlerów:



2.2 Przykład modułu http

Jak powiedziano, moduł http pozwala na dodanie własnej logiki do jednego ze zdarzeń potoku przetwarzania:



Moduł może np. logować statystyki żądania (zdarzenia **BeginRequest**/**EndRequest**), autentykować użytkowników (zdarzenie **AuthenticateRequest**) ale nawet – zmieniać bieżący handler (zdarzenie **PreRequestHandlerExecute**):

Tabela 2 Kod przykładowego modułu

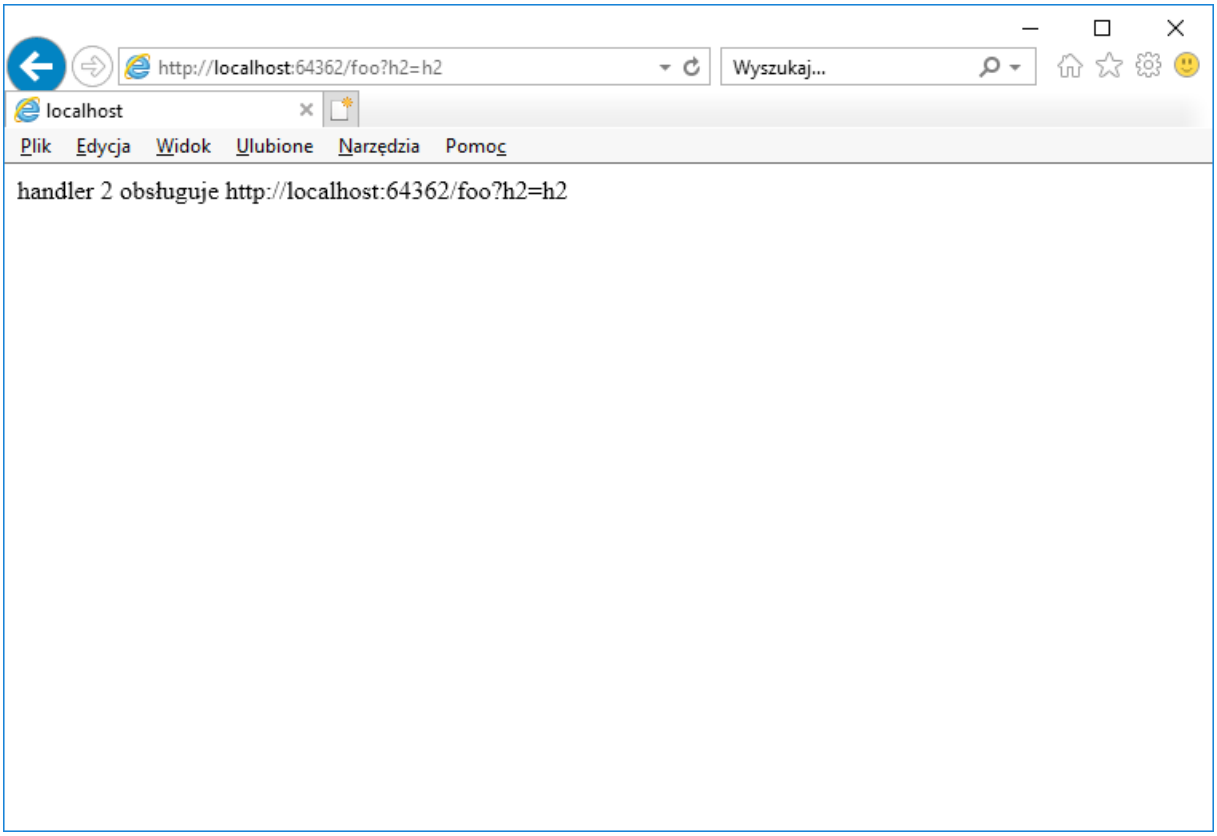
```
namespace WebApplication2
{
    public class CustomHttpModule : IHttpModule
    {
        public void Dispose()
        {

        }

        public void Init(HttpApplication context)
        {
            context.PreRequestHandlerExecute +=
                Context_PreRequestHandlerExecute;
        }

        private void Context_PreRequestHandlerExecute(
            object sender, EventArgs e)
        {
            var app = (HttpApplication)sender;
            var ctx = app.Context;

            // zamień handler na inny, jesli spełniony warunek
            if (ctx.Request.QueryString.AllKeys.Any(k => k == "h2"))
                ctx.Handler = new CustomHttpHandler2();
        }
    }
}
```

