# Dodatek do książki:


Autor:

*Krzysztof Żydzik*

Dodatek jest bezpłatny i nie ma związku z wydawnictwem Helion. Sugestie i uwagi odnośnie dodatku proszę zgłaszać na stronie [aspnetmvc.pl](aspnetmvc.pl), z której można pobrać najnowszą wersję dodatku.

V1.0

# Od autora

1. Podziękowania dla wszystkich, którzy wsparli książkę swoim zakupem. Dzięki Wam książka stała się najczęściej kupowaną pozycją techniczną w Polsce i aktualnie zajmuje pierwsze miejsca na liście bestsellerów wydawnictwa Helion.

2. Jeśli skorzystałeś z książki i była pomocna poleć ją innym tutaj klikając przycisk Poleć:


3. Paczka plików z aplikacją stworzoną w dodatku znajduje się tutaj.

## Spis treści


**Od autora** ..... **2**

**Wstęp** ..... **5**

**Rozdział 1. Wyświetlanie danych z API przy pomocy jQuery i AJAX** ..... **6**

**Teoria** ..... **6**

**Praktyka** ..... **6**

Aktualizacja pliku _Layout.cshtml ..... 6

Zmiana nazwy odnośnika ..... 6

Dodanie dodatkowych odnośników w menu ..... 6

Wyświetlanie danych z API na stronie przy pomocy jQuery ..... 7

Kontroler ..... 7

Widok ..... 8

Kod jQuery i JavaScript dla widoku ..... 8

Metoda ShowResponse() ..... 8

Kompletny kod widoku ..... 9

**Rozdział 2. Bezpieczne dodawanie i usuwanie zoptymalizowanych zdjęć z miniaturkami do chmury Azure** ..... **11**

**Teoria** ..... **11**

Dlaczego zdjęcia w chmurze? ..... 11

Zalecenia w odniesieniu do przechowywania plików ..... 11

Przechowywanie zdjęć w chmurze ..... 12

Przygotowanie konta na Microsoft Azure ..... 12

**Praktyka** ..... **15**

Dane dostępowe do chmury ..... 15

Dodawanie klasy z modelem ..... 16

Aktualizacja DbContext ..... 17

Repozytorium dla zdjęć ..... 18

Metody w repozytorium ..... 18

Interfejs dla repozytorium ..... 18

Implementacja repozytorium ..... 18

Wstrzykiwanie implementacji przy pomocy IoC ..... 19

Migracje dla klasy Zdjecie ..... 19

Nowy projekt Services ..... 19

Instalacja niezbędnych bibliotek w projekcie Services ..... 20

Implementacja klas odpowiedzialnych za obróbkę i zapis zdjęć ..... 20

Sprawdzenie rozszerzenia pliku ..... 20

Generowanie miniaturek i zmiana rozmiaru zdjęcia ..... 21

Uniwersalne rozwiązanie do generowania miniaturek ..... 21

Metoda do zmiany rozmiaru zdjęć ..... 22

Pomocnicze klasy i metody ..... 22

Klasa BlobImage ..... 22

Klasa StreamHelpers i metoda ReadFully ..... 23

Klasa ImageUpload i dodawanie oraz usuwanie zdjęć wraz z miniaturkami ..... 23

Dodawanie Zdjęć ..... 23

Metoda CreateBlobName() ..... 23

Metoda GetFullBlobName() ..... 24

Metoda UploadImageAndReturnImageName() ..... 24

Metoda GenerateImageMiniatures() ..... 24

Metoda UploadMultipleImagesToBlob() ..... 25

Usuwanie zdjęć ..... 26

Metoda DeleteImageByNameWithMiniatures() ..... 26

Metoda DeleteImageByName() ..... 26

Kod kompletnej klasy ImageUpload ..... 26

Kontroler do galerii zdjęć ..... 28

Konstruktor i wstrzykiwanie repozytorium ... 28
Metoda Lista() ... 28
Metoda UploadImage() ... 28
Metoda DeleteImage() ... 29
Kompletny kod kontrolera Galeria ... 29
Widok dla galerii zdjęć ... 30
Kod HTML dla widoku ... 31
Kod JS i jQuery dla widoku ... 32
Funkcja ReadURL() ... 32
Funkcja checkFile() ... 32
Funkcja UsunZdjecie() ... 32
Funkcja JQuery Lazy() ... 33
Funkcja wyłapująca zmiany stanu w polu input ... 33
Kompletny kod JS dla widoku ... 33
Testowanie galerii ... 35
**Rozdział 3. Bezpieczna walidacja pól wejściowych z kodem HTML z edytora WYSIWYG ... 36**
**Teoria ... 36**
**Praktyka ... 37**
Biblioteka do walidacji ... 37
Klasy z modelem ... 37
Kod klasy Edytor ... 37
Klasa Uzytkownik ... 37
Aktualizacja DbContext ... 37
Migracje ... 37
Kontroler ... 38
Kod metody GET ... 38
Kod metody POST ... 38
Kompletny kod kontrolera: ... 39
Widok ... 40
Edytor WYSIWYG ... 43
Wygląd edytora TinyMCE: ... 44
**Podsumowanie ... 45**
**Co dalej? ... 45**

# Wstęp

Dodatek jest kontynuacją rozdziału: **8. Aplikacja i wdrożenie** z książki: **C# 6.0 i MVC 5. Tworzenie nowoczesnych portali internetowych.** Książkę można zakupić **tutaj**.

W dodatku nie zostały szczegółowo opisane rzeczy takie jak: dodawanie kontrolerów, dodawanie klas z modelem, migracje itp., które zostały już wiele razy użyte podczas tworzenia aplikacji z książki i powinny być opanowane przez każdego czytelnika.

Dodatek składa się z 2 części dla każdego zagadnienia:

- **teorii** na temat tego, co jest do zrobienia i jak to zrobić,
- **praktyki** a więc gotowych kodów z opisem do użycia w aplikacji.

Aplikacja z książki poszerzona o elementy z dodatku znajduje się tutaj.

# Rozdział 1. Wyświetlanie danych z API przy pomocy jQuery i AJAX

## Teoria

W tym rozdziale zaktualizujemy plik _Layout.cshtml oraz dorobimy podstronę, która pobiera listę kategorii z API i wyświetla bez odświeżania strony przy pomocy jQuery i AJAX.

## Praktyka

### Aktualizacja pliku _Layout.cshtml

Na początek zaktualizujemy plik z szablonem strony.

#### Zmiana nazwy odnośnika

Zmieniamy nazwę odnośnika do strony głównej na "`Aplikacja C# 6.0 i MVC 5`".

Kod po zmianach będzie wyglądał następująco (23 linia w pliku _Layout.cshtml):

```
@Html.ActionLink("Aplikacja C# 6.0 i MVC 5", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
```

#### Dodanie dodatkowych odnośników w menu

Dodajemy w menu linki do już wcześniej zaimplementowanych metod z kontrolera kategoria:

```
<li class="dropdown">
   <a href="#" class="dropdown-toggle" data-toggle="dropdown">Kategorie <span class="caret"></span></a>
   <ul class="dropdown-menu" role="menu">
        <li>@Html.ActionLink("Lista kategorii", "Index", "Kategoria")</li>
        <li><a href="../JSON/">Lista kategorii w JSON</a></li>
        <li>@Html.ActionLink("Lista pobrana z API", "Lista", "Kategoria")</li>
   </ul>
</li>
```

Wklejamy kod pod kodem odpowiedzialnym za linki do metod z kontrolera ogłoszenia w pliku _Layout.cshtml. Dodaliśmy w ten sposób link do podstrony z listą kategorii, link do metody API zwracającej listę kategorii w formacie JSON oraz link do podstrony wyświetlającej listę kategorie na podstawie danych z API (ta podstrona będzie tworzona w kolejnych krokach dodatku).

Kolejnym krokiem jest dodanie dwóch nowych odnośników do metod, które będziemy tworzyć w kolejnych etapach dodatku a więc do galerii zdjęć oraz edytora HTML.

Kod, który należy wkleić poniżej wcześniej dodanego kodu z kategoriami:

```
<li>@Html.ActionLink("Zdjęcia", "Lista", "Galeria")</li>
<li>@Html.ActionLink("Edytor HTML", "EdytujTresc", "Edytor")</li>
```

Usuwamy odnośnik o nazwie HOME.

Ostatecznie kod menu wygląda następująco:

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
     <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
     </button>

            @Html.ActionLink("Aplikacja C# 6.0 i MVC 5", "Index", "Home", new { area = "" }, new {
                                                                    @class = "navbar-brand" })
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">

                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown">Ogłoszenia <span
                                                            class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li>@Html.ActionLink("Lista ogłoszeń", "Index", "Ogloszenie")</li>
                        @if (User.Identity.IsAuthenticated)
                        {
                            <li>@Html.ActionLink("Dodaj ogłoszenie", "Create", "Ogloszenie")</li>
                        }
                        <li class="divider"></li>
                        <li>@Html.ActionLink("Lista jako PartialView", "Partial", "Ogloszenie")</li>
                    </ul>
                </li>

                <li class="dropdown">
                        <a  href="#"  class="dropdown-toggle"  data-toggle="dropdown">Kategorie  <span
                                                                class="caret"></span></a>
                        <ul class="dropdown-menu" role="menu">
                                <li>@Html.ActionLink("Lista kategorii", "Index", "Kategoria")</li>
                                <li><a href="../JSON/">Lista kategorii w JSON</a></li>
                                <li>@Html.ActionLink("Lista pobrana z API", "Lista", "Kategoria")</li>
                        </ul>
                </li>

                <li>@Html.ActionLink("Zdjęcia", "Lista", "Galeria")</li>
                <li>@Html.ActionLink("Edytor HTML", "EdytujTresc", "Edytor")</li>
          </ul>

          @Html.Partial("_LoginPartial")
        </div>
    </div>
</div>
```

# Wyświetlanie danych z API na stronie przy pomocy jQuery

Wyświetlimy dane zwracane w formacie JSON na osobnej podstronie przy pomocy jQuery. To rozwiązanie można wykorzystać również w PHP, Javie, Ruby, itd. oraz na urządzeniach mobilnych, ponieważ dane są wyświetlane przy pomocy czystego HTML i jQuery. Początkowo dane miały być wyświetlone na osobnej stronie www (inna domena niż ta, z której pochodzą dane z API) jednak trzeba by użyć CORS lub JSONP, dlatego dla ułatwienia stworzymy osobną podstronę i po kliknięciu przycisku będziemy pobierać dane z API przy pomocy zapytania AJAX a następnie wyświetlać na podstronie bez odświeżania strony.

## Kontroler

Dodajemy nową metodę o nazwie Lista w kontrolerze Kategoria zwracająca widok:

```
public ActionResult Lista()
{
        return View();
```

```
}
```

## Widok

Dodajemy nowy plik z widokiem o nazwie Lista do folderu Views/Kategoria:

```
@{
    ViewBag.Tytul = "Lista kategorii z API";
}

<h1>Lista kategorii pobranych z API</h1>
<div id="button-blue" class="btn btn-primary">Pobierz kategorie z API bez odświeżania strony</div>
<div class="kategorie_lista"></div>
```

## Kod jQuery i JavaScript dla widoku

Kolejnym krokiem jest stworzenie w jQuery metody, która po kliknięciu przycisku pobierze dane z metody (KategorieWJson) zwracającej kategorie w formacie JSON:

```
@section scripts{
<script type="text/javascript">
$(document).ready(function() {
        $('#button-blue').click(function(){
                var url = "http://localhost:33819/JSON";

         $.ajax({
            url: url,
            type: 'GET',
            dataType: 'json',
            success: function (data) {
                ShowResponse(data);
            },
            error: function (x, y, z) {
                alert(x + '\n' + y + '\n' + z);
            }
                });
    });
});

</script>
}
```

## Metoda ShowResponse()

Pozostało jeszcze zaimplementować metodę ShowResponse(), która wyświetli pobrane dane w formacie HTML:

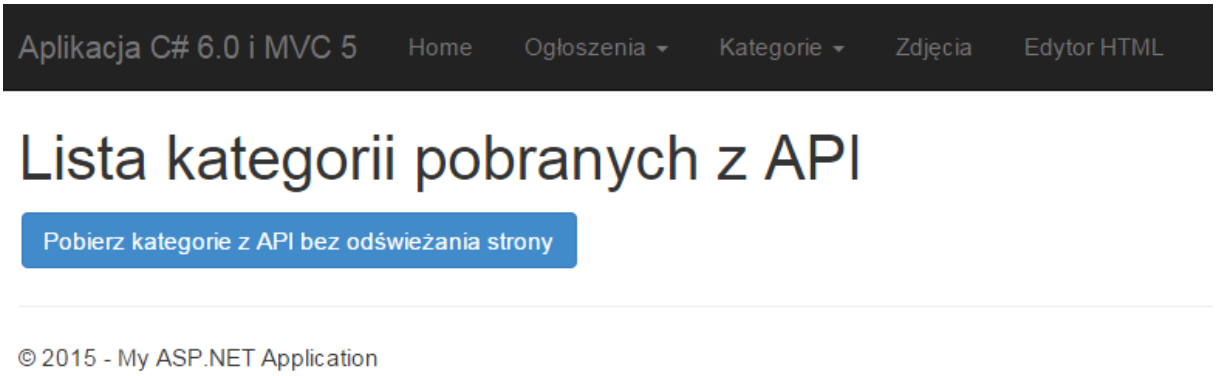
```
function ShowResponse(kategorie) {
        var strResult = '<h2>Lista kategorii wyświetlona przy pomocy jQuery:</h2><table
class="table"><tr><th>Nazwa kategorii:</th><th>Id rodzica:</th><th>Tytuł w Google:</th></tr>';
        $.each(kategorie, function (index, kategoria) {
                strResult += '<tr>'
                        + '<td><a href="/Kategoria/PokazOgloszenia/' + kategoria.Id + '">'
                                                                + kategoria.Nazwa + '</a></td>'
                        + '<td>' + kategoria.ParentId + '</td>'
                        + '<td>' + kategoria.MetaTytul + '</td>';
        });
        strResult += "</tr></table><hr />";
        //wyswietlamy sformatowane dane w HTML
        $(".kategorie_lista").html(strResult);

