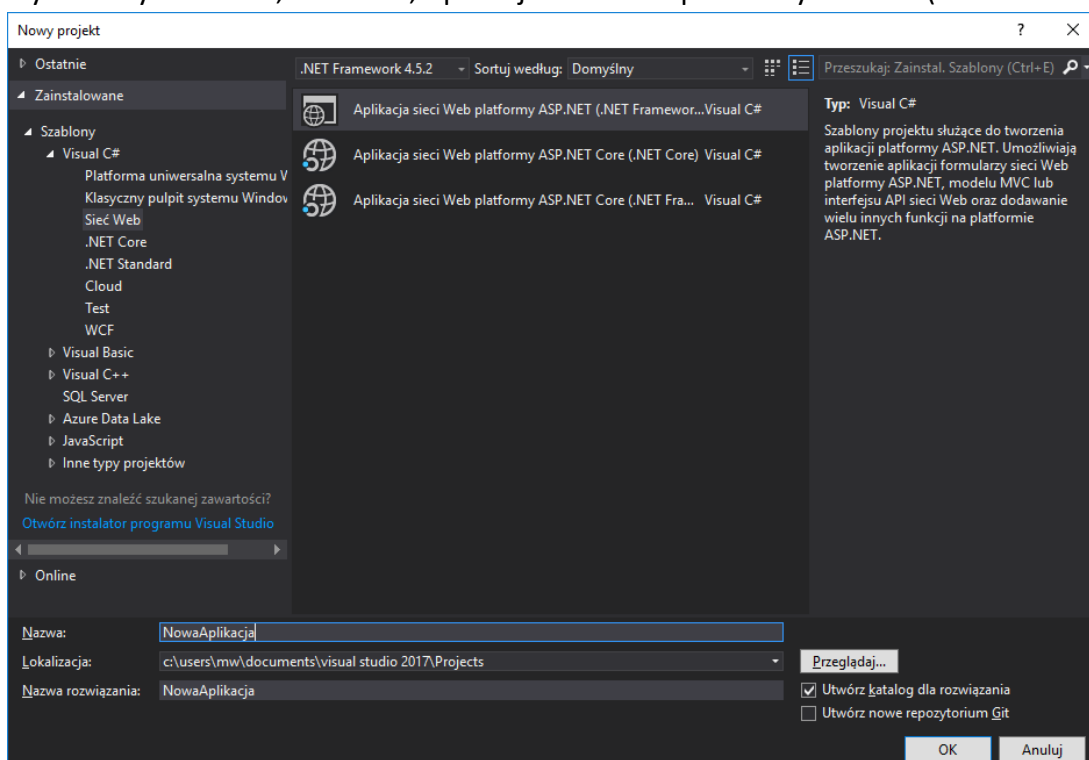


Instrukcja do laboratorium

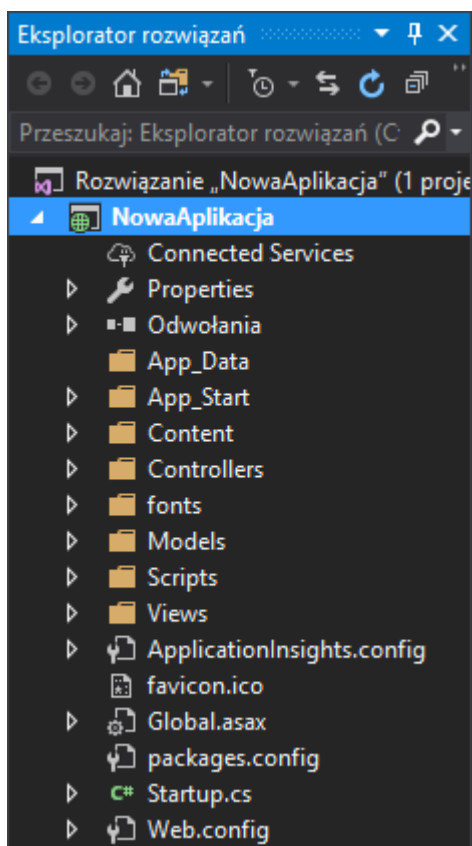
Technologia ASP.NET – tworzenie prostego systemu z podziałem na role

Część I

1. Otwieramy Visual Studio i tworzymy nowy projekt Plik->Nowy->Projekt. Z dostępnych szablonów wybieramy Visual C#, sieć Web, Aplikacja sieci Web platformy ASP.NET(.NET Framework).