W tym przykładzie – moduł jeśli zauważy argument wywołania strony (przykładowy – „h2”), zamienia handler bieżącego żądania na inny.

Rejestracja modułu wygląda podobnie jak rejestracja handlera

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.6.2"/>
    <httpRuntime targetFramework="4.6.2"/>
  </system.web>
  <system.webServer>
    <modules>
      <add name="CustomHttpModule" type="WebApplication2.CustomHttpModule" />
    </modules>
    <handlers>
      <add name="CustomHttpHandler" type="WebApplication2.CustomHttpHandler"
           path="*" verb="*"/>
    </handlers>
  </system.webServer>
</configuration>
```

Po zarejestrowaniu moduł działa w potoku przetwarzania:



3 Anatomia protokołu HTTP

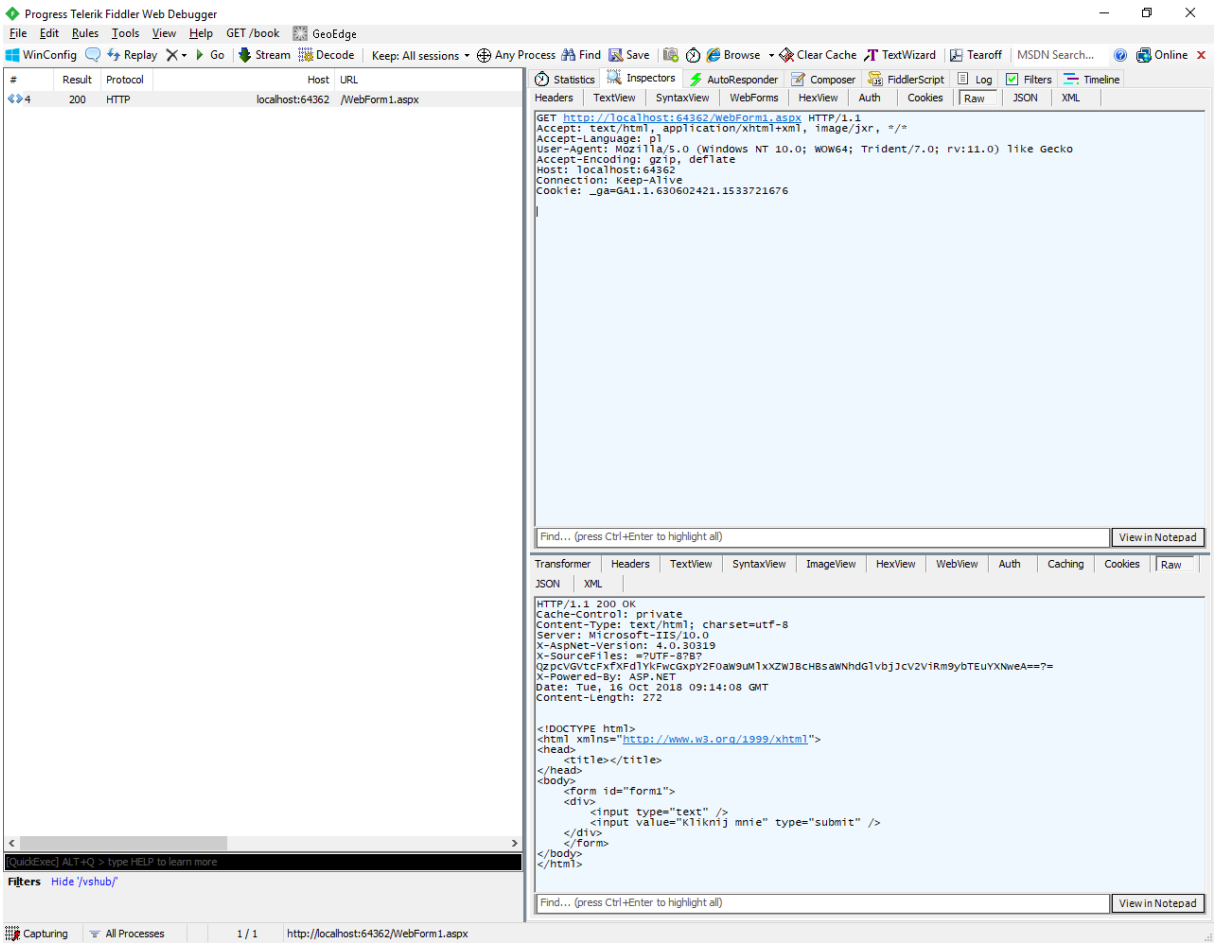
Do śledzenia protokołu HTTP można użyć oprogramowania niskopoziomowego, śledzącego cały ruch TCP/IP lub dedykowanego debuggera warstwy HTTP, który w systemie rejestruje się jako proxy. Jest kilka dobrych narzędzi śledzenia, warto polecić darmowe:

- Fiddler
- Burp

Żądania będziemy śledzić na prostej aplikacji, złożonej z jednej strony

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>
    <form id="form1" method="post">
    <div>
        <input type="text" />
        <input value="Kliknij mnie" type="submit" />
    </div>
    </form>
</body>
</html>
```