        //wyswietlamy kod JSON zwrócony w API
        $('.kategorie_lista').append("<h2>Dane pobrane z API:</h2>"+JSON.stringify(kategorie));
}
```

Aby metoda zadziałała należy podać prawidłowy port(localhost), na którym działa aplikacja.

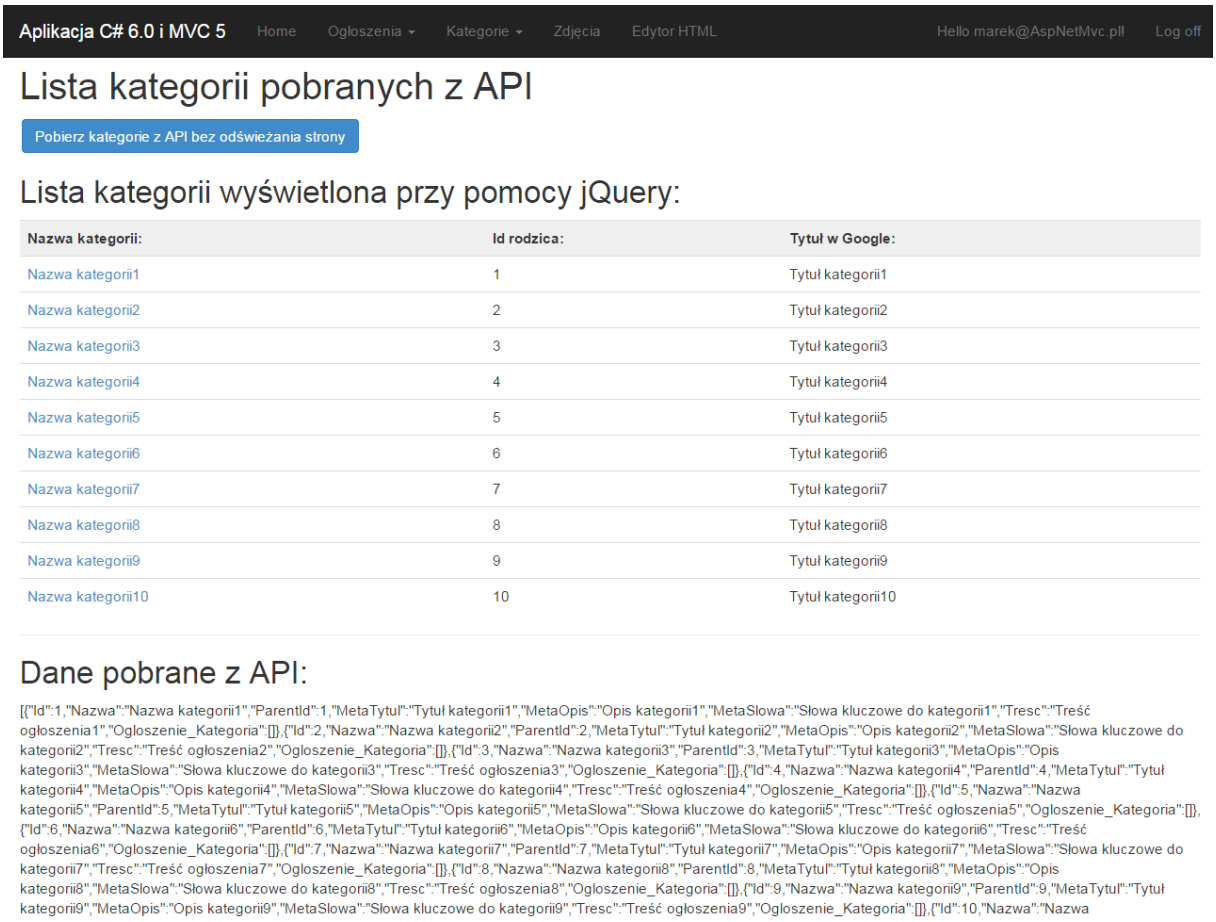
## Kompletny kod widoku

```
@{
    ViewBag.Tytul = "Lista kategorii z API";
}

<h1>Lista kategorii pobranych z API</h1>
<div id="button-blue" class="btn btn-primary">Pobierz kategorie z API bez odświeżania strony</div>
<div class="kategorie_lista"></div>

@section scripts{
<script type="text/javascript">
$(document).ready(function() {
        $('#button-blue').click(function(){
                var url = "http://localhost:33819/JSON";

         $.ajax({
            url: url,
            type: 'GET',
            dataType: 'json',
            success: function (data) {
                ShowResponse(data);
            },
            error: function (request, status, error) {
                alert(request.responseText);
            }
                });
    });
});

function ShowResponse(kategorie) {
        var strResult = '<h2>Lista kategorii wyświetlona przy pomocy jQuery:</h2><table
class="table"><tr><th>Nazwa kategorii:</th><th>Id rodzica:</th><th>Tytuł w Google:</th></tr>';
        $.each(kategorie, function (index, kategoria) {
                strResult += '<tr>'
                        + '<td><a href="/Kategoria/PokazOgloszenia/' + kategoria.Id + '">' +
kategoria.Nazwa + '</a></td>'
                        + '<td>' + kategoria.ParentId + '</td>'
                        + '<td>' + kategoria.MetaTytul + '</td>';
        });
        strResult += "</tr></table><hr />";
        //wyswietlamy sformatowane dane w HTML
        $(".kategorie_lista").html(strResult);