2. Po kliknięciu OK w następnym oknie wybieramy szablon projektu MVC i uwierzytelnianie „Pojedyncze konta użytkowników”. Zostanie wygenerowany projekt o poniższej strukturze plików.



3. Budujemy i uruchamiamy projekt klikając Debugowanie -> Rozpocznij bez debugowania. Zostanie otwarta strona startowa naszej aplikacji w przeglądarce. Obecnie strona startowa naszej aplikacji wygląda tak:

ASP.NET

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

[Learn more »](#)

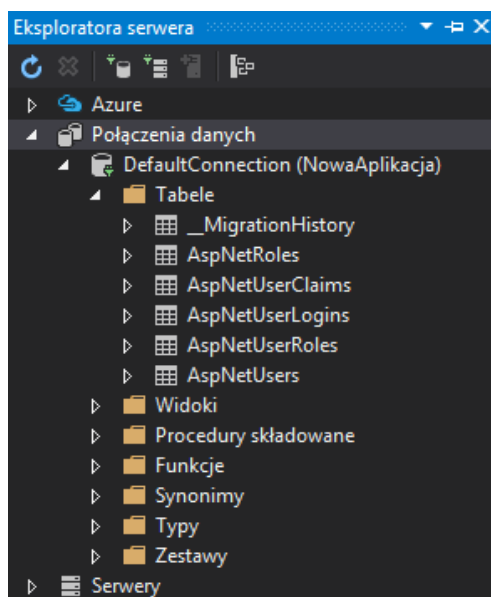
Getting started

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

[Learn more »](#)

4. Rejestrujemy się klikając Zarejestruj i uzupełniając formularz. Należy zapamiętać login i hasło, ponieważ temu konto nadamy uprawnienia administratora, a w bazie danych hasło nie jest zapisywane w sposób jawny.

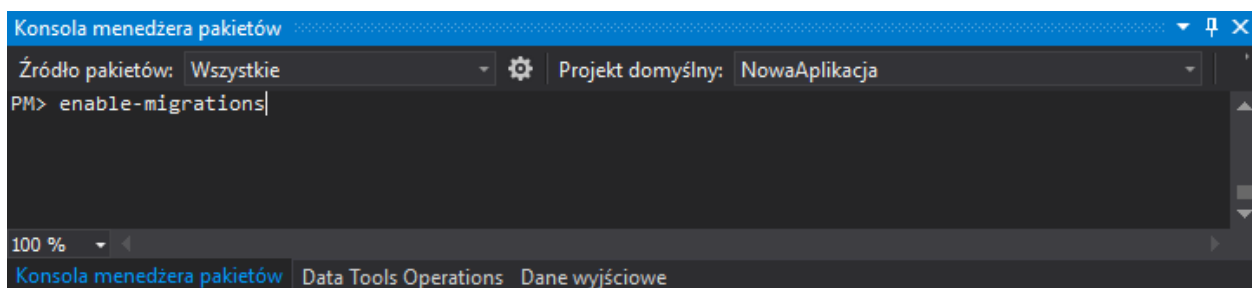
5. Zamykamy stronę i wracamy do Visual Studio. Teraz zobaczymy naszego użytkownika w bazie danych. Klikamy Widok -> Eksplorator serwera. Rozwijamy kolejno: Połączenia danych, DefaultConnection (NowaAplikacja), Tabele.