Żądanie typu **GET** przeglądarka wysyła pobierając stronę po raz pierwszy. W debuggerze po wybraniu żądania można obejrzeć zarówno żądanie jak i odpowiedź w kilku dostępnych postaciach, do szczegółowej analizy najlepiej nadaje się postać RAW w której widać jak działa protokół http – jak zbudowane jest żądanie a jak odpowiedź serwera.



Żądanie typu POST pojawia się na przykład wtedy kiedy przeglądarka odsyła formularz do serwera:

Co ważne – żądanie typu **POST** automatycznie dokłada do treści żądania pary klucz-wartość dla wszystkich formantów formularza, które mają wskazaną nazwę (atrybut **name**).

4 Architektura WebForms

W poprzednim przykładzie po odesłaniu formularza do serwera można było zaobserwować nieoczekiwane zjawisko – strona wyprodukowana przez serwer **nie zawierała** wartości wprowadzonej przez użytkownika.

Wynika to z natury protokołu http i jego naiwnego przetwarzania na serwerze – jest to tzw. bezstanowość.

Bezstanowość oznacza, że serwer:

- Nie koreluje w żaden sposób kolejnych żądań z poprzednimi żądaniami
- Nie podtrzymuje automatycznie „stanu" strony, nawet jeśli użytkownik zasila ją wartościami i odsyła zgodnie z wymogami protokołu (POST)

Rozwiązanie problemu bezstanowości możliwe jest w sposób półautomatyczny bez dodatkowego wsparcia – wystarczyłoby użyć obiektu **Request** do odczytu przysłanych wartości a następnie ręcznie dodać je do tworzonej strony, w miejscu w którym miałyby się pojawiać.

Jest to możliwe na przykład za pomocą tzw. bee stings czyli tagów formatujących zawartość dynamiczną

- `<% %>` - is for inline code (especially logic flow)
- `<%$ %>` - is for evaluating expressions (like resource variables)
- `<%@ %>` - is for Page directives, registering assemblies, importing namespaces, etc.
- `<%= %>` - is short-hand for `Response.Write` (discussed here)
- `<%# %>` - is used for data binding expressions.
- `<%: %>` - is short-hand for Response.Write(Server.HTMLEncode()) ASP.net 4.0+
- `<%#: %>` - is used for data binding expressions and is automatically HTMLEncoded.
- `<%-- --%>` - is for server-side comments

WebForms ma jednak własny pomysł na podtrzymywanie stanu strony - jest to mechanizm ViewState oraz tzw. formanty serwerowe.