        //wyswietlamy kod JSON zwrócony w API
        $('.kategorie_lista').append("<h2>Dane pobrane z API:</h2>"+JSON.stringify(kategorie));
}
</script>
}
```

Podstrona przed kliknięciem przycisku pobierającego dane:




Podstrona po kliknięciu przycisku (pobraniu i wyświetleniu danych z API):



# Rozdział 2. Bezpieczne dodawanie i usuwanie zoptymalizowanych zdjęć z miniaturkami do chmury Azure

W niniejszym rozdziale dorobimy funkcjonalność dodawania zdjęć na naszym portalu i zapisywania ich w chmurze Azure w postaci BLOBów. Podczas dodawania zdjęcia będą optymalizowane pod względem wielkości i wymiarów oraz dla bezpieczeństwa sprawdzane i walidowane rozszerzenia plików. Dorobimy również usuwanie zdjęć bez przeładowywania strony za pomocą JS i jQuery. Na początku trochę teorii później praktyka a więc dorabianie funkcjonalności dodawania zdjęć do aplikacji z książki.

## Teoria

### Dlaczego zdjęcia w chmurze?

Jeśli startujesz z nowym portalem z pewnością nie wiesz, jaką zdobędzie popularność. Chmura zapewnia pełną skalowalność rozwiązania a dodatkowo maksymalnie upraszcza zarządzanie ścieżkami do plików, dlatego jest w początkowym stadium rozwoju portalu idealnym i niedrogim rozwiązaniem. W przyszłości, gdy unormuje się zapotrzebowanie na zasoby sprzętowe będzie można w łatwy sposób przejść na dedykowany serwer plików w celu obniżenia kosztów.

### Zalecenia w odniesieniu do przechowywania plików

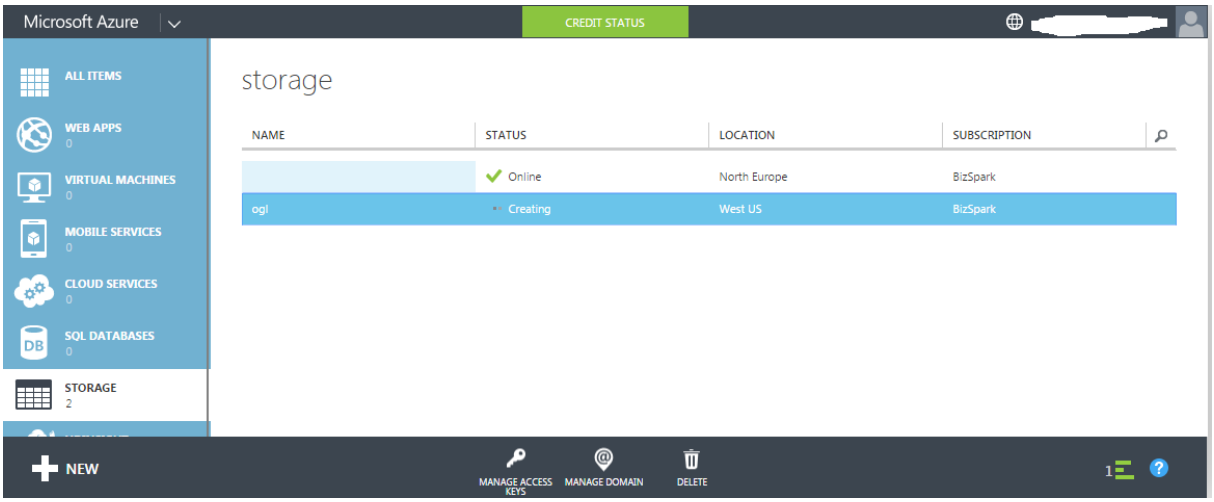
- nie ufaj plikom wgrywanym przez użytkowników,
- sprawdzaj typ pliku, rozszerzenie pliku nie zawsze musi się pokrywać z faktycznym typem pliku, który odczytuje się z początkowych bitów pliku (plik z rozszerzeniem .jpg może być w rzeczywistości złośliwym plikiem .exe),
- przechowuj pliki w przeznaczonych do tego katalogach poza strukturą projektu np. w folderze Content w projekcie ASP.NET MVC.
- nie ufaj nazwie pliku, nadawaj zawsze własną nazwę pliku, może być nią np. losowa wartość typu Guid.
- nie przechowuj zbyt wielu plików w jednym folderze, dobrym rozwiązaniem jest stworzenie osobnego folderu dla każdego użytkownika a w nim folderów dla różnych typów zdjęć lub rozmieszczanie zdjęć na kilku poziomach folderów według pierwszych liter nazwy pliku np. plik o nazwie rozpoczynającej się od abc będzie posiadał ścieżkę do pliku: ../a/b/c/nazwapliku.jpg.

# Przechowywanie zdjęć w chmurze

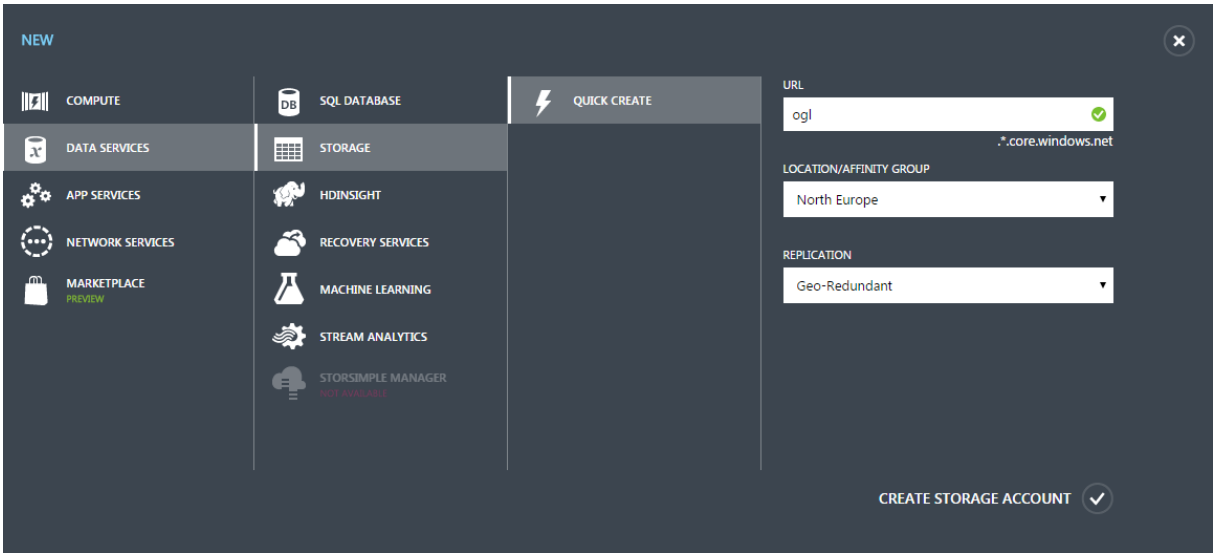
Zdjęcia najlepiej przechowywać jako BLOBy czyli duże pliki binarne. Każdy plik znajduje się w kontenerze. Nie ma ograniczeń na ilość plików w kontenerze, ponieważ jest to tylko wirtualna ścieżka. W rzeczywistości kontenery nie są folderami a pliki są składowane w innej, ukrytej dla użytkownika strukturze.

# Przygotowanie konta na Microsoft Azure

- Na początek należy założyć konto na Azure. Link do bezpłatnej wersji testowej znajduje się tutaj. Wymagane jest konto email w usłudze Microsoftu czyli outlook.com, live.com lub podobne.
- Po założeniu konta i zalogowaniu pojawia nam się panel administracyjny Microsoft Azure:

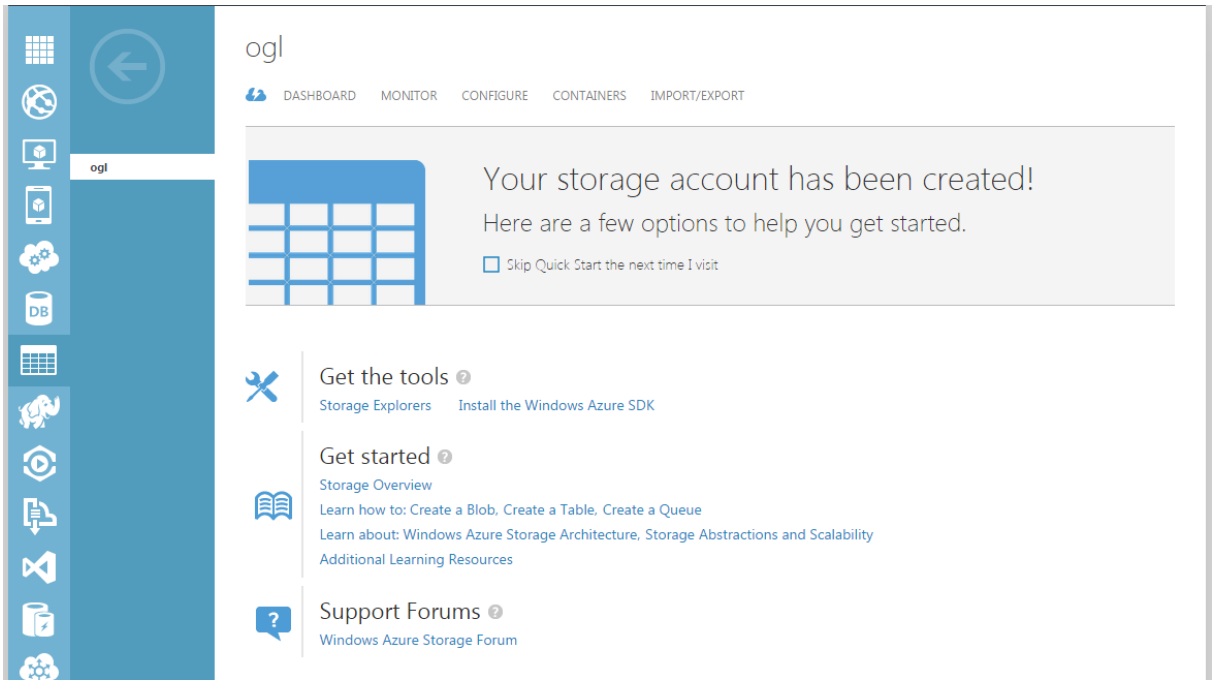



- Aby stworzyć nowe konto typu STORAGE klikamy na dole po lewej stronie przycisk NEW i otwiera nam się okno:

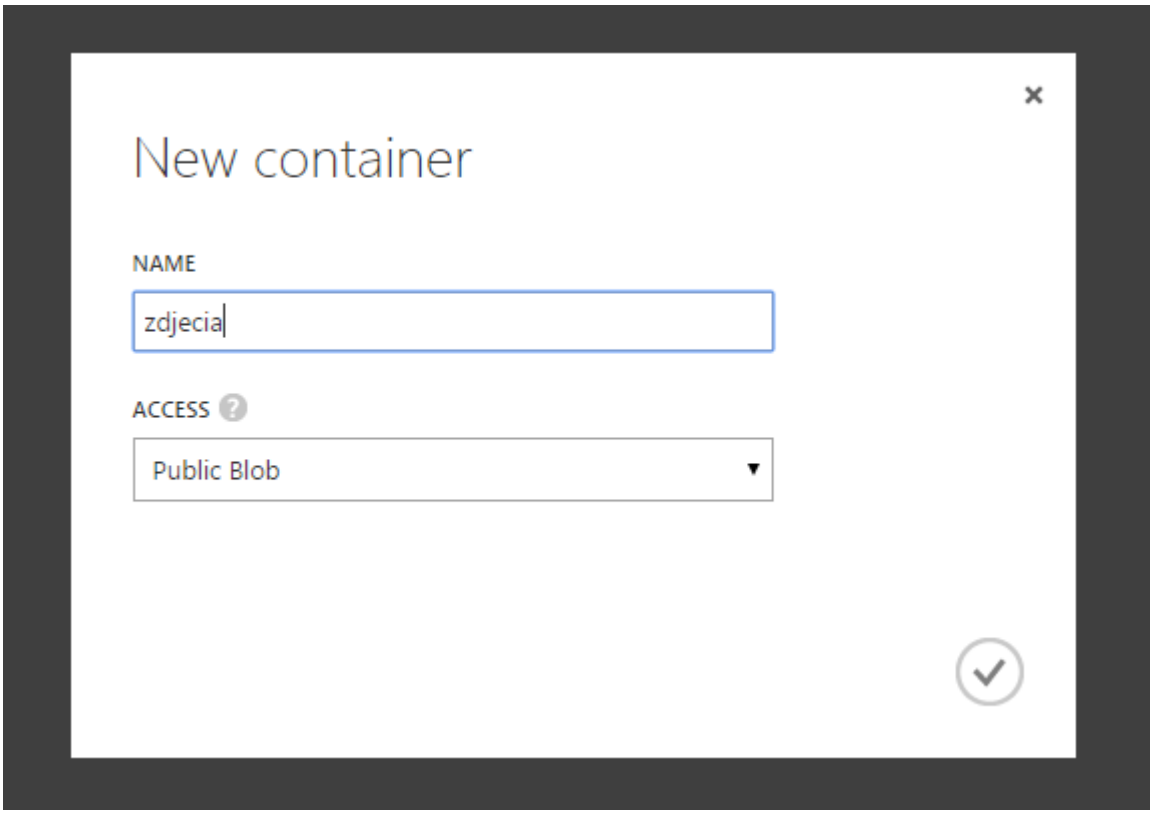


- Wybieramy analogicznie jak na zrzucie ekranu DATA SERVICES > STORAGE > QUICK CREATE i podajemy dowolną nazwę naszego konta, nazwa konta będzie równocześnie elementem adresu url do naszych plików. Dodatkowo możemy wybrać lokalizację, czyli w jakiej serwerowni mają się znajdować nasze pliki (im bliżej tym lepiej bo mniejsze opóźnienia) oraz typ replikacji czyli w jaki sposób mają być przechowywane nadmiarowe kopie plików. Wszystkie dane w Azure znajdują sie przynajmniej w 2 kopiach (za dopłata można posiadać więcej nadmiarowych kopii), w razie awarii jednej serwerowni obciążenie przenoszone jest na drugą kopie plików bez utraty danych.
  Po stworzeniu konta należy chwilę poczekać aż wszystko zostanie skonfigurowane.

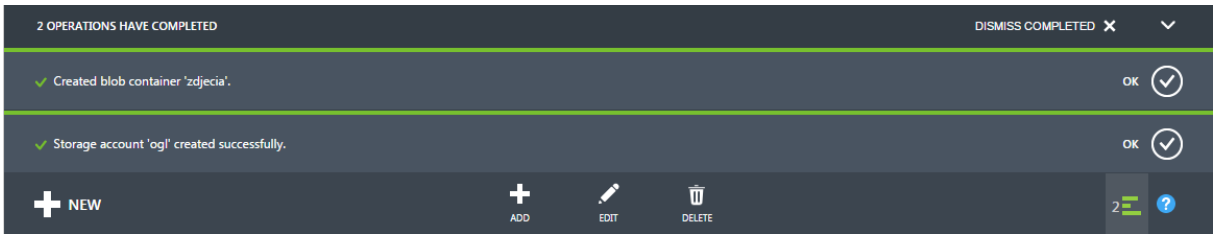
- Widok panelu administratora po utworzeniu naszego konta:



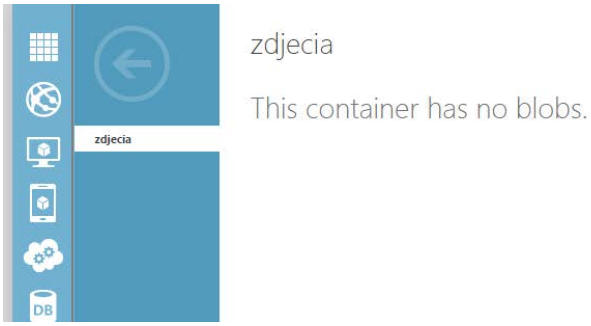

- Aby utworzyć nowy kontener przechodzimy do zakładki CONTAINERS i klikamy ADD A CONTAINER. Otwiera się następujące okno:




- Podajemy nazwę kontenera w naszym przypadku będzie to nazwa "zdjecia". Dostępność ustawiamy na public blob, czyli każdy będzie mógł wyświetlić nasze zdjęcie.
- Na dole mamy podgląd wykonanych operacji:




- Po stworzeniu kontenera i kliknięciu na jego nazwę otwiera nam sie jego zawartość. W tym momencie dostajemy wiadomość, że kontener "zdjecia" jest pusty:

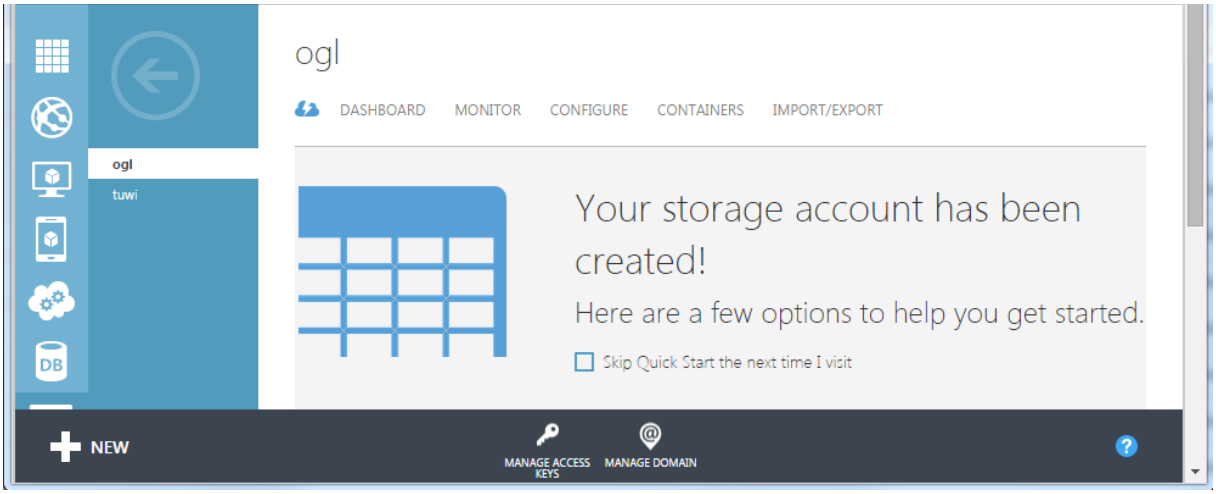


# Praktyka

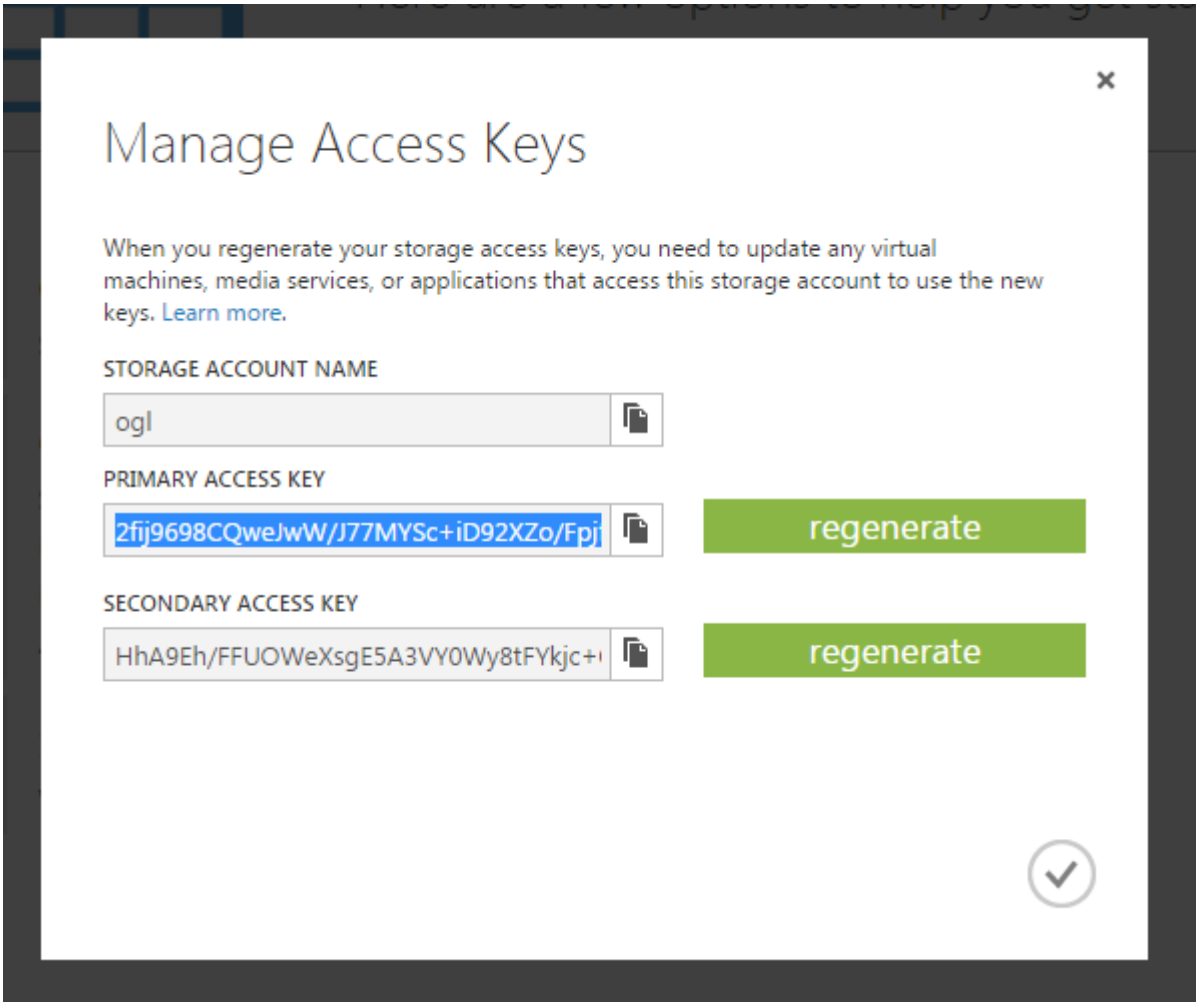
## Dane dostępowe do chmury

W tym kroku zapiszemy dane dostępowe do chmury w pliku Web.config.

Aby odczytać klucz dostępowy przechodzimy do naszego konta typu STORAGE w panelu administracyjnym Azure i na dole ekranu klikamy MANAGE ACCES KEYS:




Po kliknięciu MANAGE ACCESS KEY otwiera nam się okno:



Kopiujemy wartość PRIMARY ACCESS KEY i wklejamy ją w miejsce TwojKlucz w poniższym kodzie, oraz wartość STORAGE ACCOUNT NAME w miejsce NazwaKonta (w 2 miejscach).

