Projektowanie aplikacji ASP.NET
Wykład 14/15
AJAX/CORS

Wiktor Zychla 2018/2019

Spis treści

2 Wprowadzenie

Pod zbiorczą nazwą AJAX (**A**synchronous **J**avascript **A**nd **X**ML) kryje się szereg technik bazujących na wykorzystaniu możliwości obiektu XMLHttpRequest. Współcześnie rzadko używa się AJAX w wersji która kryje się za akronimem – wynika to z faktu że odpowiedź serwera w postaci dokumentu XML nie nadaje się do łatwego przetwarzania w Javascript. Co prawda obiekt XMLHttpRequest ma wbudowany parser (alternatywnie do właściwości **responseText** można użyć **responseXML** ale kryje się pod nim niewygodne do przeglądania drzewo DOM, którego węzły można przeglądać za pomocą metod **getElementByName/getElementById** i właściwości **childNodes**.

Podstawowy przykład użycia AJAX mieliśmy okazję obejrzeć na poprzednim wykładzie. Przypomnijmy:

```
var xhr = new XMLHttpRequest();

xhr.open('POST', '/api/Index');
xhr.setRequestHeader("Content-Type", "application/json");

xhr.onreadystatechange = function () {
    if (xhr.readyState == XMLHttpRequest.DONE) {
        window.alert(xhr.responseText);
    }
};

var data = { … }

xhr.send(JSON.stringify(data));
```

Główne elementy struktury kodu:

- Zadeklarowanie parametrów połączenia – metoda **open**
- Ustawienie dodatkowych nagłówków – metoda **setRequestHeader**
- Ustawienie funkcji zwrotnej do powiadomień o zmianie stanu połączenia – callback **onreadystatechange**
- Wysłanie żądania – metoda **send**

3 AHAH

Asynchronous **H**TTP **A**nd **H**TML to jedna z wersji AJAX, tu serwer zamiast danych odsyła fragment HTML, który klient powinien włączyć do drzewa DOM bieżącej strony.

Przykładem użycia tej techniki jest formant UpdatePanel podsystemu WebForms. Zbiór formantów zamkniętych wewnątrz UpdatePanel jest aktualizowany żądaniem do serwera zwracającym fragment strony, zamiast – jak w klasycznym WebForms – całej strony. Sama strona nie jest przeładowywana przez co zatrzymuje stan w przeglądarce między poszczególnymi żądaniami – można to wykorzystać dodając np. w pełni stanowe skrypty Javascript po stronie przeglądarki.

Dodatkowo, oprócz UpdatePanel można użyć formantu **UpdateProgress**, który renderuje swój szablon wyświetlania na czas obsługi żądania przez UpdatePanel. Standardowo wykorzystuje się więc UpdateProgress do zaprezentowania w przeglądarce informacji typu „Proszę czekać”.

```
<asp:ScriptManager ID="ScriptManager1" runat="server"></asp:ScriptManager>
        <asp:UpdatePanel runat="server" ID="panel1">
            <ContentTemplate>
                <div>
                    <asp:Button ID="button1" Text="Klik" runat="server" OnClick="
button1_Click" />
                </div>
                <div>
                    <asp:Label runat="server" ID="label1" />
                </div>
            </ContentTemplate>
        </asp:UpdatePanel>
        <asp:UpdateProgress ID="UpdateProgress1" runat="server" AssociatedUpdateP
anelID="panel1">
            <ProgressTemplate>
                <div class="modal">
                    <div class="center">
                        Proszę czekać
                    </div>
                </div>
            </ProgressTemplate>
        </asp:UpdateProgress>
```

UpdatePanel do działania wymaga szeregu skryptów, które są generowanie przez umieszczony na stronie formant typu **ScriptManager**. Za ScriptManagerem stoi szerszy pomysł budowania formantów odświeżających widok przy pomocy serwera bez przeładowywania strony. O ile samą techniką nie będziemy się zajmować, warto pamiętać o gotowych bibliotekach formantów, które z niej korzystają – jest to m.in. opensource’owa biblioteka AjaxControlToolkit.

4 AJAJ

Asynchronous **J**avascript **A**nd **J**SON to najczęściej wykorzystywana w praktyce wersja AJAX. Tu serwer zwraca dane w formacie JSON, który łatwo parsuje się w przeglądarce – metoda JSON.parse zwraca gotowy obiekt Javascript, którego można łatwo użyć z poziomu kodu Javascript.