Na tabeli AspNetUsers klikamy prawym i wybieramy Pokaż dane tabeli. Zauważamy jeden rekord – jest to nasze konto użytkownika.

dbo.AspNetUsers [Data]					
	Id	Email	EmailConfirmed	PasswordHash	SecurityStamp
	e3-5909e5bd13b9	test@gmail.com	False	AlvuNGDBD5lh...	1e50a1c4-bc24-...
	NULL	NULL	NULL	NULL	NULL

6. Następnie musimy zezwolić na migracje z bazą danych. W tym celu uruchamiamy Konsolę menedżera pakietów: Narzędzia -> Menedżer pakietów NuGet -> Konsola menedżera pakietów. Po otwarciu konsoli wpisujemy w niej: enable-migrations. Pozwoli to zaktualizować strukturę bazy danych lub jej zawartość na podstawie kodu.



7. W strukturze plików projektu pojawił się folder Migrations, otwieramy go a następnie plik Configuration.cs. Teraz dodamy role do naszej aplikacji, w tym przykładzie są one trzy: Admin, Teacher, Student. Dodajemy poniższe linijki do metody Seed:

```
var RoleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));
string[] roleNames = { "Admin", "Teacher", "Student" };
IdentityResult roleResult;
foreach (var roleName in roleNames)
{
    if (!RoleManager.RoleExists(roleName))
    {
        roleResult = RoleManager.Create(new IdentityRole(roleName));
    }
}
```

Natomiast na początek klasy dodajemy zależności, aby zlikwidować podkreślenie:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
```

Tak powinna wyglądać cała metoda:

```
protected override void Seed(NowaAplikacja.Models.ApplicationDbContext context)
{
    // This method will be called after migrating to the latest version.

    // You can use the DbSet<T>.AddOrUpdate() helper extension method
    // to avoid creating duplicate seed data. E.g.
    //
    // context.People.AddOrUpdate(
    //     p => p.FullName,
    //     new Person { FullName = "Andrew Peters" },
    //     new Person { FullName = "Brice Lambson" },
    //     new Person { FullName = "Rowan Miller" }
    // );
    //
    var RoleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));
    string[] roleNames = { "Admin", "Teacher", "Student" };
    IdentityResult roleResult;
    foreach (var roleName in roleNames)
    {
        if (!RoleManager.RoleExists(roleName))
        {
            roleResult = RoleManager.Create(new IdentityRole(roleName));
        }
    }
}
```

8. Teraz zaktualizujemy bazę danych. Wpisujemy update-database w konsoli menedżera pakietów. W efekcie do tabeliAspNetRoles zostały dodane trzy role.

	Id	Name
	de1-9c8c244cfa08	Admin
	7431ae87-2f3c-...	Student
	a9e15a06-7029-...	Teacher
	NULL	NULL

9. Następnie dodamy do naszego konta uprawnienia administratora. W tym celu w metodzie Seed dodajemy jeszcze dwie linijki, gdzie „UserID” jest to identyfikator użytkownika z tabeliAspNetUsers. Należy go skopiować w to miejsce.

```
var UserManager = new UserManager<ApplicationUser>(new
UserStore<ApplicationUser>(context));
UserManager.AddToRole("UserID", "Admin");
```

Dodajemy kolejną zależność, gdzie pierwszy człon to nazwa naszej aplikacji jaką wybraliśmy przy tworzeniu projektu.

```
using NowaAplikacja.Models;
```

W efekcie w tabeli `AspNetUserRoles` możemy zauważyć rekord zawierający id naszego konta oraz id roli administratora.

dbo.AspNetUserRoles [Data]		dbo.AspNetUsers [Data]	Configuration.cs
		Max Rows: 1000	
	UserId	RoleId	
▶	e0e190bf-7cdc-452f-8ce3-5909e5bd13b9	2fead449-be82-4ec8-8de1-9c8c244cfa08	
⊞	NULL	NULL	

10. Kolejnym krokiem jest dodanie linijki odpowiedzialnej za przypisanie uprawnień użytkownikowi od razu podczas rejestracji. W kontrolerze `AccountController`, który znajduje się w folderze `Controllers` odszukujemy metodę `Register(RegisterViewModel model)` i dodajemy linijkę:

```
userManager.AddToRole(user.Id, "Student");
```

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            userManager.AddToRole(user.Id, "Student");
            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

            // For more information on how to enable account confirmation and password reset please
            // Wyślij wiadomość e-mail z tym łączem
            // string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            // var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code
            // await UserManager.SendEmailAsync(user.Id, "Potwierdź konto", "Potwierdź konto, klikaj

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

    // Dotarcie do tego miejsca wskazuje, że wystąpił błąd, wyświetl ponownie formularz
    return View(model);
}
```

Po tej czynności możemy uruchomić naszą aplikację, zarejestrować kolejnego użytkownika i zobaczyć jak to wygląda w bazie danych.

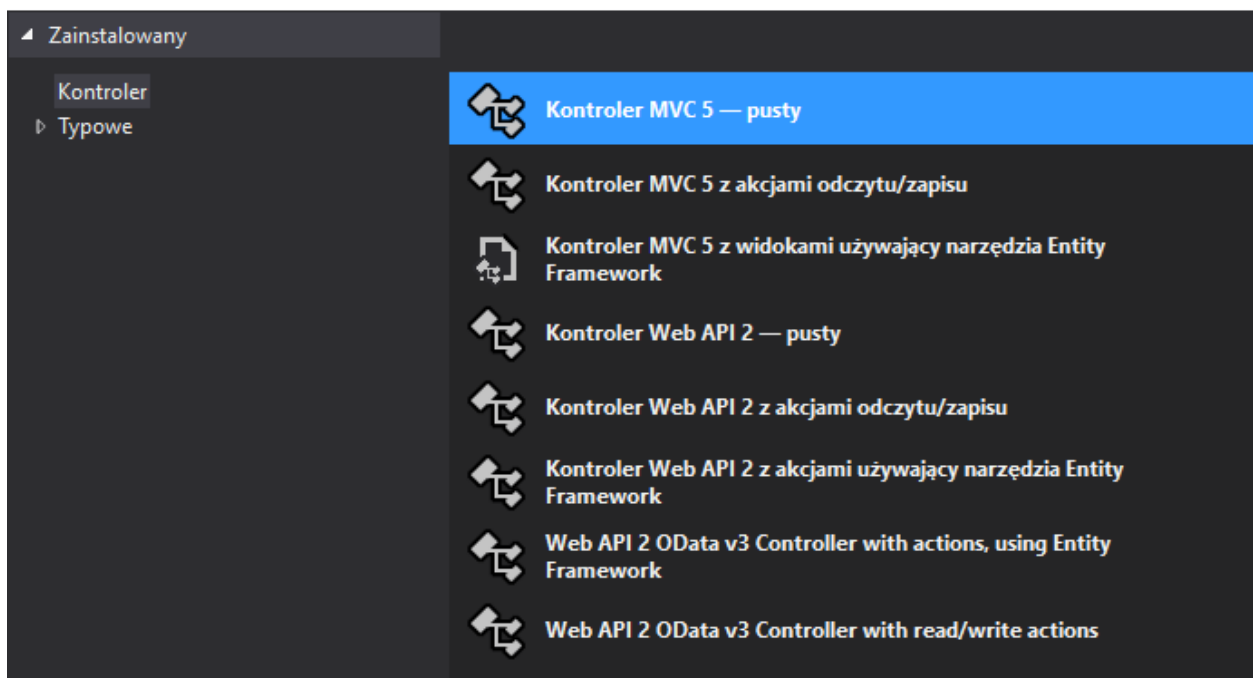
Część 2

1. W folderze Models tworzymy nową klasę o nazwie: AdminViewModels. W ciele nowo utworzonej klasy dodajemy model danych dla panelu administratora wyświetlającego listę użytkowników.

```
public class AdminUserViewModel
{
    public string UserName { get; set; }
    public string RankName { get; set; }
    public string UserId { get; set; }
    public string UserFullName { get; set; }
    public string RankId { get; set; }
}
```

2. W folderze Controllers dodajemy nowy pusty kontroler o nazwie AdminController. Znajdować się w nim będzie logika zarządzania użytkownikami przez administratora.