```
<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=NazwaKonta;AccountKey=TwojKlucz" />
<add key="ImageUrl" value="https://NazwaKonta.blob.core.windows.net/" />
```

W moim przypadku gotowy kod do wklejenia wygląda następująco:

```
<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=ogl;AccountKey=2fij9698CQweJwW/J77MYSc+iD92XZo/FpjfX/
IyncbEaA2f" />
<add key="ImageUrl" value="https://ogl.blob.core.windows.net/" />
```

Kod wklejamy do sekcji <appSetings> w pliku Web.config w głównym projekcie OGL.

Sekcja po wklejeniu kodu wygląda następująco:

```
<appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=ogl;AccountKey=2fij9698CQweJwW/J77MYSc+iD92XZo/Fp" />
    <add key="ImageUrl" value="https://ogl.blob.core.windows.net/" />
</appSettings>
```

Od teraz możemy odczytywać w dowolny miejscu w aplikacji dane dostępowe do chmury znajdujące się w pliku Web.config. Złym i niebezpiecznym podejściem jest wklejanie tych danych w kodzie aplikacji a wiec w klasach aplikacji.

## Dodawanie klasy z modelem

Dodajemy mową klasę z modelem o nazwie Zdjecie:

```
public class Zdjecie
{
        public int Id { get; set; }
        public string Name { get; set; }

        public string UzytkownikId { get; set; }

        public Uzytkownik Uzytkownik { get; set; }
}
```

Tworzymy relację 1 do wielu z klasą Uzytkownik a więc musimy dodać w kalsie Uzytkownik następujący kod:

```
public virtual ICollection<Zdjecie> Zdjecia { get; private set; }
```

a następnie w konstruktorze klasy:

```
this.Zdjecia = new HashSet<Zdjecie>();
```

Ostatecznie kod klasy Uzytkownik wygląda następująco:

```
public class Uzytkownik : IdentityUser
{
        public Uzytkownik()
        {
            this.Ogloszenia = new HashSet<Ogloszenie>();
            this.Zdjecia = new HashSet<Zdjecie>();
        }
```

```
        // Klucz podstawowy odziedziczony po klasie IdentityUser

        // Dodajemy pola Imie i Nazwisko:
        public string Imie { get; set; }
        public string Nazwisko { get; set; }
        public int? Wiek { get; set; }

        #region dodatkowe pole notmapped

        [NotMapped]     // using System.ComponentModel.DataAnnotations.Schema;
        [Display(Name = "Pan/Pani:")]
        public string PelneNazwisko
        {
            get { return Imie + " " + Nazwisko; }
        }

        #endregion

        public virtual ICollection<Ogloszenie> Ogloszenia { get; private set; }
        public virtual ICollection<Zdjecie> Zdjecia { get; private set; }

        public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<Uzytkownik> manager)
        {
            var userIdentity = await manager.CreateIdentityAsync(this,
                                                    DefaultAuthenticationTypes.ApplicationCookie);
            // Add custom user claims here
            return userIdentity;
        }
}
```

# Aktualizacja DbContext

Dodajemy DbSety dla klasy Zdjecie w interfejsie (IOglContext) oraz klasie z contextem (OglContext):

```
public DbSet<Zdjecie> Zdjecia { get; set; }
```

Kod interfejsu IOglContext po aktualizacji:

```
public interface IOglContext
{
    DbSet<Kategoria> Kategorie { get; set; }
    DbSet<Ogloszenie> Ogloszenia { get; set; }
    DbSet<Uzytkownik> Uzytkownik { get; set; }
    DbSet<Ogloszenie_Kategoria> Ogloszenie_Kategoria { get; set; }
    DbSet<Zdjecie> Zdjecia { get; set; }

    int SaveChanges();
    Database Database { get; }

    DbEntityEntry Entry(object entity);
}
```

Kod klasy OglContex po aktualizacji:

```
public class OglContext : IdentityDbContext, IOglContext
{
        public OglContext()
            : base("DefaultConnection")
        {
        }

        public static OglContext Create()
        {
            return new OglContext();
        }
        public DbSet<Kategoria> Kategorie { get; set; }
        public DbSet<Ogloszenie> Ogloszenia { get; set; }
        public DbSet<Uzytkownik> Uzytkownik { get; set; }
        public DbSet<Ogloszenie_Kategoria> Ogloszenie_Kategoria { get; set; }
        public DbSet<Zdjecie> Zdjecia { get; set; }
```

```
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
            modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
            modelBuilder.Entity<Ogloszenie>().HasRequired(x => x.Uzytkownik)
                .WithMany(x => x.Ogloszenia)
                .HasForeignKey(x => x.UzytkownikId)
                .WillCascadeOnDelete(true);
        }
}
```

# Repozytorium dla zdjęć

W tym kroku stworzymy repozytorium dla klasy Zdjecie.

## Metody w repozytorium

- Dodaj zdjęcie,
- Usuń zdjęcie na podstawie nazwy(wartość guid),
- Zwróć listę zdjęć dla użytkownika,
- Metoda SaveChanges().

## Interfejs dla repozytorium

```
        public interface IZdjecieRepo
        {
                void AddImage(Zdjecie img);
                void DeleteImageByBlobName(string blobName);
                List<Zdjecie> GetAllImages(string userId);
                void SaveChanges();
        }
```

## Implementacja repozytorium

```
public class ZdjecieRepo : IZdjecieRepo
{

        private IOglContext _db;
        public ZdjecieRepo(IOglContext db)
        {
                _db = db;
        }

        public void AddImage(Zdjecie img)
        {
                _db.Zdjecia.Add(img);
        }
        public void DeleteImageByBlobName(string blobName)
        {
                Zdjecie img = _db.Zdjecia.Where(x => x.Name == blobName).FirstOrDefault();
                _db.Zdjecia.Remove(img);
        }
        public List<Zdjecie> GetAllImages(string userId)
        {
                return _db.Zdjecia.Where(x=>x.UzytkownikId == userId).ToList();
        }
        public void SaveChanges()
        {
                _db.SaveChanges();
        }
}
```

### Wstrzykiwanie implementacji przy pomocy IoC

Kolejnym krokiem jest ustawienie w pliku UnityCofig.cs jaka implementacja ma być wstrzykiwana w miejsce interfejsu IZdjecieRepo. Dodajemy linijkę:

```
container.RegisterType<IZdjecieRepo, ZdjecieRepo>(new PerRequestLifetimeManager());
```

Ostatecznie kod metody RegisterTypes wygląda następująco:

```
public static void RegisterTypes(IUnityContainer container)
{
        container.RegisterType<AccountController>(new InjectionConstructor());
        container.RegisterType<ManageController>(new InjectionConstructor());
        container.RegisterType<IOgloszenieRepo, OgloszenieRepo>(new PerRequestLifetimeManager());
        container.RegisterType<IKategoriaRepo, KategoriaRepo>(new PerRequestLifetimeManager());
        container.RegisterType<IOglContext, OglContext>(new PerRequestLifetimeManager());
        container.RegisterType<IZdjecieRepo, ZdjecieRepo>(new PerRequestLifetimeManager());
}
```

## Migracje dla klasy Zdjecie

Kolejnym krokiem jest dodanie nowej migracji w projekcie repozytorium ( komenda **add-migration nazwa_migracji**) a następnie aktualizacja bazy danych (komenda **update-database**).

Kod wygenerowanej migracji wygląda następująco:

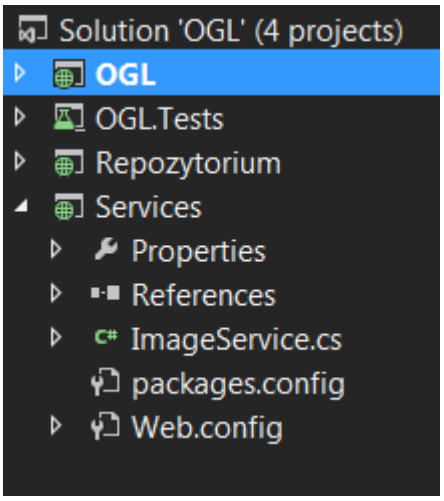
```
public partial class _zdjecie : DbMigration
{
        public override void Up()
        {
            CreateTable(
                "dbo.Zdjecie",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Name = c.String(),
                        UzytkownikId = c.String(maxLength: 128),
                    })
                .PrimaryKey(t => t.Id)
                .ForeignKey("dbo.AspNetUsers", t => t.UzytkownikId)
                .Index(t => t.UzytkownikId);

        }

        public override void Down()
        {
            DropForeignKey("dbo.Zdjecie", "UzytkownikId", "dbo.AspNetUsers");
            DropIndex("dbo.Zdjecie", new[] { "UzytkownikId" });
            DropTable("dbo.Zdjecie");
        }
}
```

## Nowy projekt Services

W tym kroku stworzymy dedykowany projekt do pracy z chmurą Azure. Cała logika związana z chmurą i obróbką zdjęć będzie się znajdować w tym projekcie.

Tworzymy nowy pusty (empty) projekt (web application) o nazwie Services w naszej solucji o nazwie OGL i dodajemy w nim plik o nazwie ImageService.cs.

Struktura solucji powinna wyglądać następująco:

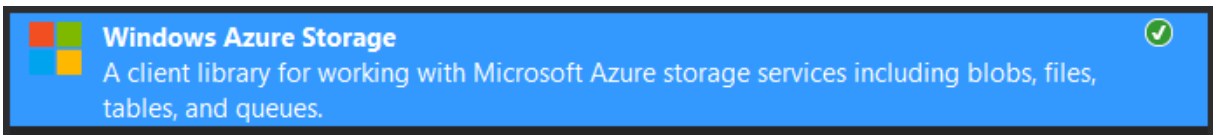



Na koniec dodajemy referencje do projektu Services w projekcie OGL. Klikamy PPM na References w projekcie OGL i wybieramy Add Reference, następnie zaznaczamy projekt Services.

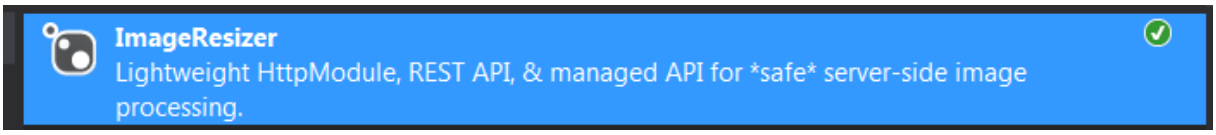
### Instalacja niezbędnych bibliotek w projekcie Services

Będziemy potrzebować biblioteki odpowiedzialne za kontakt z chmura Azure oraz bibliotekę do walidacji i zmiany rozmiaru zdjęć.

Poprzez NuGet instalujemy bibliotekę Windows Azure Storage:




oraz ImageResizer:




## Implementacja klas odpowiedzialnych za obróbkę i zapis zdjęć

W tym rozdziale zajmiemy się:

- sprawdzeniem faktycznego rozszerzenia plików,
- uniwersalnym rozwiązaniem generującym miniaturki,
- zapisem oraz usuwaniem zdjęć z chmury.

### Sprawdzenie rozszerzenia pliku

Na początek tworzymy typ wyliczeniowy enum z typami plików graficznych:

```
enum ImageExtension
{
        bmp,
        jpeg,
        gif,
        png,
        unknown
}
```

W kolejnym kroku tworzymy nową statyczną klasę o nazwie ImageOptimization a w niej publiczną metodę statyczną, która sprawdza rozszerzenie pliku:

```
public static ImageExtension GetImageExtension(byte[] bytes)
{
        var bmp = Encoding.ASCII.GetBytes("BM");     // BMP
        var gif = Encoding.ASCII.GetBytes("GIF");    // GIF
        var png = new byte[] { 137, 80, 78, 71 };    // PNG
        var jpeg = new byte[] { 255, 216, 255, 224 }; // jpeg
        var jpeg2 = new byte[] { 255, 216, 255, 225 }; // jpeg canon

        if (bmp.SequenceEqual(bytes.Take(bmp.Length)))
                return ImageExtension.bmp;

        if (gif.SequenceEqual(bytes.Take(gif.Length)))
                return ImageExtension.gif;

        if (png.SequenceEqual(bytes.Take(png.Length)))
                return ImageExtension.png;

        if (jpeg.SequenceEqual(bytes.Take(jpeg.Length)))
                return ImageExtension.jpeg;

        if (jpeg2.SequenceEqual(bytes.Take(jpeg2.Length)))
                return ImageExtension.jpeg;

        return ImageExtension.unknown;
}
```

Jeśli plik jest innego lub nieprawidłowego typu to zwracamy wartość unknown.

Następnie dodajemy do tej samej klasy dodajemy publiczną, statyczną metodę zwracająca wartość true, jeśli plik jest poprawny lub false gdy niepoprawny na podstawie wyniku działania metody GetImageExtension():

```
public static bool ValidateImage(byte[] image)
{
        if (GetImageExtension(image) != ImageExtension.unknown)
           return true;

        return false;
}
```

## Generowanie miniaturek i zmiana rozmiaru zdjęcia

W tym kroku stworzymy uniwersalne rozwiązanie, które pozwala dodawać nowe wielkości miniaturek i automatycznie na podstawie listy miniaturek będą dodawane i usuwane zdjęcia z chmury oraz metodę zmieniającą rozmiar zdjęcia.

## Uniwersalne rozwiązanie do generowania miniaturek

Na początek tworzymy strukturę, w której w konstruktorze możemy podawać nazwę miniaturek oraz wysokość i szerokość a następnie odczytywać te wartości poprzez właściwości.

Kod struktury wygląda następująco:

```
public struct ImageDimensions
{
        private readonly string _sizeName;
        private readonly int _width;
        private readonly int _height;

        public ImageDimensions(string sizeName, int width, int height)
        {
                this._sizeName = sizeName;
                this._width = width;
                this._height = height;
```

```
        }
        public string SizeName { get { return _sizeName; } }
        public int Width { get { return _width; } }
        public int Height { get { return _height; } }
}
```

Kolejnym krokiem jest stworzenie statycznej klasy zwracającej kolekcję (tylko do odczytu) miniaturek a wiec listę struktur ImageDimensions.

Kod klasy GalleryImages:

```
public static class GalleryImages
{
        public static readonly ReadOnlyCollection<ImageDimensions> GalleryDimensionsList = new
ReadOnlyCollection<ImageDimensions>
         (new[] {
                new ImageDimensions("large",900,600),
                new ImageDimensions("small",200,200)
        });
}
```

Jak widać posiadamy dwie wielkości zdjęć large oraz small. Wartości podane jako wysokość i szerokość są to maksymalne wartości. Zdjęcia będą skalowane z zachowaniem proporcji.

Aby dodać kolejny rozmiar miniaturek wystarczy dodać tylko jedną linie np.:

```
new ImageDimensions("medium",400,400)
```

### Metoda do zmiany rozmiaru zdjęć

Metoda zamienia rozmiar zdjęć przy pomocy wcześniej zainstalowanej biblioteki ImageResizer. Dodajemy tę metodę do klasy ImageOptimization.