Przykład od którego rozpoczęto wykład unaocznia jednak pewną żmudność techniczną – gdyby w każdym miejscu aplikacji komunikacja z serwerem musiała wyglądać tak samo, aplikacja po stronie klienta miałaby mnóstwo zduplikowanego kodu. Z pewnością więc, po stronie Javascript w przeglądarce komunikacja z serwerem wymaga *jakiegoś* pomysłu na organizację kodu.

Jedną z ciekawych możliwości jest **automatyczne generowanie kodu** po stronie przeglądarki, na podstawie kontraktu metod udostępnianych przez serwer. W ten sposób klient – Javascript – ma zawsze aktualną wersję kodu i jeśli na serwerze zmianie ulegają nazwy/sygnatury metod, odpada konieczność aktualizowania kodu kilenta w sposób intencjonalny.

4.1 Automatyczne proxy do metod strony WebForms

Pierwszym przykładem automatycznego generowania kodu jest możliwość jaką daje WebForms. Tu – metoda w kodzie strony opatrzona atrybutem **WebMetod** generuje po stronie przeglądarki odpowiadającą metodę, dostępną z obiektu PageMethods.

Kod na serwerze:

```
public partial class WebForm3 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    [WebMethod]
    public static string DoWork( string work )
    {
        return "from server " + work;
    }
}
```

Kod w przeglądarce:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script>
        window.addEventListener('load', function () {
            document.getElementById('button1').onclick = function (e) {
                e.preventDefault();
                PageMethods.DoWork('foo', function (result) {