Dodawanie szkieletu



Na początek w ciele kontrolera dodajemy linijki:

```
public UserManager<ApplicationUser> UserManager { get; set; }
public ApplicationDbContext context { get; set; }
public static List<AdminUserViewModel> userList = new List<AdminUserViewModel>();
public static List<SelectListItem> roleList = new List<SelectListItem>();
public static string AdmUsrName { get; set; }
public static string AdmUsrEmail { get; set; }
public static string AdmUsrRole { get; set; }
public static string AdmUsrSrch { get; set; }
public static string AdmRankSrch { get; set; }
```

Pierwsze dwie linijki zapewniają dostęp do użytkowników oraz do bazy danych. Następne dwie to listy przechowujące użytkowników i dostępne w systemie role. Reszta to zmienne pomocnicze.

3. Następnie dodajemy konstruktor oraz kolejną metody:

```
public AdminController()
{
    context = new ApplicationDbContext();
    UserManager = new
    UserManager<ApplicationUser>(new UserStore<ApplicationUser>(context));
}
```

Teraz metodę odpowiedzialną za sporządzenie listy użytkowników. Pierwsza linijka już określa, że aby dostać się do listy użytkowników potrzebne będą uprawnienia administratora. Niżej można zauważyć wymienione role (muszą się one zgadzać z tymi określonymi na początku projektu), którym jest przypisywana „ważność”, gdzie 1 to najważniejszy.

```
[Authorize(Roles = "Admin")]
[AllowAnonymous("Index")]
public async Task<ActionResult> ShowUserDetails(AdminUserViewModel model)
{
    userList.Clear();
    IList<ApplicationUser> users = context.Users.ToList();
    foreach (var user in users)
    {
        var roles = await UserManager.GetRolesAsync(user.Id);
        model.UserName = user.UserName;
        foreach (var role in roles)
        {
            model.RankName = role;
            switch (role)
            {
                case "Admin":
                    model.RankId = "1";
                    break;
                case "Teacher":
                    model.RankId = "2";
                    break;
                case "Student":
                    model.RankId = "3";
                    break;
            }
        }
        model.UserId = user.Id;
        model.UserFullName = user.UserName;
        userList.Add(new AdminUserViewModel() { UserName = model.UserName, RankName =
        model.RankName, UserId = model.UserId, RankId = model.RankId, UserFullName =
        model.UserFullName });
        model.RankName = null;
    }
    return PartialView("ShowUserDetails");
}
```