```
public static byte[] OptimizeImageFromBytes(int imgWidth, int imgHeight, byte[] imgBytes)
{
        var settings = new ResizeSettings
        {
                MaxWidth = imgWidth,
                MaxHeight = imgHeight
        };

        MemoryStream ms = new MemoryStream();
        ImageBuilder.Current.Build(imgBytes, ms, settings);
        return ms.ToArray();
}
```

Jako parametry przyjmuje szerokość i wysokość, do jakich ma dostosować zdjęcie oraz zdjęcie w pełnych rozmiarach a zwraca pomniejszone zdjęcie w postaci tablicy bajtów.

### Pomocnicze klasy i metody

Kilka pomocniczych klas i metod, które ułatwią pracę ze zdjęciami.

### Klasa BlobImage

Kolejnym krokiem będzie stworzenie klasy BlobImage, która będzie reprezentować zdjęcie o określonym rozmiarze:

```
public class BlobImage
{
        //---  ContainerName/SizeName/ImageName
        //---  zdjecia/small/56.jpg
        public byte[] ImgBytes;
        public string SizeName;
        public string ImageName;
}
```

W komentarzu przedstawiona została struktura, w jakiej będą przechowywane pliki w chmurze.

## Klasa StreamHelpers i metoda ReadFully

Statyczna metoda zamienia ciąg wejściowy Stream na tablice bajtów a więc odczytuje cały plik i zapisuje w tablicy. Ułatwia to dalszą obróbkę zdjęcia, ponieważ nie pracujemy już na strumieniu danych, ale na całym już wczytanym zdjęciu.

```
public static class StreamHelpers
{
        public static byte[] ReadFully(this Stream input)
        {
                using (MemoryStream ms = new MemoryStream())
                {
                        input.CopyTo(ms);
                        return ms.ToArray();
                }
        }
}
```

## Klasa ImageUpload i dodawanie oraz usuwanie zdjęć wraz z miniaturkami

Publiczna klasa ImageUpload posiada 3 publiczne metody:

- **UploadImageAndReturnImageName()** - pobierająca jako parametr zdjęcie i zwracająca nazwę zdjęcia która została nadana podczas dodawania poprzez prywatną metodę CreateBlobName(),
- **DeleteImageByNameWithMiniatures()** - usuwająca zdjęcia wraz z miniaturkami na podstawie nazwy zdjęcia,
- **GetFullBlobName()** - zwracająca pełną nazwę zdjęcia wraz ze ścieżką - pozwala na generowanie indywidualnych ścieżek wirtualnych w jednym miejscu.

Klasa ImageUpload posiada również 4 metody prywatne niewidoczne dla innych klas:

- **GenerateImageMiniatures()** - zwracająca listę miniaturek zdjęć w rożnych wymiarach,
- **UploadMultipleImagesToBlob()** - zapisująca w chmurze wcześniej wygenerowaną listę miniaturek,
- **DeleteImageByName()** - usuwająca pojedyncze zdjęcie z chmury na podstawie nazwy,
- **CreateBlobName()** - metoda generująca nazwę bloba a więc w naszym wypadku zdjęcia.

## Dodawanie Zdjęć

## Metoda CreateBlobName()

Metoda służy do generowania nazw dla plików (BLOBów).

Ciało metody:

```
static string CreateBlobName()
{
        return System.Guid.NewGuid().ToString();
}
```

Metoda generuje zwykła wartość Guid i nie koniecznie musi istnieć, jednak powstała, aby w jednym miejscu zarządzać tym, w jaki sposób będą nadawane nazwy plikom. Gdybyśmy w wielu miejscach w aplikacji gdzie trzeba stworzyć nową nazwę dla pliku generowali nowego Guida to w przypadku stworzenia nowych zasad generowania nazw konieczna by była zmiana we wszystkich miejscach. Dzięki stworzeniu tej pomocniczej metody zmiana w jednym

miejscu powoduje zmianę dla wszystkich miejsc, w których trzeba wygenerować nową nazwę.

### Metoda GetFullBlobName()

Metoda zwraca całkowitą nazwę dla bloba.

Ciało metody:

```
public static string GetFullBlobName(string sizeName, string imageNameWithExtension)
{
        return sizeName + "/" + imageNameWithExtension;
}
```

W naszym przypadku przed nazwą zdjęcia jest jeszcze dopisana ścieżka wirtualna z nazwą rozmiaru. Ta metoda zwraca całkowitą ścieżkę do pliku. Jeśli zdecydujemy, aby w ścieżce znajdowało się jeszcze id użytkownika to wystarczy zmiana tylko w tej metodzie. Aktualnie metoda przyjmuje jako parametr nazwę rozmiaru oraz nazwę pliku razem z rozszerzeniem i zwraca całkowity adres dla pliku.

### Metoda UploadImageAndReturnImageName()

Publiczna metoda wywoływana w kontrolerze, aby zapisać zdjęcie w chmurze. Jeśli zapis się powiedzie to zwraca nazwę pliku a jeśli nie zwraca wartość null.

```
public string UploadImageAndReturnImageName(HttpPostedFileBase fileBase)
{
        byte[] image = fileBase.InputStream.ReadFully();
        if (!ImageOptimization.ValidateImage(image))
                return null;

        List<BlobImage> imagesToUpload = GenerateImageMiniatures(image);
        try
        {
                UploadMultipleImagesToBlob(imagesToUpload);
        }
        catch
        {
                return null;
        }
        //the same image name for all
        return imagesToUpload.First().ImageName;
}
```

Na początku wykorzystujemy metodę pomocniczą ReadFully() aby załadować cały plik, następnie walidujemy plik przy pomocy metody ValidateImage(). Jeśli plik jest nieprawidłowy to zwracamy wartość null, jeśli plik jest prawidłowy to wykorzystujemy prywatną metodę GenerateImageMiniatures(), aby stworzyć listę miniaturek dla zdjęcia. W kolejnym kroku w bloku try przy pomocy prywatnej metody UploadMultipleImagesToBlob() zapisujemy zdjęcie razem z miniaturkami w chmurze. Jeśli zapis się powiedzie to zwracamy nazwę zdjęcia (pierwszego, ponieważ wszystkie miniaturki mają tę samą nazwę, różnią sie tylko ścieżką). Jeśli zapis się nie powiedzie zwracamy wartość null.

### Metoda GenerateImageMiniatures()

Metoda generuje miniaturki na podstawie listy wymiarów GalleryDimensionsList z klasy GalleryImages.

Ciało metody:

```
 List<BlobImage> GenerateImageMiniatures(byte[] image)
{
        List<BlobImage> imagesToUpload = new List<BlobImage>();

        string blobName = CreateBlobName();
```

```
        foreach (var img in GalleryImages.GalleryDimensionsList)
        {
                byte[] imgBytes = ImageOptimization.OptimizeImageFromBytes(img.Width,
                                                                        img.Height, image);
                BlobImage blobImage = new BlobImage()
                {
                        ImgBytes = imgBytes,
                        SizeName = img.SizeName,
                        ImageName = blobName + "."
                                + ImageOptimization.GetImageExtension(imgBytes).ToString()
                };
                imagesToUpload.Add(blobImage);
        }
        return imagesToUpload;
}
```

Metoda przyjmuje zdjęcie w formacie tablicy bajtów i zwraca listę miniaturek. Na początek tworzymy pustą listę zdjęć, następnie przy pomocy metody CreateBlobName() generujemy nazwę dla zdjęcia, ponieważ nie można ufać oryginalnej nazwie zdjęcia. Następnie w pętli dla każdego wymiaru z kolekcji GalleryDimensionsList z wykorzystaniem metody OptimizeImageFromBytes() skalujemy zdjęcie do wybranych rozmiarów, zapisujemy w postaci BlobImage i dodajemy do listy miniaturek stworzonej na początku. Po zakończeniu generowania miniaturek zwracamy listę obiektów typu BlobImage.

### Metoda UploadMultipleImagesToBlob()

Metoda zapisuje w chmurze listę zdjęć.

Ciało metody:

```
void UploadMultipleImagesToBlob(List<BlobImage> images)
        {
                CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
                                CloudConfigurationManager.GetSetting("StorageConnectionString"));

                CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
                foreach (var img in images)
                {
                        CloudBlobContainer container = blobClient.GetContainerReference("zdjecia");
                        if (container.CreateIfNotExists())
                        {
                                var permissions = container.GetPermissions();
                                permissions.PublicAccess = BlobContainerPublicAccessType.Container;
                                container.SetPermissions(permissions);
                        }
                        string blobName = GetFullBlobName(img.SizeName, img.ImageName);
                        CloudBlockBlob blockBlob = container.GetBlockBlobReference(blobName);
                        blockBlob.Properties.ContentType = "image/png";

                        blockBlob.UploadFromByteArray(img.ImgBytes, 0, img.ImgBytes.Length);
                }
        }
```

Metoda przyjmuje jako parametr listę zdjęć a wiec obiektów BlobImage i nic nie zwraca. Na początek łączymy się z naszym kontem na Azure za pomocą danych zapisanych w pliku Web.config. Później w pętli łączymy się z naszym kontenerem o nazwie zdjęcia. Jeśli nie istnieje to go tworzymy przy pomocy metody CreateIfNotExist() z biblioteki Azure Storage i ustawiamy dostęp na PublicAccess. Następnie ciągle w pętli generujemy pełna nazwę BLOBa przy pomocy metody GetFullBlobName(), ustawiamy typ pliku na zdjęcie i zapisujemy w chmurze przy pomocy metody UploadFromByteArray().

## Usuwanie zdjęć

### Metoda DeleteImageByNameWithMiniatures()

Publiczna metoda służąca do usuwania zdjęć z chmury.

```
public void DeleteImageByNameWithMiniatures(string imageNameWithExtension)
{
        CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
                        CloudConfigurationManager.GetSetting("StorageConnectionString"));

        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

        CloudBlobContainer container = blobClient.GetContainerReference("zdjecia");
        foreach (var img in GalleryImages.GalleryDimensionsList)
        {
                DeleteImageByName(container, GetFullBlobName( img.SizeName,
                                                                imageNameWithExtension));
        }
}
```

Metoda przyjmuje jako parametr nazwę zdjęcia i nic nie zwraca. Tak jak przy dodawaniu łączymy się z chmura i w pętli dla wszystkich wymiarów miniaturek wywołujemy prywatną metodę DeleteImageByName() przekazując jako parametr kontener oraz pełną nazwę zdjęcia do usunięcia.

### Metoda DeleteImageByName()

Metoda służąca do usuwania pojedynczego zdjęcia z wybranego kontenera.

```
void DeleteImageByName(CloudBlobContainer container, string blobName)
{
        CloudBlockBlob blockBlob = container.GetBlockBlobReference(blobName);
        blockBlob.DeleteIfExists();
}
```

Metoda przyjmuje jako parametr kontener z którego ma zostać usunięte zdjęcie oraz nazwę zdjęcia do usunięcia. Zdjęcia sa usuwane przy pomocy metody DeleteIfExists() pochodzącej z biblioteki Azure Storage.

### Kod kompletnej klasy ImageUpload

```
public class ImageUpload
{
        public string UploadImageAndReturnImageName(HttpPostedFileBase fileBase)
        {
                byte[] image = fileBase.InputStream.ReadFully();
                if (!ImageOptimization.ValidateImage(image))
                        return null;

                List<BlobImage> imagesToUpload = GenerateImageMiniatures(image);
                try
                {
                        UploadMultipleImagesToBlob(imagesToUpload);
                }
                catch { }
                //the same image name for all
                return imagesToUpload.First().ImageName;
        }

        List<BlobImage> GenerateImageMiniatures(byte[] image)
        {
                List<BlobImage> imagesToUpload = new List<BlobImage>();

                string blobName = CreateBlobName();

                foreach (var img in GalleryImages.GalleryDimensionsList)
```

```
                {
                        byte[] imgBytes = ImageOptimization.OptimizeImageFromBytes(img.Width,
                                                                                   img.Height, image);
                        BlobImage blobImage = new BlobImage()
                        {
                                ImgBytes = imgBytes,
                                SizeName = img.SizeName,
                                ImageName = blobName + "."
                                        + ImageOptimization.GetImageExtension(imgBytes).ToString()
                        };
                        imagesToUpload.Add(blobImage);
                }
                return imagesToUpload;
        }

        void UploadMultipleImagesToBlob(List<BlobImage> images)
        {
                CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
                                CloudConfigurationManager.GetSetting("StorageConnectionString"));

                CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
                foreach (var img in images)
                {
                        CloudBlobContainer container = blobClient.GetContainerReference("zdjecia")
                        if (container.CreateIfNotExists())
                        {
                                var permissions = container.GetPermissions();
                                permissions.PublicAccess = BlobContainerPublicAccessType.Container;
                                container.SetPermissions(permissions);
                        }
                        string blobName = GetFullBlobName(img.SizeName, img.ImageName);
                        CloudBlockBlob blockBlob = container.GetBlockBlobReference(blobName);
                        blockBlob.Properties.ContentType = "image/png";

                        blockBlob.UploadFromByteArray(img.ImgBytes, 0, img.ImgBytes.Length);
                }
        }


        public void DeleteImageByNameWithMiniatures(string imageNameWithExtension)
        {
                CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
                        CloudConfigurationManager.GetSetting("StorageConnectionString"));

                CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

                CloudBlobContainer container = blobClient.GetContainerReference("zdjecia");
                foreach (var img in GalleryImages.GalleryDimensionsList)
                {
                        DeleteImageByName(container, GetFullBlobName( img.SizeName,
                                                                      imageNameWithExtension));
                }
        }

        void DeleteImageByName(CloudBlobContainer container, string blobName)
        {
                CloudBlockBlob blockBlob = container.GetBlockBlobReference(blobName);
                blockBlob.DeleteIfExists();
        }


        public static string GetFullBlobName(string sizeName, string imageNameWithExtension)
        {
                return sizeName + "/" + imageNameWithExtension;
        }

        static string CreateBlobName()
        {
                return System.Guid.NewGuid().ToString();
        }
}
```

# Kontroler do galerii zdjęć

Tworzymy kontroler o nazwie Galeria, który będzie posiadał 3 metody:

- **Lista()** - zwraca widok z listą dodanych zdjęć,
- **UploadImage()** - metoda POST odpowiedzialna za dodawanie zdjęć do galerii,
- **DeleteImage()** - metoda POST odpowiedzialna za usuwanie zdjęć z galerii.

## Konstruktor i wstrzykiwanie repozytorium

```
private IZdjecieRepo _zdjecieRepo;
public GaleriaController(IZdjecieRepo zdjecieRepo)
{
        _zdjecieRepo = zdjecieRepo;
}
```

## Metoda Lista()

```
public ActionResult Lista()
{
        List<Zdjecie> zdjecia = _zdjecieRepo.GetAllImages(User.Identity.GetUserId());
        return View(zdjecia);
}
```

## Metoda UploadImage()

Metoda POST służąca do zapisywania zdjęć w chmurze.

Kod metody:

```
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult UploadImage(HttpPostedFileBase fileBase)
        {
                if (fileBase != null && fileBase.ContentLength > 0)
                {
                        try
                        {
                                ImageUpload imageUpload = new ImageUpload();
                                string nameWithExtension =
                                        imageUpload.UploadImageAndReturnImageName(fileBase);
                                if(nameWithExtension == null)
                                        return RedirectToAction("Lista", "Galeria");
                                try
                                {
                                        Zdjecie img = new Zdjecie()
                                        {
                                                UzytkownikId = User.Identity.GetUserId(),
                                                Name = nameWithExtension
                                        };
                                        _zdjecieRepo.AddImage(img);
                                        _zdjecieRepo.SaveChanges();
                                }
                                catch
                                {
                                      imageUpload.DeleteImageByNameWithMiniatures(nameWithExtension);
                                }
                        }
                        catch {}
                }

                return RedirectToAction("Lista", "Galeria");
        }
```

Jako parametr przyjmuje strumień danych pliku. Na początku sprawdzamy czy plik nie jest pusty, następnie w bloku try przy pomocy metody UploadImageAndReturnImageName() zapisujemy zdjęcie wraz z miniaturkami w chmurze. Jeśli zapis się powiedzie to zostanie zwrócona nazwa zdjęcia i w kolejnym bloku try przy pomocy metody AddImage() z repozytorium zapisujemy nazwę zdjęcia do bazy danych. Gdy zapis do bazy się nie powiedzie

w bloku catch wywołana zostaje metoda DeleteImageByNameWithMiniatures(), która usuwa zdjęcie wraz z miniaturkami z chmury. Gdyby nie została wywołana ta metoda w bazie danych nie było by informacji o zdjęciu a w chmurze zostałyby zapisane zdjęcia, do których nie było by dostępu. Za każdy gigabajt trzeba płacić co miesiąc, dlatego nie było by to dobre rozwiązanie.

### Metoda DeleteImage()

Metoda służy do usuwania zdjęć z galerii.

Kod metody:

```
[HttpPost]
[ValidateAntiForgeryToken]
public bool DeleteImage(string blobName)
{
        if (blobName == null)
        {
                return false;
        }
        try
        {
                ImageUpload imageUpload = new ImageUpload();
                imageUpload.DeleteImageByNameWithMiniatures(blobName);
                try
                {
                        _zdjecieRepo.DeleteImageByBlobName(blobName);
                        _zdjecieRepo.SaveChanges();
                        return true;
                }
                catch
                {
                        return false;
                }
        }
        catch
        {
                return false;
        }
}
```

Metoda przyjmuje jako parametr nazwę zdjęcia do usunięcia i zwraca wartość true jeśli usuwanie się powiedzie lub false gdy usuwanie sie nie powiedzie. Jeśli nazwa zdjęcia nie jest pusta zostaje wywołana metoda DeleteImageByNameWithMiniatures() usuwająca zdjęcie z chmury, gdy usuwanie się powiedzie w bloku try wywoływana jest metoda DeleteImageByBlobName() z repozytorium usuwająca zdjęcie z bazy danych. Jeśli wszystko się powiedzie zwracana jest wartość true i zdjęcie znika z galerii w przeciwnym wypadku zwracana jest wartość false i zdjęcie nie ostaje usunięte z galerii.

### Kompletny kod kontrolera Galeria

Kompletny kod kontrolera wygląda następująco:

```
public class GaleriaController : Controller
{
        private IZdjecieRepo _zdjecieRepo;
        public GaleriaController(IZdjecieRepo zdjecieRepo)
        {
                _zdjecieRepo = zdjecieRepo;
        }

        public ActionResult Lista()
        {
                List<Zdjecie> zdjecia = _zdjecieRepo.GetAllImages(User.Identity.GetUserId());
                return View(zdjecia);
        }

         [HttpPost]
```

```
         [ValidateAntiForgeryToken]
        public ActionResult UploadImage(HttpPostedFileBase fileBase)
        {
                if (fileBase != null && fileBase.ContentLength > 0)
                {
                        try
                        {
                                ImageUpload imageUpload = new ImageUpload();
                                string nameWithExtension =
imageUpload.UploadImageAndReturnImageName(fileBase);
                                if(nameWithExtension == null)
                                        return RedirectToAction("Lista", "Galeria");
                                try
                                {
                                        Zdjecie img = new Zdjecie()
                                        {
                                                UzytkownikId = User.Identity.GetUserId(),
                                                Name = nameWithExtension
                                        };
                                        _zdjecieRepo.AddImage(img);
                                        _zdjecieRepo.SaveChanges();
                                }
                                catch
                                {
                                      imageUpload.DeleteImageByNameWithMiniatures(nameWithExtension);
                                }
                        }
                        catch {}
                }

                return RedirectToAction("Lista", "Galeria");
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public bool DeleteImage(string blobName)
        {
                if (blobName == null)
                {
                        return false;
                }
                try
                {
                        ImageUpload imageUpload = new ImageUpload();
                        imageUpload.DeleteImageByNameWithMiniatures(blobName);
                        try
                        {
                                _zdjecieRepo.DeleteImageByBlobName(blobName);
                                _zdjecieRepo.SaveChanges();
                                return true;
                        }
                        catch
                        {
                                return false;
                        }
                }
                catch
                {
                        return false;
                }
        }
}
```

# Widok dla galerii zdjęć

Widok będzie się składał z kilku części. Formularza do dodawania zdjęć razem z podglądem dodawanego zdjęcia oraz formularzy do usuwania zdjęć dla każdego zdjęcia z galerii. Dodajemy widok dla metody Lista().

## Kod HTML dla widoku

```
@model IEnumerable<Repozytorium.Models.Zdjecie>
@using Services;
@using Microsoft.AspNet.Identity;

@{
    ViewBag.Title = "Lista zdjęć";
}

<h2>Lista zdjeć</h2>

<div class="row">
        <div class="col-xs-12">
                <div style="background-color:#e5e5e5; float:left; border-radius: 4px; margin:5px;
display: block; font-size:10px; padding:10px; width:220px; max-height:350px; text-align:center;">
                        @using (Html.BeginForm("UploadImage", "Galeria", FormMethod.Post,
        new { enctype = "multipart/form-data" }))
                        {
                                <input type="hidden" value="@User.Identity.GetUserId()" id="userId"/>
                                @Html.AntiForgeryToken()
                                <fieldset>
                                        <h4>Dodaj zdjęcie</h4>
                                        <input type="file" name="fileBase" id="imgInp"
onchange="checkFile(this)" />

                                        <img id="preview_image" src="#" alt="twoje zdjęcie"
class="img-thumbnail" style="max-width:200px; max-height:200px; display:none; margin-bottom:5px;" />
                                        <p>
                                                <input type="submit" class="btn btn-sm btn-success"
id="UploadImage" style="display:none" value="Dodaj to zdjęcie do albumu" />
                                        </p>
                                </fieldset>
                        }
                </div>

                @using (Html.BeginForm())
                {
                        @Html.AntiForgeryToken()
                        foreach (var modelItem in Model)
                        {
                                <div style="float:left; padding:5px; max-width:200px; height:240px;
text-align:center; background-color:ActiveBorder" id="@modelItem.Name">
                                        <a
href="@(System.Configuration.ConfigurationManager.AppSettings["ImageUrl"]+"zdjecia/"
                  + ImageUpload.GetFullBlobName("large", modelItem.Name))">

                                                <img
src="@(System.Configuration.ConfigurationManager.AppSettings["ImageUrl"]+"zdjecia/"
                                        + ImageUpload.GetFullBlobName("small", modelItem.Name))"
                                                        alt="" class="img-thumbnail" />
                                        </a>
                                        <br />
                                        <a class="btn btn-xs btn-danger usun-zdjecie2 lazy"
onclick="if(confirm('CZy na pewno usunąć to zdjęcie?')){UsunZdjecie('@(modelItem.Name)');}">Usuń</a>
                                </div>
                        }
                }
        </div>
</div>
```

W pierwszym formularzu służącym do dodawania zdjęć, po każdej zmianie ścieżki do pliku, który ma zostać dodany, wywoływana jest metoda checkFile(), służąca do walidacji pliku po stronie klienta a więc przeglądarki. Podgląd zdjęcia i przycisk Dodaj zdjęcie są ukryte aż do momentu, gdy zostanie wybrane z dysku zdjęcie do dodania.

Zdjęcia z modelu są wyświetlane w pętli i pod każdym zdjęciem jest dodawany przycisk służący do usunięcia zdjęcia. Po kliknięciu przycisku Usuń wywoływana jest w jQuery funkcja UsunZdjecie() z nazwą zdjęcia podawaną jako parametr.

## Kod JS i jQuery dla widoku

Kod JS i jQuery jest tutaj bardzo ważny, ponieważ wywołuje metody POST z kontrolera i pozwala usuwać zdjęcia.

## Funkcja ReadURL()

Funkcja służy do wczytywania zdjęcia i wyświetlenia miniaturki podczas dodawania.

```
function readURL(input) {

		if (input.files && input.files[0]) {
			var reader = new FileReader();

			reader.onload = function (e) {
				$('#preview_image').attr('src', e.target.result);
			}

			reader.readAsDataURL(input.files[0]);
		}
	}
```

## Funkcja checkFile()

Funkcja checkFile() służy do walidacji pliku po stronie klienta. W naszym rozwiązaniu walidacja odbywa się zarówno po stronie klienta jak i po stronie serwera.

```
function checkFile(fieldObj) {
		var FileName = fieldObj.value;
		var FileExt = FileName.substr(FileName.lastIndexOf('.') + 1);
		var FileSize = fieldObj.files[0].size;
		var FileSizeMB = 4;

		if ((FileExt != "jpg" && FileExt != "gif" && FileExt != "png" && FileExt != "jpeg" &&
                                                FileExt != "bmp") || FileSize > 1048576 * 4) {
			var error = "Typ pliku : " + FileExt + "\n\n";
			error += "Rozmiar pliku: " + FileSizeMB + " MB \n\n";
			error += "Akceptowane typy plików: jpg, png, gif, bmp. Maksymalny rozmiar 4
                                                                                MB.\n\n";

			alert(error);
			return false;
		}
		return true;
	}
```

Funkcja jako parametr przyjmuje plik a zwraca wartość true jeśli plik jest prawidłowy lub wartość false gdy plik nie spełnia wymogów. Odczytujemy z pliku nazwę, rozszerzenie i rozmiar, następnie sprawdzamy czy rozszerzenie się zgadza i czy rozmiar nie jest większy od maksymalnego rozmiaru pliku, na jaki zezwalamy. W aplikacji ASP.NET MVC domyślnie maksymalny rozmiar pliku wynosi 4MB, dlatego do walidacji pliku po stronie klienta również ustawiliśmy 4MB. Gdy plik jest za duży lub posiada nieprawidłowe rozszerzenie to konstruowany jest komunikat błędu i wyświetlane okienko z powiadomieniem. Jeśli natomiast plik jest prawidłowy to zwracana jest wartość true.

## Funkcja UsunZdjecie()

Funkcja wykonuje żądanie POST usuwające zdjęcie z galerii.