                    document.getElementById('content1').innerHTML = result;
                });
            }
        })
    </script>
</head>
<body>
```

```
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server" EnablePageMethods=
"true"></asp:ScriptManager>

        <div>
            <div id="content1"></div>
            <button id="button1">Klik</button>
        </div>
    </form>
</body>
</html>
```

Warto zwrócić uwagę na pewną charakterystyczną cechę automatycznie wygenerowanego kodu – jest to jego asynchroniczność. Tu oznacza to że mimo iż na serwerze metoda **DoWork** zwraca literał, w kodzie na kliencie tak nie jest – do pozyskania wartości funkcji użyta jest funkcja zwrotna (callback), który wywoływany jest asynchronicznie dopiero wtedy, kiedy do przeglądarki trafia odpowiedź z serwera.

4.2 Automatyczne proxy do metod WCF

Poprzednia technika, o ile ciekawa, jest ograniczona do WebForms. Kolejna, którą omówimy, nie ma już tego ograniczenia. Chodzi o możliwość automatycznego generowania proxy Javascript do usługi WCF.

Aby skorzystać z tej możliwości potrzena jest usługa WCF:

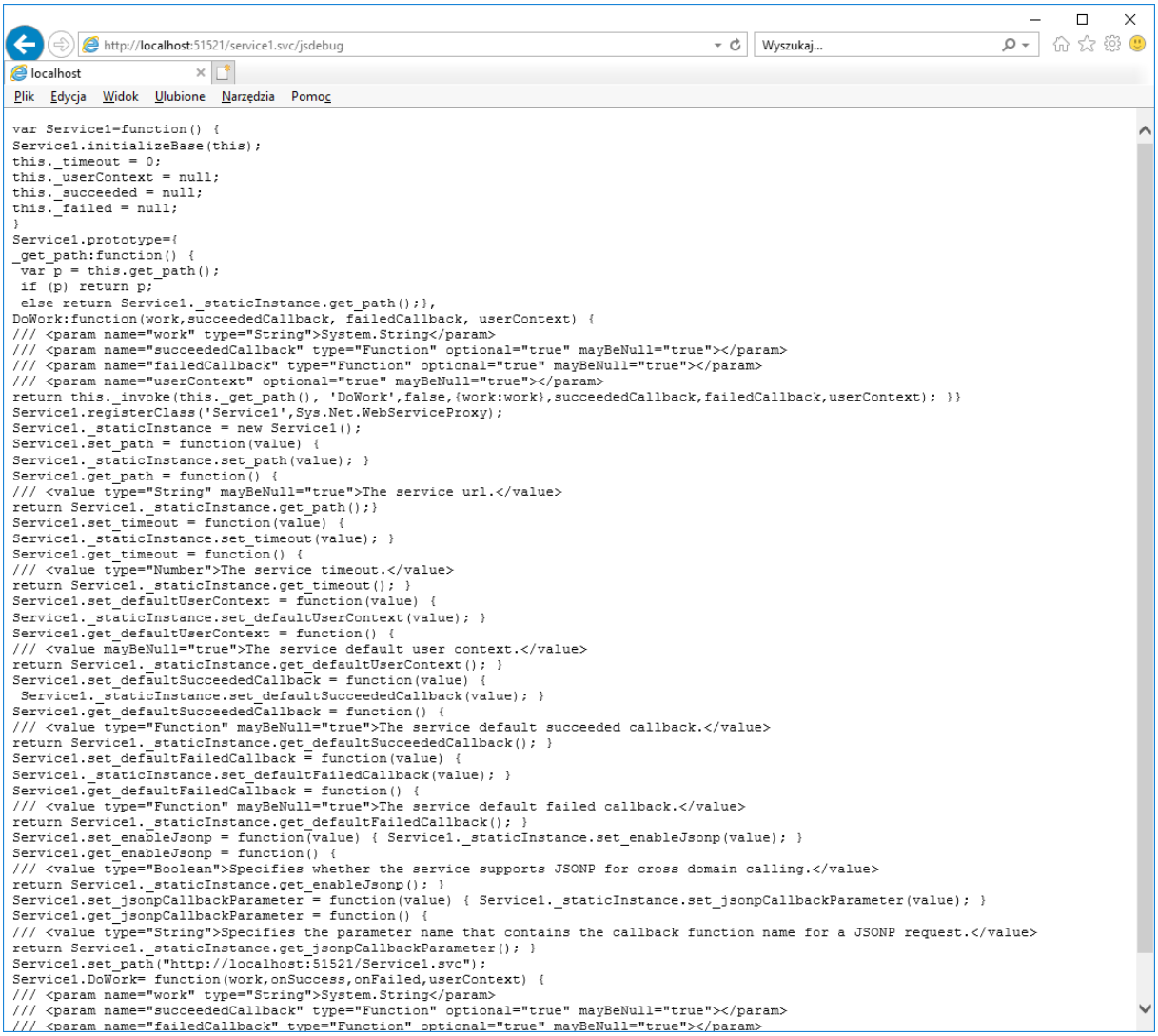
```
[ServiceContract(Namespace = "")]
[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequireme
ntsMode.Allowed)]
public class Service1
{
    [OperationContract]
    public string DoWork( string work )
    {
        // Add your operation implementation here
        return "from server " + work;
    }