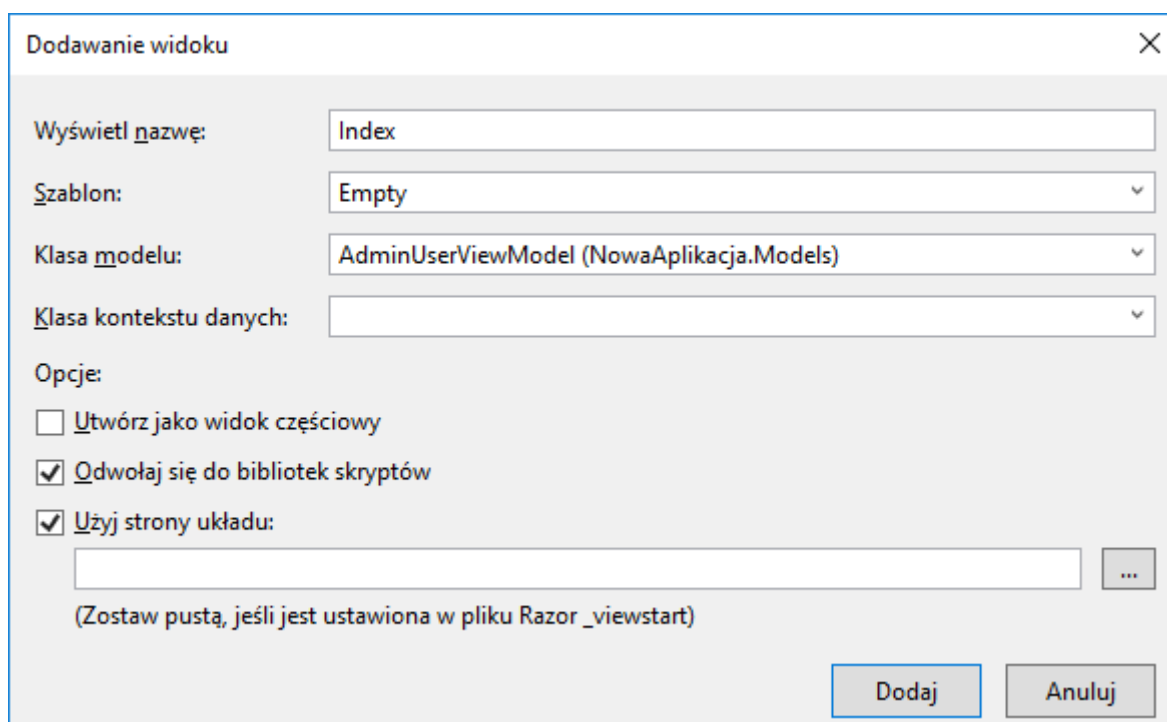
Kolejną metodą do dodania jest metoda odpowiedzialna za wyświetlenie listy.

```
[HttpGet]
[Authorize(Roles = "Admin")]
public async Task<ActionResult> Index(AdminUserViewModel model)
{
    await ShowUserDetails(model);
    return View();
}
```


Na koniec uzupełniamy przestrzeń nazw, za pomocą żarówek obok podkreślonych fragmentów lub od razu:

```
using Microsoft.AspNet.Identity;  
using Microsoft.AspNet.Identity.EntityFramework;  
using NowaAplikacja.Models;  
using System.Threading.Tasks;
```

4. Kolejnym krokiem jest dodanie widoku do ostatnio dodanej metody. W tym celu klikamy prawym przyciskiem na nazwę metody i dodaj widok. Wybieramy tak jak na poniższym zrzucie ekranu.



W widoku usuwamy wszystko i wklejamy:

```
@model NowaAplikacja.Models.AdminUserViewModel  
{  
    ViewBag.Title = "Admin Page";  
}  
<div class="userview">  
    @{Html.RenderPartial("ShowUserDetails");}  
}</div>
```

Tak samo dodajemy widok dla metody ShowUserDetails i zastępujemy zawartość następującymi linijkami:

```
@model NowaAplikacja.Models.AdminUserViewModel
@using NowaAplikacja.Controllers
@using System.Globalization

<h1> Użytkownicy w systemie</h1>
@{
    var totalUsers = AdminController.usrList.ToList().Count();
    int num = 0;
}
<h3>Liczba: @totalUsers</h3>

<table class="userDetails">
    <tr>
        <th class="tblnum">Nr</th>
        <th>Nazwa</th>
        <th>Ranga</th>
        <th>Opcje</th>
    </tr>
    @foreach (var usr in AdminController.usrList)
    {
        ViewBag.UserName =
CultureInfo.CurrentCulture.TextInfo.ToTitleCase(usr.UserName);
        ViewBag.UserFullName =
CultureInfo.CurrentCulture.TextInfo.ToTitleCase(usr.UserFullName);
        num++;
        <tr>
            <td class="tblnum">@num</td>
            <td title="@ViewBag.UserFullName">@ViewBag.UserName</td>
            <td>@usr.RankName</td>
            <td>
                @using (Html.BeginForm("EditUser", "Admin",
FormMethod.Post, new { @role = "form" }))
                {
                    @Html.AntiForgeryToken()
                    <input type="hidden" name="id" value="@usr.UserId"
                    <input type="submit" class="btn-default"
value="Edytuj" />
                }
            </td>
        </tr>
    }
</table>
```

