Projektowanie aplikacji ASP.NET

Wykład 02/15
Infrastruktura ASP.NET

Wiktor Zychla 2018/2019

Spis treści

1 Architektura ASP.NET

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

Architektura 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.

Moduły to obiekty implementujące zdarzenia potoku przetwarzania.

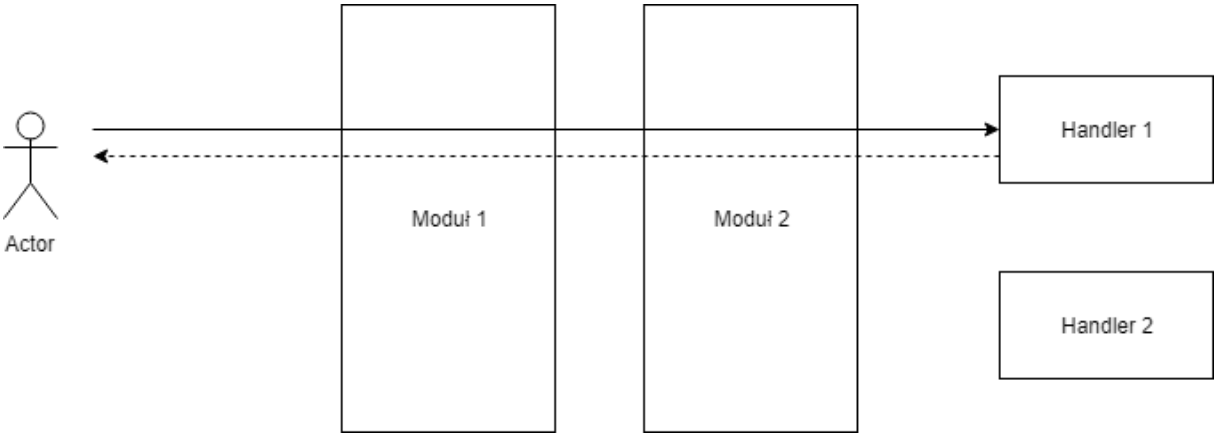
Cykl życia pojedynczego żądania to nie tylko bowiem wyprodukowanie odpowiedzi, ale również szereg innych zdarzeń, które dla wygody przetwarzania uporządkowano w powtarzalną sekwencję, w której każde pojedyncze zdarzenie może być oprogramowane własną, dodatkową logiką (jeśli jest taka potrzeba).

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.

1.1 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.:

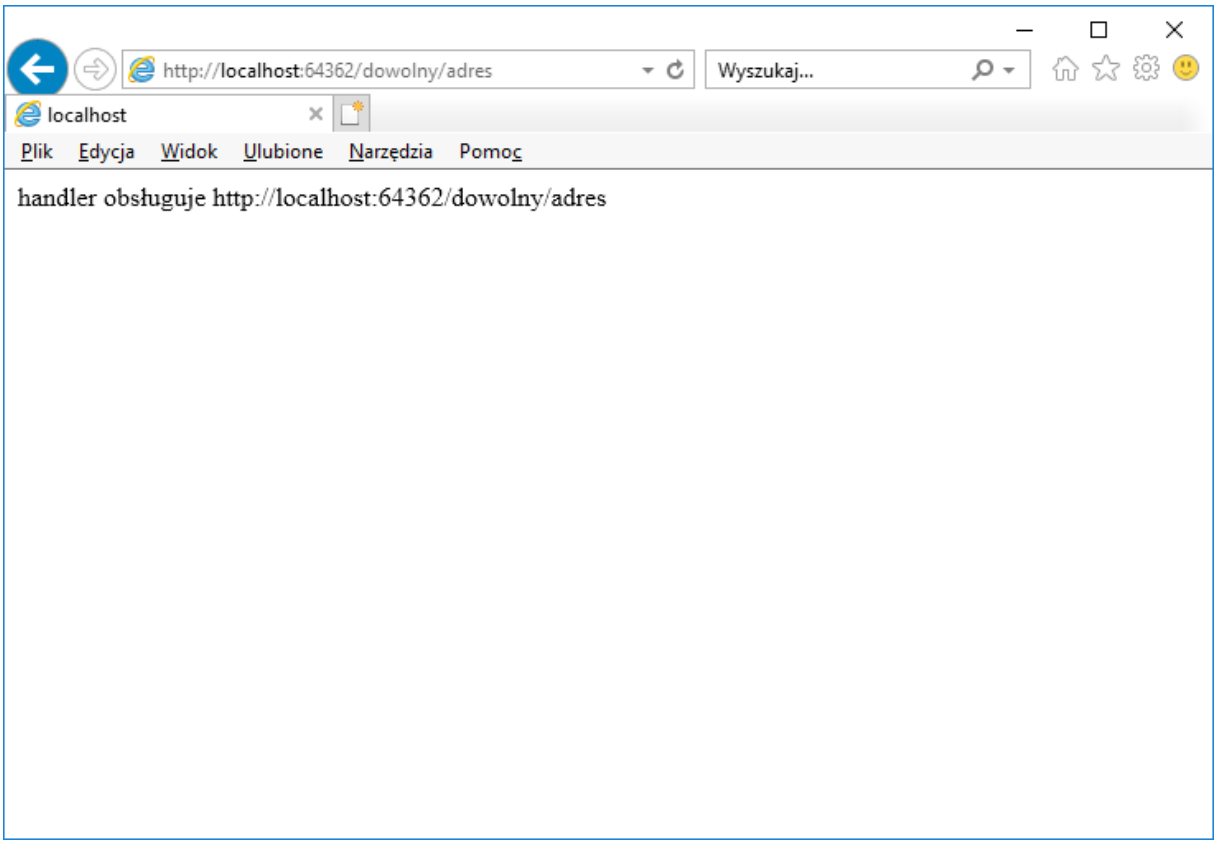
- Obiekt reprezentujący żądanie – HttpRequest
- Obiekt reprezentujący odpowiedź – HttpResponse
- Obiekt reprezentujący kontekst serwera - HttpServerUtility