}
```

i taka modyfikacja konfiguracji, w której usługa obsługuje żądania JSON

```
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name="WebApplication10.Service1AspNetAjaxBehavior">
          <enableWebScript />
        </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
        <behavior name="">
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="false" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
```

```
      multipleSiteBindingsEnabled="true" />
    <services>
      <service name="WebApplication10.Service1">
        <endpoint address="" behaviorConfiguration="WebApplication10.Service1AspNetAjaxBehavior"
          binding="webHttpBinding" contract="WebApplication10.Service1" />
      </service>
    </services>
  </system.serviceModel>
```

Po stronie przeglądarki można już automatycznie generować proxy Javascript do usługi, jest ono dostępne pod adresem usługi, z sufiksem **/js** (lub **/jsdebug** dla kodu niezminifikowanego), np. http://localhost:12345/service.svc/js



Tak wygenerowany skrypt do poprawnej pracy wymaga jednak fragmentu infrastruktury AJAX, udostępnianego przez Microsoft z CDN pod adresem

```
https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjax.js
```

Pełny kod przykładowego klienta:

```
<!DOCTYPE html>
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script src="https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjax.js"></scr
ipt>
    <script src="/service1.svc/js"></script>
    <script>
        window.addEventListener('load', function () {
            document.getElementById('button1').onclick = function (e) {
                e.preventDefault();
                Service1.DoWork('from javascript', function (result) {
                    document.getElementById('content1').innerHTML = result;
                });
            }
        })
    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <div id="content1"></div>
        <button id="button1">Klik</button>
    </div>
    </form>
</body>
</html>
```

5 CSRF

Korzystanie z AJAX niesie ze sobą nie tylko szereg ułatwień ale również zagrożeń i ograniczeń.

Zacząć wypada od najpoważniejszego zagrożenia – jest to tzw. **C**ross-**s**ite **R**equest **F**orgery. Zagrożenie to można opisać następującym scenariuszem:

- Użytkownik w zakładce przeglądarki przegląda stronę wrażliwą, na przykład stronę banku. Strona ta wymaga uwierzytelnienia, uwierzytelnienie podtrzymywane jest ciastkiem. Hasło do strony banku zna tylko użytkownik. Aplikacja skonstruowana jest standardowo, pewne wrażliwe operacje (np. zlecenie przelewu) powoduje wyświetlenie formularza, który przeglądarka odsyła na serwer jako POST. Na serwerze żądanie jest autentykowane.
- Atakujący przygotowuje witrynę, w której osadza fragment formularza z adresem zwrotnym (action) takim jak adres formularza na atakowanej stronie. Atakujący kontroluje formularz – może go odesłać na serwer na przykład za pomocą Javascript. Atakujący nakłania użytkownika do odwiedzenia jego strony w tej samej przeglądarce w której już w sąsiedniej zakładce otwarta jest wrażliwa strona do której użytkownik jest zalogowany.
- Sesje w przeglądarce które są otwarte na różnych zakładkach **współdzielą** ciastka w obrębie tych samych domen – ten mechanizm wprowadzony dla wygody użytkowników powoduje że ta sama aplikacja otwarta w kolejnej zakładce nie wymaga kolejnego logowania (uznano że byłoby to dla użytkowników nieintuicyjne i kłopotliwe)
- Skrypt ze złośliwej strony **nie ma możliwości odczytu** wartości ciastka wydanego przez wrażliwą stronę, ale **ma możliwość wysłania** żądania typu POST z formularza, który to POST zostanie poprawnie obsłużony przez wrażliwą aplikację.

Mówiąc obrazowo – ze swoje strony atakujący może zlecać przelewy na stronie banku użytkownika, pod warunkiem że użytkownik jest do niej zalogowany.

5.1 Przykład - atak

Strona wrażliwa:

```
[Authorize]
public class HomeController : Controller
{
    [HttpGet]
    public ActionResult Index()
    {
        HomeModel model = new HomeModel();
        return View(model);
    }

    [HttpPost]
    public ActionResult Index(HomeModel model)
    {
        if ( !string.IsNullOrEmpty(model.SensitiveData))
        {
            this.Submits.Add(model.SensitiveData);
        }
        model.Submits = this.Submits;
        return View(model);
    }

    public List<string> Submits
    {
        get
```

```
        {
            if ( this.Session["submits"] == null )
            {
                this.Session["submits"] = new List<string>();
            }

            return this.Session["submits"] as List<string>;
        }
    }
}
```

Jej model

```
public class HomeModel
{
    public string SensitiveData { get; set; }

    public List<string> Submits { get; set; }
}
```

Jej widok

```
@model ActualWebApplication.Models.HomeModel
@{
    ViewBag.Title = "View";
}

<h2>View</h2>

@using (var form = Html.BeginForm())
{
    <div>
        Zalogowany jako @this.User.Identity.Name
    </div>
    <div>
        @Html.TextBoxFor(m => m.SensitiveData)
    </div>
    <div>
        <button>Zapisz</button>
    </div>
    <div>
        @if (Model.Submits != null)
        {
            <table>
                @foreach ( var submit in Model.Submits )
                {
                    <tr>
                        <td>
                            @submit
                        </td>
                    </tr>
                }
            </table>
        }
    </div>
}
```

Żądanie Home/Index typu POST jest tu przykładem wrażliwego żądania. Programista dochował staranności – żądanie jest poprawnie chronione przez **Authorize**.

Atakujący przygotowuje jednak stronę:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
        <meta charset="utf-8" />
    <script>
        window.addEventListener('load', function () {
            document.getElementById('button1').addEventListener('click', functio
n () {
                document.forms[0].submit();
                window.alert('submited');
            });
         })
    </script>
</head>
<body>

    <div style="border: 1px solid black; padding: 30px">

        <h4>Cross site request forgery</h4>

        <iframe width="0" height="0" border="0" name="dummyframe" id="dummyframe
"></iframe>

        <form action="http://localhost:60461" method="post" target="dummyframe">
            <input name="SensitiveData" />
        </form>

        <button id="button1">Wyślij formularz do sąsiedniej aplikacji http://loc
alhost:60461</button>

    </div>
</body>
</html>
```

w której umieszcza formularz i fragment skryptu który w którymś momencie odsyła formularz na serwer – tu do wrażliwej aplikacji.