5. Ostatnim krokiem przed sprawdzeniem co zostało zrobione jest dodanie zakładki do panelu administratora. W tym celu w pliku Views/Shared/_Layout dodajemy:

```
@if (User.IsInRole("Admin"))
{
    <li>@Html.ActionLink("Admin Panel", "Index", "Admin")</li>
}
```

```

<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li>@Html.ActionLink("Strona główna", "Index", "Home")</li>
    <li>@Html.ActionLink("Informacje", "About", "Home")</li>
    <li>@Html.ActionLink("Kontakt", "Contact", "Home")</li>
    @if (User.IsInRole("Admin"))
    {
      <li>@Html.ActionLink("Admin Panel", "Index", "Admin")</li>
    }
  </ul>
  @Html.Partial("_LoginPartial")
</div>

```

Layout stanowi szablon wyglądu naszej aplikacji, to co się nie zmienia na stronie. W tym kroku dodaliśmy link do panelu admina. W dodanej metodzie ActionLink kolejne stringi oznaczają: nazwę wyświetlaną na stronie, nazwę metody, która ma być wywołana po kliknięciu i nazwę kontrolera z którego pochodzi metoda do wywołania.

Teraz uruchamiamy aplikację, logujemy się na konto administratora i klikamy na Admin Panel. Widzimy listę użytkowników w systemie. Aktualnie nic z nimi nie możemy zrobić.

6. Dodajemy metody i widok wyświetlający stronę edycji użytkownika.

Na początek do modelu AdminViewModel dodamy dwa modele: do wyświetlania danych użytkownika i dla listy ról.

```

public class AdminEditViewModel
{
    public string UserName { get; set; }
    public string RankName { get; set; }
    public string Email { get; set; }
}
public class AdminRoleViewModel
{
    public string Role { get; set; }
    public string RoleId { get; set; }
    public string RoleValue { get; set; }
}

```

Do kontrolera AdminController dodajemy metodę EditUser odpowiedzialną za edycję użytkownika:

```

[HttpGet]
[Authorize(Roles = "Admin")]
public ActionResult EditUser()
{
    return View();
}

[HttpPost]
[Authorize(Roles = "Admin")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> EditUser(string id, AdminEditViewModel model)
{
    try
    {

```

```

        var user = UserManager.FindById(id);
        model.Email = user.Email;
        var roles = await UserManager.GetRolesAsync(user.Id);
        model.UserName = user.UserName;
        foreach (var role in roles)
        {
            model.RankName = role;
        }
        AdmUsrName = model.UserName;
        AdmUsrEmail = model.Email;
        AdmUsrRole = model.RankName;
        return RedirectToAction("EditUser");
    }
    catch
    {
        return View();
    }
}

```

Oprócz tego należy dodać jeszcze metodę odpowiedzialną za zwrócenie listy ról w systemie:

```

public IEnumerable<SelectListItem> GetUserRoles(string usrrole)
{
    var roles = context.Roles.OrderBy(x => x.Name).ToList();
    List<AdminRoleViewModel> rList = new List<AdminRoleViewModel>();
    rList.Add(new AdminRoleViewModel() { Role = "Admin", RoleId = "1" });
    rList.Add(new AdminRoleViewModel() { Role = "Teacher", RoleId = "2" });
    rList.Add(new AdminRoleViewModel() { Role = "Student", RoleId = "3" });
    rList = rList.OrderBy(x => x.RoleId).ToList();
    List<SelectListItem> roleNames = new List<SelectListItem>();
    foreach (var role in rList)
    {
        roleNames.Add(new SelectListItem()
        {
            Text = role.Role,
            Value = role.Role
        });
    }
    var selectedRoleName = roleNames.FirstOrDefault(d => d.Value == usrrole);
    if (selectedRoleName != null) selectedRoleName.Selected = true;
    return roleNames;
}