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

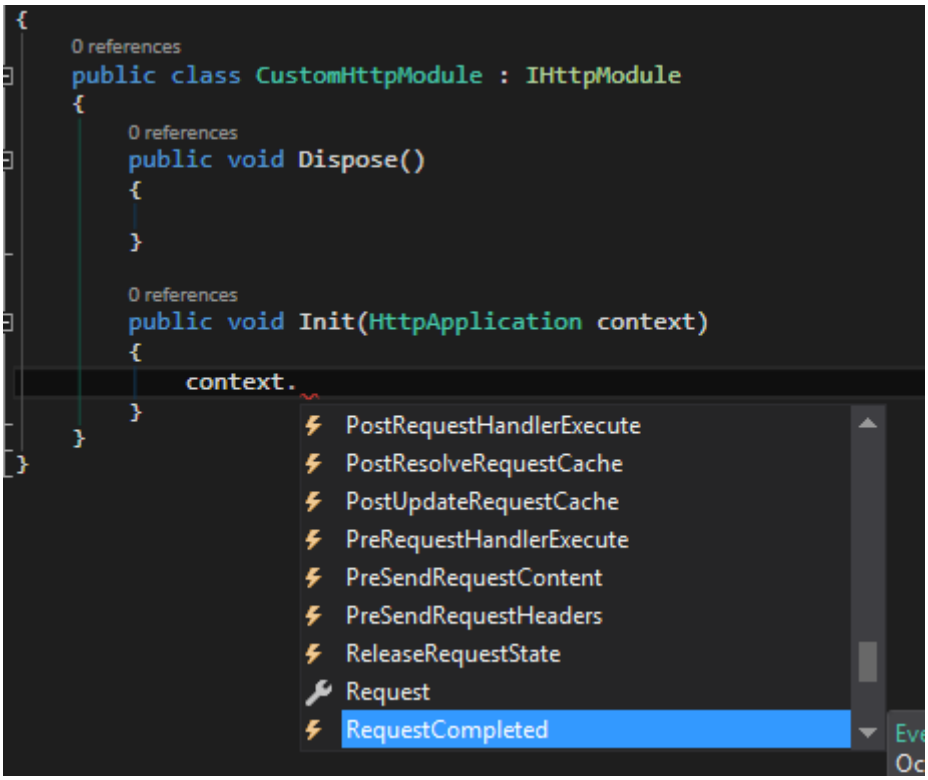
```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.6.2"/>
    <httpRuntime targetFramework="4.6.2"/>
  </system.web>
  <system.webServer>
    <handlers>
      <add name="CustomHttpHandler" type="WebApplication2.CustomHttpHandler"
           path="*" verb="*"/>
    </handlers>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

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



1.2 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

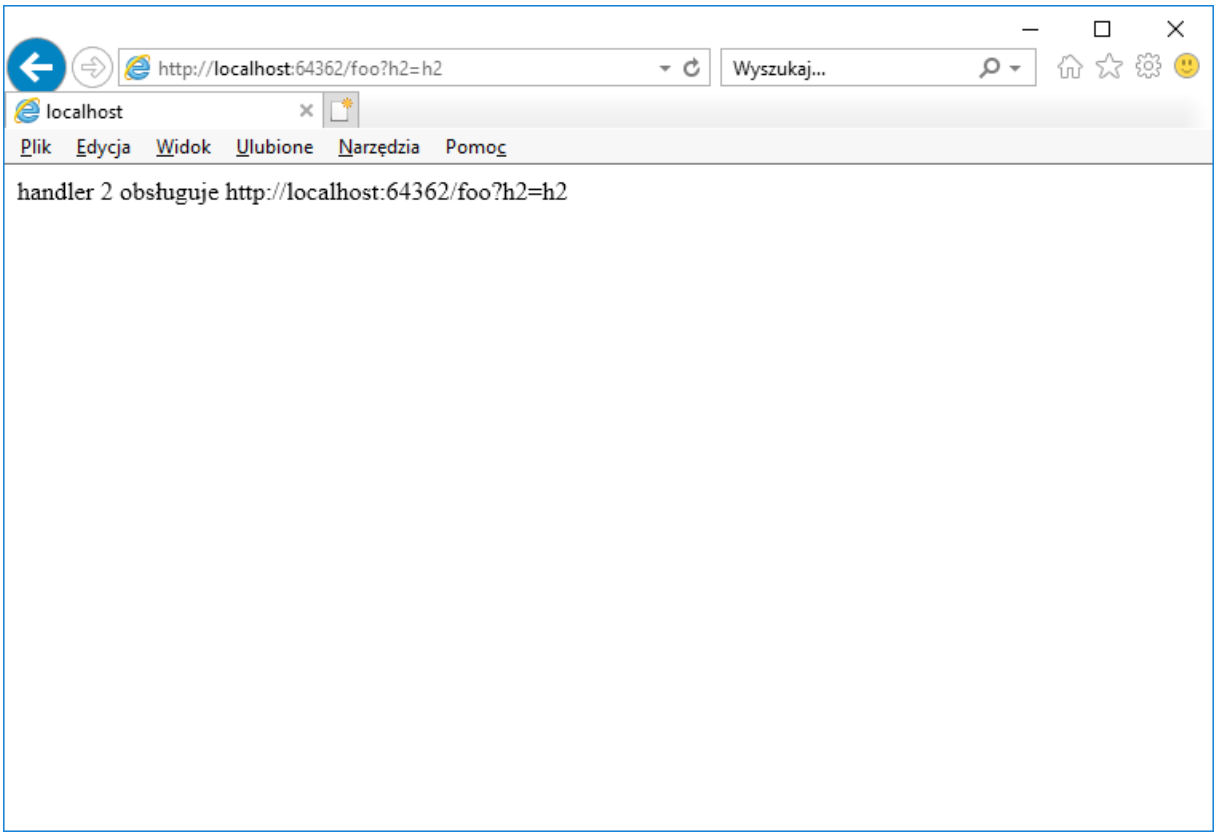
```
namespace WebApplication2
{
    public class CustomHttpModule : IHttpModule
    {
        public void Dispose()
        {

        }

        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:


http://localhost:64362/foo?h2=h2
Wyszukaj...
localhost
Plik Edycja Widok Ulubione Narzędzia Pomoc
handler 2 obsługuje http://localhost:64362/foo?h2=h2