Żądanie do strony WebForms obsługiwane jest przez dedykowany handler PageHandlerFactory jako takie podlega logice potoku zdarzeń opisanego wcześniej. W ramach przetwarzania strony przez handler, występuje również potok wewnętrzny który składa się z następujących elementów:

Page Event	Typical Use
PreInit	Raised after the start stage is complete and before the initialization stage begins. Use this event for the following: • Check the IsPostBack property to determine whether this is the first time the page is being processed. The IsCallback and IsCrossPagePostBack properties have also been set at this time. • Create or re-create dynamic controls. • Set a master page dynamically. • Set the Theme property dynamically.

	• Read or set profile property values. **Note** If the request is a postback, the values of the controls have not yet been restored from view state. If you set a control property at this stage, its value might be overwritten in the next event. •
Init	Raised after all controls have been initialized and any skin settings have been applied. The Init event of individual controls occurs before the Init event of the page. Use this event to read or initialize control properties.
InitComplete	Raised at the end of the page's initialization stage. Only one operation takes place between the Init and InitComplete events: tracking of view state changes is turned on. View state tracking enables controls to persist any values that are programmatically added to the ViewState collection. Until view state tracking is turned on, any values added to view state are lost across postbacks. Controls typically turn on view state tracking immediately after they raise their Init event. Use this event to make changes to view state that you want to make sure are persisted after the next postback.
PreLoad	Raised after the page loads view state for itself and all controls, and after it processes postback data that is included with the Request instance.
Load	The Page object calls the OnLoad method on the Page object, and then recursively does the same for each child control until the page and all controls are loaded. The Load event of individual controls occurs after the Load event of the page. Use the OnLoad event method to set properties in controls and to establish database connections.
Control events	Use these events to handle specific control events, such as a Button control's Click event or a TextBox control's TextChangedevent. **Note** In a postback request, if the page contains validator controls, check the IsValid property of the Page and of individual validation controls before performing any processing.

LoadComplete	Raised at the end of the event-handling stage. Use this event for tasks that require that all other controls on the page be loaded.
PreRender	Raised after the Page object has created all controls that are required in order to render the page, including child controls of composite controls. (To do this, the Page object calls EnsureChildControls for each control and for the page.) The Page object raises the PreRender event on the Page object, and then recursively does the same for each child control. The PreRender event of individual controls occurs after the PreRender event of the page. Use the event to make final changes to the contents of the page or its controls before the rendering stage begins.
PreRenderComplete	Raised after each data bound control whose DataSourceID property is set calls its DataBind method. For more information, see Data Binding Events for Data-Bound Controls later in this topic.
SaveStateComplete	Raised after view state and control state have been saved for the page and for all controls. Any changes to the page or controls at this point affect rendering, but the changes will not be retrieved on the next postback.
Render	This is not an event; instead, at this stage of processing, the Page object calls this method on each control. All ASP.NET Web server controls have a Render method that writes out the control's markup to send to the browser. If you create a custom control, you typically override this method to output the control's markup. However, if your custom control incorporates only standard ASP.NET Web server controls and no custom markup, you do not need to override the Rendermethod. For more information, see Developing Custom ASP.NET Server Controls. A user control (an .ascx file) automatically incorporates rendering, so you do not need to explicitly render the control in code.
Unload	Raised for each control and then for the page. In controls, use this event to do final cleanup for specific controls, such as closing control-specific database connections. For the page itself, use this event to do final cleanup work, such as closing open files and database connections, or finishing up logging or other request-specific tasks. **Note** During the unload stage, the page and its controls have been rendered, so you cannot make further changes to the response

	stream. If you attempt to call a method such as the **Response.Write** method, the page will throw an exception.