```
function UsunZdjecie(nazwa) {
		var token = $('input[name="__RequestVerificationToken"]').val();
		$.ajax({
			url: "/Galeria/DeleteImage/",
			dataType: 'text',
			type: 'POST',
			data: {
				__RequestVerificationToken: token,
				blobName: nazwa
```

```
                        },
                        success: function (result) {
                                if (result == "True") {
                                        $(document.getElementById(nazwa)).closest('div').remove();
                                }
                        }
                });
        }
```

Funkcja przyjmuje jako parametr nazwę zdjęcia i przy pomocy żądania AJAX wywołuje metodę DeleteImage() z kontrolera Galeria. Jako parametry przekazujemy AntiForgeryToken oraz nazwę zdjęcia. Każda metoda POST powinna posiadać token zabezpieczający przed atakami CSRF(szczegóły w książce w rozdziale o bezpieczeństwie). Typ danych przekazywanych jako parametr do metody z kontrolera ustawiony został na typ tekstowy. Jeśli kontroler zwróci wartość true to przy pomocy jQuery usuwamy diva ze zdjęciem bez odświeżania strony.

## Funkcja JQuery Lazy()

Funkcja Lazy() służy do tego, aby zdjęcia były ładowane dopiero, gdy są widoczne na ekranie a więc po przewinięciu strony na dół. W standardowym podejściu wszystkie zdjęcia są ładowane podczas wczytywania strony i jeśli ktoś nie przewinie strony w dół to są wczytywane na darmo. Funkcja Lazy pozwala zmniejszyć rachunek za chmurę Azure, ponieważ wczytywana będzie mniejsza ilość zdjęć.

Kod funkcji:

```
jQuery(document).ready(function () {
                jQuery("img.lazy").lazy();
        });
```

Każde znacznik <img> posiadający klasę lazy zostanie leniwie ładowany.

## Funkcja wyłapująca zmiany stanu w polu input

Funkcja zostaje wywołana, gdy wybierzemy ścieżkę do zdjęcia, które chcemy dodać do galerii.

Kod funkcji:

```
$("#imgInp").change(function () {
                if (checkFile(this)) {
                        readURL(this);
                        $('#UploadImage').css('display', 'block');
                        $('#preview_image').css('display', 'block');
                }
                else {
                        $('#preview_image').attr('src', '');
                        $('#UploadImage').css('display', 'none');
                        $('#preview_image').css('display', 'none');
                }
        });
```

Na początek wywołujemy funkcję sprawdzającą zdjęcie checkFile(), jeśli plik jest prawidłowy to wywoływana zostaje funkcja readURL() wczytująca zdjęcie do podglądu. Następnie wyświetlamy przycisk dodaj do galerii oraz podgląd zdjęcia. Jeśli plik jest nieprawidłowy to resetujemy wartość ścieżki do pliku oraz ukrywamy przycisk i podgląd zdjęcia.

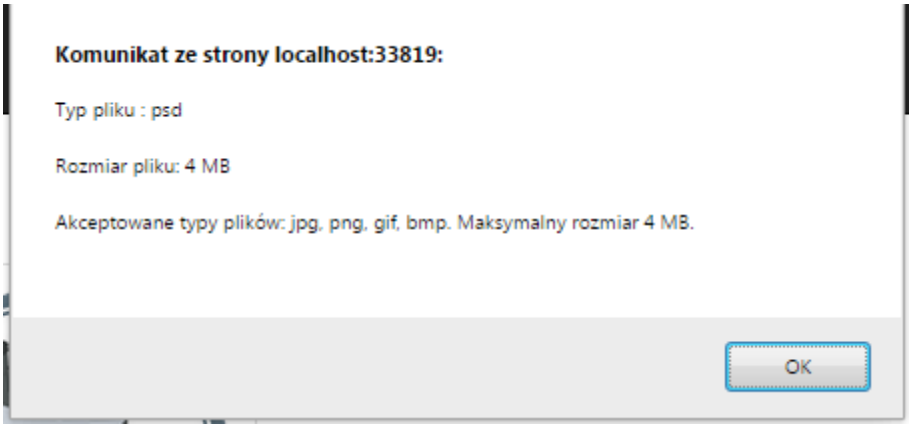
## Kompletny kod JS dla widoku

```
@section Scripts{
        <script type="text/javascript" src="/Content/js/jquery.lazy.min.js"></script>
```

```
<script>

        jQuery(document).ready(function () {
                jQuery("img.lazy").lazy();
        });

        function readURL(input) {

                if (input.files && input.files[0]) {
                        var reader = new FileReader();

                        reader.onload = function (e) {
                                $('#preview_image').attr('src', e.target.result);
                        }

                        reader.readAsDataURL(input.files[0]);
                }
        }

        $("#imgInp").change(function () {
                if (checkFile(this)) {
                        readURL(this);
                        $('#UploadImage').css('display', 'block');
                        $('#preview_image').css('display', 'block');
                }
                else {
                        $('#preview_image').attr('src', '');
                        $('#UploadImage').css('display', 'none');
                        $('#preview_image').css('display', 'none');
                }
        });

        function checkFile(fieldObj) {
                var FileName = fieldObj.value;
                var FileExt = FileName.substr(FileName.lastIndexOf('.') + 1);
                var FileSize = fieldObj.files[0].size;
                var FileSizeMB = 4;

                if ((FileExt != "jpg" && FileExt != "gif" && FileExt != "png" && FileExt != "jpeg" &&
                                                        FileExt != "bmp") || FileSize > 1048576 * 4) {
                        var error = "Typ pliku : " + FileExt + "\n\n";
                        error += "Rozmiar pliku: " + FileSizeMB + " MB \n\n";
                        error += "Akceptowane typy plików: jpg, png, gif, bmp. Maksymalny rozmiar 4
                                                                                        MB.\n\n";

                        alert(error);
                        return false;
                }
                return true;
        }

        function UsunZdjecie(nazwa) {
                var token = $('input[name="__RequestVerificationToken"]').val();
                $.ajax({
                        url: "/Galeria/DeleteImage/",
                        dataType: 'text',
                        type: 'POST',
                        data: {
                                __RequestVerificationToken: token,
                                blobName: nazwa
                        },
                        success: function (result) {
                                if (result == "True") {
                                        $(document.getElementById(nazwa)).closest('div').remove();
                                }
                        }
                });
        }
</script>

}
```

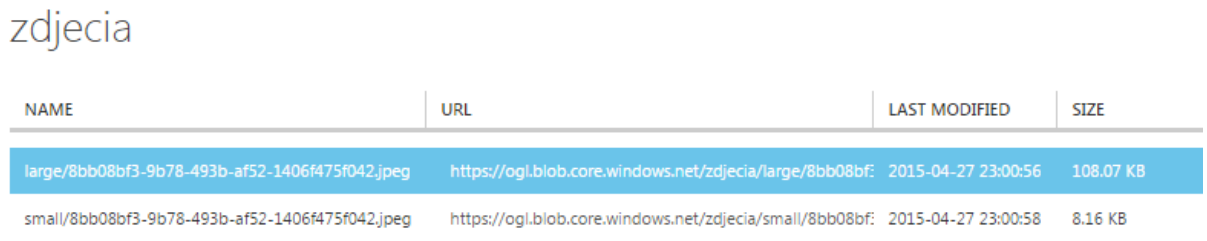
### Testowanie galerii

Jeśli chcemy dodać zdjęcie o wielkości powyżej 4MB lub z nieprawidłowym rozszerzeniem to pojawia się nam komunikat:

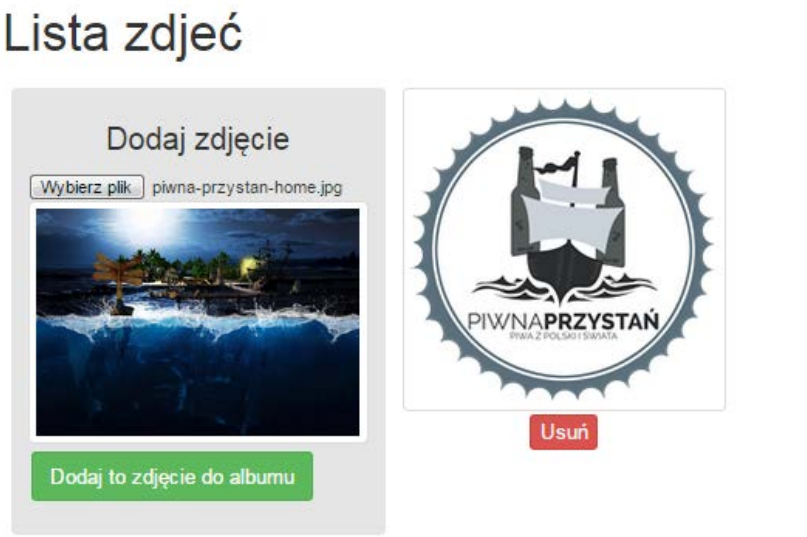



Lista zdjęć po dodaniu jednego zdjęcia:


Po dodaniu jednego zdjęcia do chmury zostały dodane 2 zdjęcia (miniaturki). Zawartość kontenera w panelu administracyjnym Azure po dodaniu 1 zdjęcia:




Lista zdjęć po wybraniu kolejnego zdjęcia do dodania. Kliknięcie przycisku "Dodaj to zdjęcie do albumu" spowoduje zapisanie zdjęcia i miniaturek w chmurze oraz dodanie zdjęcia do albumu.




# Rozdział 3. Bezpieczna walidacja pól wejściowych z kodem HTML z edytora WYSIWYG

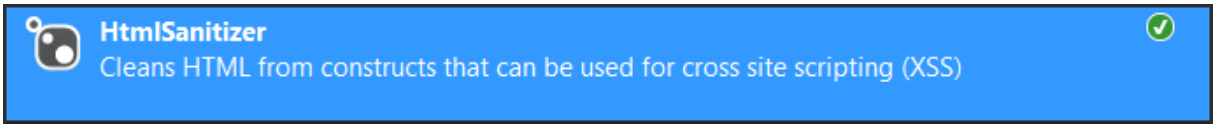
## Teoria

Aby umożliwić użytkownikowi wklejenie własnego kodu HTML na stronę lub skorzystać z edytora WYSIWYG konieczne jest zapisanie kodu HTML w bazie danych. W ASP.NET MVC domyślnie jest to zabronione i wklejenie kodu html do pola typu string wywołuje błąd walidacji. Aby zezwolić na zapis kodu HTML w polu tekstowym należy użyć atrybutu [AllowHtml] w polu klasy z modelem. Jednak zwykłe zezwolenie na wklejenie kodu HTML lub JS w polu tekstowym niesie ze sobą bardzo duże niebezpieczeństwo, dlatego trzeba użyć specjalnej biblioteki, która przeskanuje kod w celu wykrycia potencjalnych niebezpieczeństw. Zwykle zabezpieczenie polega na wybraniu znaczników HTML, które są dozwolone tzw. "whitelist", wszystkie pozostałe zostają usunięte z kodu.

# Praktyka

## Biblioteka do walidacji

Na początek instalujemy w projekcie OGL poprzez NuGet bibliotekę HtmlSanitizer, która posłuży nam do walidacji kodu Html.




## Klasy z modelem

Na początek stworzymy prostą klasę z modelem o nazwie Edytor z polem typu string, do którego będziemy zapisywać kod HTML

## Kod klasy Edytor

```
public class Edytor
{
        [Key, ForeignKey("Uzytkownik")]
        public string Id { get; set; }

        [AllowHtml]
        public string Tresc { get; set; }

        public Uzytkownik Uzytkownik { get; set; }
}
```

Tabela w relacji 1:1 z tabelą użytkownik. Przy polu Tresc został dodany atrybut [AllowHtml] pozwalający na zapis kodu, HTML w polu typu string. Pole Id jest typu string, ponieważ klucz podstawowy w tabeli Uzytkownik jest Guidem. W relacji 1 do 1 Id obu klas musi być identyczne.

## Klasa Uzytkownik

W klasie Uzytkownik dodajemy:

```
public virtual Edytor Edytor { get; set; }
```

## Aktualizacja DbContext

Dodajemy do klasy i interfejsu Contextu właściwość:

```
public DbSet<Edytor> Edytor { get; set; }
```

## Migracje

Gdy mamy stworzoną nową klasę z modelem oraz dodane właściwości do contextu możemy stworzyć migrację a następnie zaktualizować strukturę bazy danych.

Kod wygenerowanej migracji:

```
public partial class edytor : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Edytor",
            c => new
                {
                    Id = c.String(nullable: false, maxLength: 128),
                    Tresc = c.String(),
                })
            .PrimaryKey(t => t.Id)
```

```
                .ForeignKey("dbo.AspNetUsers", t => t.Id)
                .Index(t => t.Id);

        }

        public override void Down()
        {
            DropForeignKey("dbo.Edytor", "Id", "dbo.AspNetUsers");
            DropIndex("dbo.Edytor", new[] { "Id" });
            DropTable("dbo.Edytor");
        }
    }
```

## Kontroler

Dodajemy kontroler o nazwie Edytor oznaczamy atrybutem [Authorize], aby dostęp do metod był możliwy tylko dla zalogowanych użytkowników. W tym przypadku nie tworzymy repozytorium tylko tworzymy obiekt contextu w klasie kontrolera. Nie jest to dobre rozwiązanie, ponieważ context jest tworzony w kontrolerze a niewstrzykiwany za pomocą IoC. Dodatkowo zapytanie LINQ znajduje się w klasie kontrolera.

## Kod metody GET

```
public ActionResult EdytujTresc()
{
        using (OglContext context = new OglContext())
        {
                string userId = User.Identity.GetUserId();
                Edytor edytor = context.Edytor.Where(x => x.Id == userId).FirstOrDefault();
                return View(edytor);
        }
}
```

W tej metodzie tworzymy obiekt contextu, przy pomocy bloku using. Następnie pobieramy z bazy dane z tabeli Edytor dla aktualnie zalogowanego użytkownika. Jeśli nie ma jeszcze w bazie zapisanych danych dla tego użytkownika to metoda FirstOrDefault zwróci wartość null. Na końcu zwracamy widok i do widoku przekazujemy obiekt edytor.

## Kod metody POST

```
[HttpPost]
[ValidateAntiForgeryToken]
[ValidateInput(false)]
public ActionResult EdytujTresc([Bind(Include="Id,Tresc")]Edytor edytor)
{
        if (ModelState.IsValid)
        {
                var userId = User.Identity.GetUserId();
                using(OglContext context = new OglContext())
                {
                        var sanitizer = new HtmlSanitizer();
                        var trescSprawdzona = sanitizer.Sanitize(edytor.Tresc);
                        edytor.Tresc = trescSprawdzona;
                        edytor.Id = userId;
                        if(context.Edytor.Where(x=>x.Id==userId).Any())
                        {
                                context.Entry(edytor).State =
                                        System.Data.Entity.EntityState.Modified;
                        }
                        else
                        {
                                context.Edytor.Add(edytor);
                        }
                        context.SaveChanges();
                        return RedirectToAction("EdytujTresc", new { edytowanoDodanoInfo = 2 });
                }
        }
        ViewBag.Warning = "Coś poszło nie tak - spróbuj ponownie";
        return View(edytor);
}
```

W metodzie post używamy 3 atrybutów [HttpPost] określającego, że jest to metoda POST, [ValidateAntiForgeryToken] - określającego ze wymagany jest token bezpieczeństwa oraz [ValidateInput(false)] - który określa że nie należy walidować pól wejściowych i użytkownik przejmuje odpowiedzialność za zabezpieczenie aplikacji.

Na początku przy pomocy using tworzymy obiekt contextu podobnie jak to miało miejsce w metodzie GET. Tutaj również zamiast tego powinno zostać użyte repozytorium i wstrzykiwanie jego implementacji w konstruktorze, ale to już we własnym zakresie należy poprawić, jako ćwiczenie.

Aby sprawdzić czy dane nie zawierają szkodliwego kodu wykorzystujemy wcześniej zainstalowaną bibliotekę HtmlSanitizer. Tworzymy nowy obiekt HtmlSanitizer i przy pomocy metody Sanitize() sprawdzamy czy treść przesłana przez użytkownika nie zawiera szkodliwego kodu. Następnie jako id podajemy Id użytkownika ponieważ jest to relacja 1 do 1 dlatego id posiada taką samą wartość. W kolejnej linijce przy pomocy LINQ i metody Any() sprawdzamy czy już wcześniej były zapisane jakieś dane i istnieje wpis w bazie danych z danym id użytkownika. Jeśli tak to informujemy context ze obiekt został zmieniony i należy go zapisać do bazy, jeśli nie było takiego obiektu w bazie to dodajemy nowy obiekt do contextu. Następnie wywołujemy metodę SaveChanges() na obiekcie contextu, która zapisuje zmiany w bazie danych.

W naszym rozwiązaniu brakuje powiadomień, gdy wystąpi błąd podczas zapisu lub coś innego pójdzie nie tak jak powinno. Było to już opisywane w książce, dlatego nie będzie kolejny raz opisywane.

### Kompletny kod kontrolera:

```
[Authorize]
public class EdytorController : Controller
{
    public ActionResult EdytujTresc()
    {
            using (OglContext context = new OglContext())
            {
                    string userId = User.Identity.GetUserId();
                    Edytor edytor = context.Edytor.Where(x => x.Id == userId).FirstOrDefault();
                    return View(edytor);
            }
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    [ValidateInput(false)]
    public ActionResult EdytujTresc([Bind(Include="Id,Tresc")]Edytor edytor)
    {
            if (ModelState.IsValid)
            {
                    var userId = User.Identity.GetUserId();
                    using(OglContext context = new OglContext())
                    {
                            var sanitizer = new HtmlSanitizer();
                            var trescSprawdzona = sanitizer.Sanitize(edytor.Tresc);
                            edytor.Tresc = trescSprawdzona;
                            edytor.Id = userId;
                            if(context.Edytor.Where(x=>x.Id==userId).Any())
                            {
                                    context.Entry(edytor).State =
                                                    System.Data.Entity.EntityState.Modified;
                            }
                            else
                            {
                                    context.Edytor.Add(edytor);
                            }
```

```
                                context.SaveChanges();
                                return RedirectToAction("EdytujTresc", new {
                                                                        edytowanoDodanoInfo = 2 });
                        }
                }
                ViewBag.Warning = "Coś poszło nie tak - spróbuj ponownie";
                return View(edytor);
        }

    }
```

## Widok

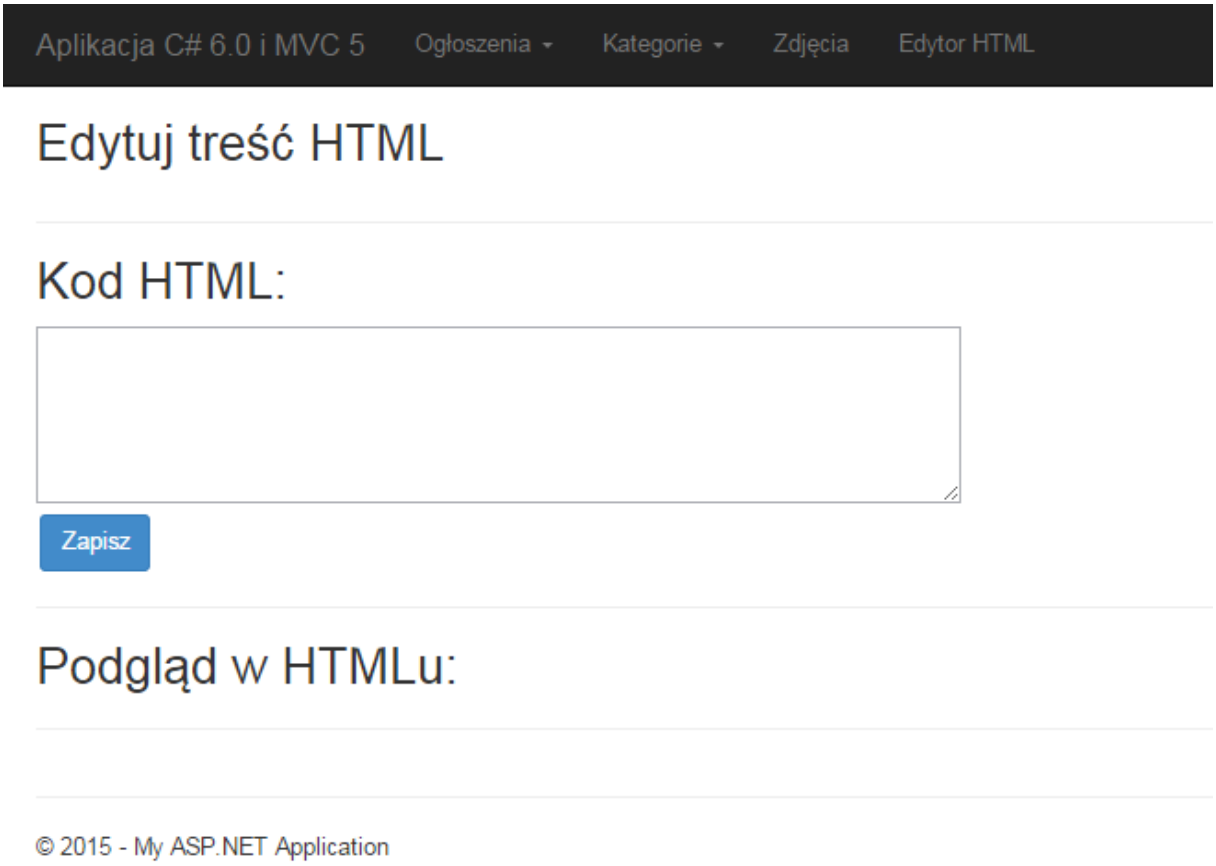
Kod widoku:

```
@model Repozytorium.Models.Edytor
@{
        ViewBag.Tytul = "Edytuj treść HTML";
}

<h2>Edytuj treść HTML</h2>

<div class="row" style="max-width:800px">
        <div class="col-xs-12">
                @using (Html.BeginForm())
                {
                        @Html.AntiForgeryToken()
                        <input type="hidden" name="Id" value="Model.Id"/>
                        <hr />
                        <h2>Kod HTML:</h2>
                        @Html.TextAreaFor(model => model.Tresc, 5, 80, new { style = "max-width:90%;"
})

                        <div class="form-actions align-right clearfix">
                                <button type="submit" class="btn btn-primary pull-left">
                                        <i class="ace-icon fa fa-check bigger-110"></i>
                                        Zapisz
                                </button>
                        </div>
                }

                @if (Model != null)
                {
                        <hr />
                        <h2>Podgląd w HTMLu:</h2>
                        <hr />
                        @Html.Raw(Model.Tresc);
                }
        </div>
</div>
```

W widoku posiadamy jeden formularz, w którym generowany jest AntiForgeryToken, oraz posiadamy jedno ukryte pole, w którym przekazujemy Id. Następnie wykorzystujemy pole tekstowe (TextArea) posiadające 88 kolumn i 5 wierszy, w którym będziemy wpisywać kod html. Kolejnym elementem jest przycisk Zapisz.

Na samym końcu wyświetlamy wcześniej zapisany kod w polu treśc jako kod HTML na stronie. Aby wyświetlić kod jako HTML a nie zwykły napis typu tekstowego należy użyć metody Html.Raw().

Wygląd podstrony przed dodaniem kodu HTML:




Wklejamy w okno następujący kod:

```
<p style="color: red">Polecamy:</p>
<a href="http://aspnetmvc.pl/" target="_blank">
     <img style="max-width: 300px" src="http://aspnetmvc.pl/Content/okladka1.png" alt="C# 6.0 i MVC 5">
</a>
```

Następnie klikamy przycisk Zapisz.

Po zapisaniu i odświeżeniu strona wygląda następująco:

Aplikacja C# 6.0 i MVC 5 Ogłoszenia Kategorie Zdjęcia Edytor HTML

# Edytuj treść HTML

## Kod HTML:

```
<p style="color: red">Polecamy:</p>
<a href="http://aspnetmvc.pl/" target="_blank">
    <img style="max-width: 300px" src="http://aspnetmvc.pl/Content/okladka1.png"
alt="C# 6.0 i MVC 5">
</a>
```

Zapisz

## Podgląd w HTMLu:

Polecamy:


© 2015 - My ASP.NET Application

W polu input kod traktowany jest jako treść a w drugiej jako kod HTML.

# Edytor WYSIWYG

Wykorzystamy prawdopodobnie najpopularniejszy edytor o nazwie TinyMCE. Pobieramy paczkę plików ze strony tinymce.com a następnie wypakowujemy i kopiujemy folder tinymce do katalogu Content w naszej solucji. Następnie klikamy pokaż wszystkie pliki w VS i na wklejonym folderze w solution explorerze klikamy PPM i wybieramy Include in project.

Struktura będzie wyglądać następująco:


Następnie w pliku _Layout.cshtml w sekcji <head> dodajemy kod:

```
<script type="text/javascript" src="~/Content/tinymce/js/tinymce/tinymce.min.js"></script>
<script type="text/javascript">
        tinymce.init({
                selector: "textarea"
        });
</script>
```

Ostatnim krokiem jest zmiana w widoku EdytujTresc.cshtml.

Zamieniamy linię:

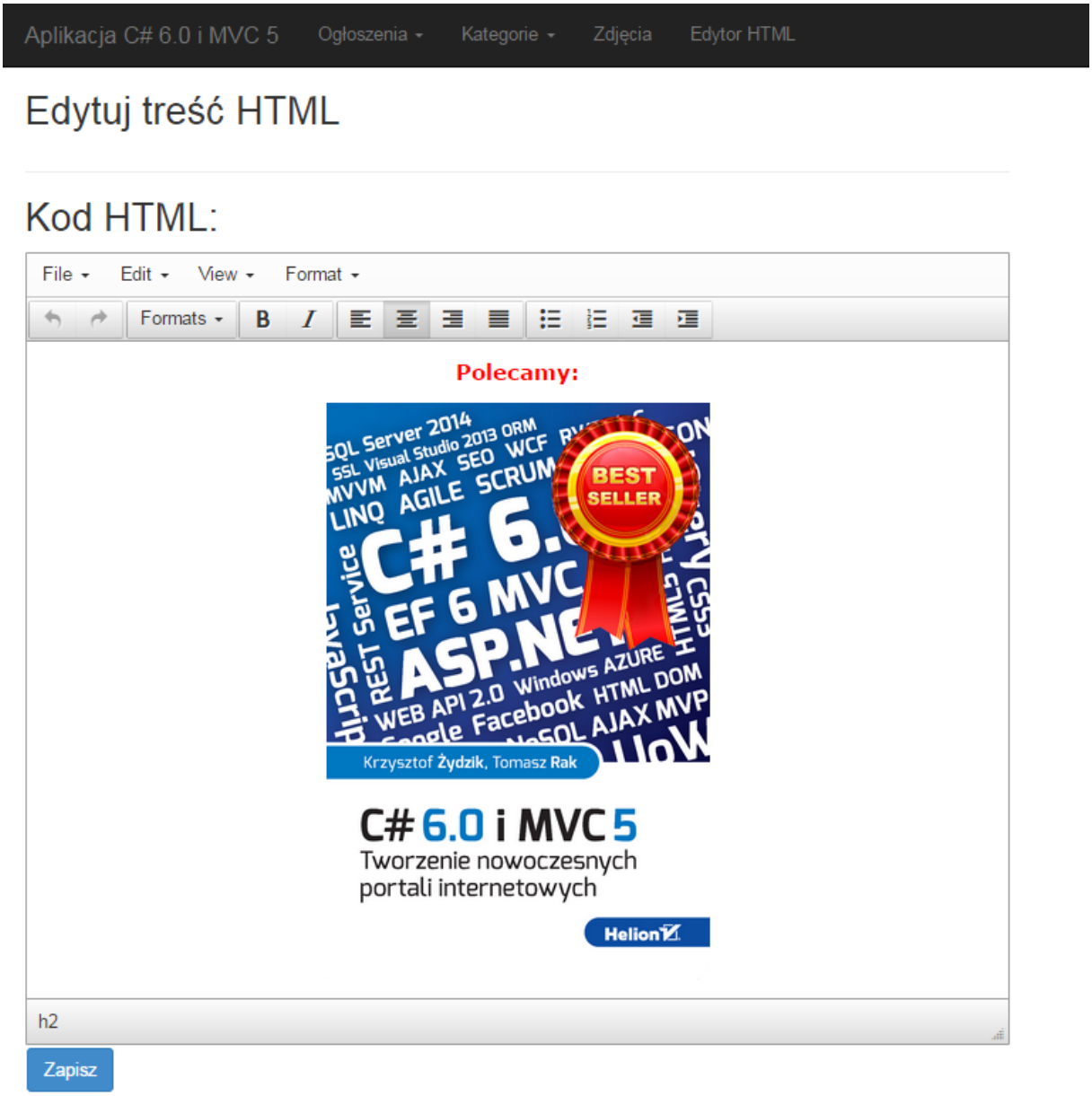
```
@Html.TextAreaFor(model => model.Tresc, 5, 80, new { style = "max-width:90%;" })
```

na:

```
@Html.TextAreaFor(model => model.Tresc)
```

Od teraz w miejscu dawnego pola wejściowego do wpisywania kodu HTML będzie się znajdował nasz edytor WYSIWYG. Działający w całości po stronie klienta dzięki wykorzystaniu ściągniętych bibliotek JavaScript.

**Wygląd edytora TinyMCE:**



# Podsumowanie

Po przerobieniu tego dodatku praktycznie wszystko, co jest potrzebne, aby stworzyć własny portal powinno być opanowane. Budowa bardziej skomplikowanych rozwiązań będzie się opierała głownie na miksie tych rozwiązań, które tutaj zostały opisane. Tworzenie ViewModeli składających się zarówno ze zdjęć jak i różnych innych pól.

# Co dalej?

Kilka pomysłów, które można zaimplementować, jako dalsza cześć nauki:

1. Powiązanie z płatnościami online,
2. Moduł do wysyłania wiadomości email w tle,
3. Moduł do fakturowania i generowania PDF,
4. Zarządzanie relacją wiele do wielu,
5. Logowanie błędów przy pomocy ELMAH lub LOG4NET,
6. Operacje na plikach,
7. Jakieś propozycje? - zapytania@aspnetmvc.pl