5.2 Przykład – ochrona

Ochrona przez atakiem CSRF jest możliwa i wręcz – wymagana. Uważa się ją za standardowy element aplikacji w której pewne żądania są obsługiwane we wrażliwy sposób (POST). Oznacza to że wrażliwe fragmenty aplikacji **należy bezwarunkowo i zawsze** chronić w ten sposób.

Ochrona polega na użyciu tzw. antiforgery token.

Pomysł polega na tym, żeby wrażliwa aplikacja do wrażliwego formularza dodawała jeszcze dodatkowo:

- Losową wartość literalną, dla wygody użytkownika do formantu **input** typu **hidden**, przez co wartość ta będzie obecna w żądaniu zwrotnym ale użytkownik jej nie zobaczy
- Ciasteczka z powtórzeniem tej samej wartości losowej, którą dołączono do formularza

Przy obsługiwaniu żądania typu POST serwer wrażliwej aplikacji dodatkowo sprawdza **zgodność** obu wartości – tej z formularza z tą z ciasteczka. W przypadku braku zgodności, serwer wrażliwej aplikacji uznaje że jest to próba ataku.

W ASP.NET MVC użycie antiforgery token jest zautomatyzowane i wymaga dwóch elementów:

- Atrybutu nad akcją kontrolera która ma być chroniona

```
[HttpPost]
// ochrona przed atakiem:
[ValidateAntiForgeryToken]
public ActionResult Index(HomeModel model)
{
    if ( !string.IsNullOrEmpty(model.SensitiveData))
```

- Użyciu funkcji pomocniczej do wyrenderowania formantu z losową wartością

```
@using (var form = Html.BeginForm())
{
    @* ochrona przed atakiem *@
    @Html.AntiForgeryToken()
    <div>
        Zalogowany jako @this.User.Identity.Name
    </div>
```

Po zastosowaniu tej ochrony, atakujący przy próbie odesłania wrażliwego formularza na serwer zobaczy wyjątek HTTP 500.

Jak i dlaczego działa ta ochrona?

Okazuje się że zastosowanie równocześnie pary miejsc w których umieszczona jest informacja (formant i ciasteczko) powoduje że atakujący nie może nigdy doprowadzić do sytuacji w której uda mu się odesłać formularz na serwer tak aby obie te wartości były równe (co jest wymagane na serwerze i co dzieje się w trybie normalnej pracy witryny):

- Atakujący może do swojego formularza dodać formant z dowolną losową wartością. Ale atakujący nie może z poziomu własnego skryptu **ustawiać** ciasteczka dla innej domeny (tu: wrażliwej domeny z której pochodzi aplikacja) – jest to ograniczenie samej przeglądarki
- Atakujący mógłby próbować dodać do formularza formant z nie „dowolną" wartością, tylko dokładnie taką wartością jaka została ustawiona w ciastku wydanym jako element ochrony. Atakujący nie może jednak z poziomu własnego skryptu **odczytać** ciasteczka dla innej domeny (tu: wrażliwej domeny z której pochodzi aplikacja) – jest to ograniczenie samej przeglądarki

6 CORS

O ile CSRF jest zagrożeniemo tyle CORS jest **ograniczeniem** dla technologii AJAX. Sama idea jest podobna jak w przypadku CSRF, dotyczy jednak nie formularzy ale usług typu REST/SOAP, do których inna aplikacja (hipotetycznego atakującego) próbowałaby uzyskać dostęp nie przez odesłanie formualrza, ale za pomocą obiektu XMLHttpRequest (AJAX):

- Wrażliwa aplikacja udostępnia część danych swojej aplikacji klienckiej jako usługi REST/SOAP. Aplikacja poprawnie uwierzytelnia dostęp – wymaga ciasteczka które jest przyznawane sesji użytkownika
- Atakujacy przygotowuje witrynę w której umieszcza skrypt Javascript, który za pomocą obiektu XMLHttpRequest będzie tworzył żądania do wrażliwej aplikacji. Obiekt XMLHttpRequest automatycznie (jak w sytuacji z dwiema zakładkami) dołączy ciasteczka z wrażliwej domeny do żądania kierowanego do wrażliwej domeny

I tu niespodzianka – o ile w przypadku CSRF takie zachowanie domyślnie jest dozwolone, a ochrona przed nim wymaga dodatkowej pracy (antiforgery token), to w przypadku żądań AJAX takie zachowanie jest **domyślnie niedozwolone!** Próba wykonania żądania między różnymi domenami jest blokowana przez przeglądarkę.