2 Anatomia protokołu HTTP

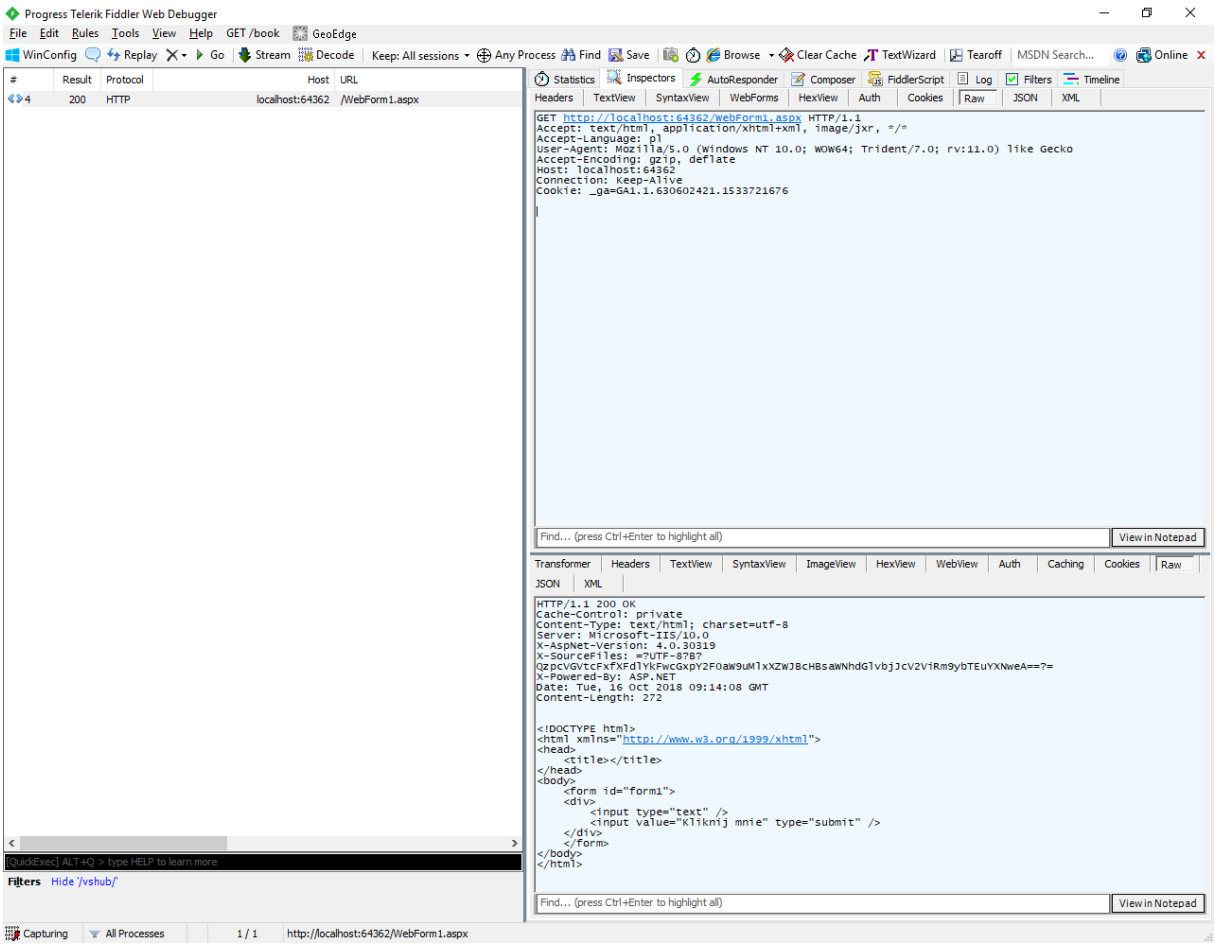
Do śledzenia protokołu HTTP można użyć oprogramowania niskopoziomowego, śledzącego cały ruch TCP/IP lub dedykowanego debuggera warstwy HTTP, który w systemie rejestruje się jako proxy. Jest kilka dobrych narzędzi śledzenia, warto polecić darmowe:

- 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**).

3 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.