Takie domyślne zachowanie chroni wrażliwą aplikację przed zagrożeniami ze strony innych aplikacji, udostępnianych w sieci przed hipotetycznymi atakującymi.

Ograniczenie polega jednak na tym, że czasem taka sytuacja w której aplikacja A udostępnia usługę z której chciałaby skorzystać aplikacja B jest **pożądana**. Niestety, domyślne z powodu ograniczeń bezpieczeństwa, tak nie jest. CORS jest więc techniką która wychodzi naprzeciw takiemu scenariuszowi.

6.1 Jak działa ochrona

Ochrona wbudowana w przeglądarkę przed żądaniami między domenami polega na tym że przeglądarka:

- Wykonując żądanie typu GET zakłada że wielokrotne wykonanie GET jest *idempotentne* (czyli nie zmienia stanu danych na serwerze). Przeglądarka wykona więc żądanie, a następnie sprawdzi czy w odpowiedzi znajdują się nagłówki:
  - Access-Control-Allow-Origin
  - Access-Control-Allow-Metods
  - Access-Control-Allow-Credentials

  Jeśli nagłówków nie ma lub ich wartości nie zgadzają się z parametrami żądania – obiekt XMLHttpRequest zgłasza błąd bezpieczeństwa

- Wykonując żądanie innego typu niż GET (np. PUT) przeglądarka mogłaby *zmienić stan danych* na serwerze, mimo tego żądanie mogłoby być niedozwolone – ale o tym przeglądarka jeszcze nie wie, ponieważ odczyta to z nagłówków odpowiedzi. Jest to sytuacja trochę „patowa" – przeglądarka musiałaby uzależnić wysłanie żądania od tego jaką dostanie na nie odpowiedź, co jest oczywiście sprzecznością. Dlatego w takich sytuacjach przeglądarka wykonuje dodatkowe żądanie, tzw. preflight. Jest to żądanie typu HEAD, a więc nie ma żadnych danych i żadnego ciała odpowiedzi, ale przeglądarka zwraca uwagę czy do odpowiedzi są dołączane nagłówki CORS i jeśli są – wykonuje kolejne, właściwe żądanie (np. PUT)

6.2 Przykład

Serwerowa usługa WebAPI

```
public class DataController : ApiController
 {
     public IHttpActionResult Options()
     {
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Origin", "http://localhost:60443");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Methods", "GET,PUT,POST,DELETE");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Credentials", "true");

         return Ok();
     }

     public IHttpActionResult Get()
     {
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Origin", "http://localhost:60443");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Methods", "GET,PUT,POST,DELETE");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Credentials", "true");

         return Ok(new[]
         {
             new DataModel() { ID = 1, Data = "foo" },
             new DataModel() { ID = 2, Data = "bar" }
         });
     }

     public IHttpActionResult Put()
     {
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Origin", "http://localhost:60443");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Methods", "GET,PUT,POST,DELETE");
         HttpContext.Current.Response.AppendHeader("Access-Control-Allow-
Credentials", "true");

         return Ok(new[]
         {
             new DataModel() { ID = 1, Data = "foo" },
             new DataModel() { ID = 2, Data = "bar" }
         });
     }
 }
```

```
public class DataModel
{
    public int ID { get; set; }
    public string Data { get; set; }
}
```

Klient:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
        <meta charset="utf-8" />
    <script>
        window.addEventListener('load', function () {

            document.getElementById('button2').addEventListener('click', functio
n () {

                var xhr = new XMLHttpRequest();
                xhr.open('put', 'http://localhost:60461/api/Data', true);
                xhr.withCredentials = true;

                xhr.onreadystatechange = function () {
                    if (xhr.readyState == XMLHttpRequest.DONE) {
                        document.getElementById('content1').innerHTML = xhr.resp
onseText;
                    }
                }
                xhr.onerror = function () {
                    document.getElementById('content1').innerHTML = 'błąd';
                }

                xhr.send();


            });
        })
    </script>
</head>
<body>

    <div style="border: 1px solid black; padding: 30px">

        <h4>CORS</h4>

        <div id="content1">

        </div>

        <button id="button2">Wyślij żądanie typu CORS</button>

    </div>
</body>
</html>
```