```

Klikamy prawym klawiszem myszy na metodę post EditUser i dodajemy widok, jako klasę modelu wybieramy AdminEditViewModel. Do widoku wklejamy kod, który jest odpowiedzialny za wyświetlenie danych użytkownika.

```

@model NowaAplikacja.Models.AdminEditViewModel
@using NowaAplikacja.Controllers
@{
    ViewBag.Title = "EditUser";
}

<div class="mngmcont">
    <div class="mngmsub">
        <h1>Edycja użytkownika</h1>
        @using (Html.BeginForm("SaveUser", "Admin", FormMethod.Post, new { role =
"form" }))
        {
            @Html.AntiForgeryToken()

```

```

        {
            var admCtrl = new AdminController();
            ViewBag.Roles =
admCtrl.GetUserRoles(AdminController.AdmUsrRole);
        }
        <table>
            <tr>
                <td>@Html.LabelFor(m => m.UserName)</td>
                <td>@Html.TextBoxFor(m => m.UserName, new { @class =
"form-control", @readonly = "readonly", Value = AdminController.AdmUsrName })</td>
            </tr>
            <tr>
                <td>@Html.LabelFor(m => m.RankName)</td>
                <td>@Html.DropDownListFor(m => m.RankName,
(IEnumerable<SelectListItem>)ViewBag.Roles, new { @class = "form-control" })</td>
            </tr>
            <tr>
                <td>@Html.LabelFor(m => m.Email)</td>
                <td>@Html.TextBoxFor(m => m.Email, new { @class = "form-
control", @readonly = "readonly", Value = AdminController.AdmUsrEmail })</td>
            </tr>
        </table>

        <input type="submit" value="Zapisz" class="btn btn-success" />
    }

    @using (Html.BeginForm("DeleteUser", "Admin", FormMethod.Get, new { role =
"form" }))
    {
        @Html.TextBoxFor(model => model.RankName, new { @class = "form-
control", @type = "hidden", Value = AdminController.AdmUsrRole })
        @Html.TextBoxFor(m => m.UserName, new { @class = "form-control", @type
= "hidden", @readonly = "readonly", Value = AdminController.AdmUsrName })
        <input type="submit" value="Usuń" class="btn btn-danger" />
    }
    <input type="button" value="Wstecz"
onclick="location.href='@Url.Action("Index","Admin")'" class="btn btn-default" />
</div>
</div>

```

7. Następnym krokiem jest dodanie metody zapisującej zmiany:

```

[HttpPost]
[Authorize(Roles = "Admin")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> SaveUser(string id, AdminEditViewModel model)
{
    try
    {
        AdmUsrRole = model.RankName;
        AdmUsrName = model.UserName;
        var userid = context.Users.Where(x => x.UserName == AdmUsrName).Select(x =>
x.Id).FirstOrDefault();
        var user = await UserManager.FindByIdAsync(userid);
        var userRoles = await UserManager.GetRolesAsync(user.Id);
        string[] roles = new string[userRoles.Count];
        userRoles.CopyTo(roles, 0);
        await UserManager.RemoveFromRolesAsync(user.Id, roles);
        await UserManager.AddToRoleAsync(user.Id, AdmUsrRole);
    }
}

```

```

        return RedirectToAction("Index", "Admin", new { Message =
ManageMessageId.UserUpdated });
    }
    catch
    {
        return RedirectToAction("Index", "Admin", new { Message =
ManageMessageId.Error });
    }
}

```

Oraz dodanie typu wyliczeniowego na końcu klasy kontrolera, ma to związek z komunikatami informującymi czym operacje się powiodła czy nie.

```

public enum ManageMessageId
{
    HighRankedUser,
    Error,
    UserDeleted,
    UserUpdated
}

```

8. Następnie dodajemy metodę get usuwania użytkownika:

```

[HttpGet]
[Authorize(Roles = "Admin")]
public ActionResult DeleteUser()
{
    return View();
}

```

Dodajemy do niej widok i nim wklejamy:

```

@{
    ViewBag.Title = "DeleteUser";
}

<h1>Usuwanie</h1>
@using (Html.BeginForm("DeleteUser", "Admin", FormMethod.Post, new { @role = "form" }))
{
    @Html.AntiForgeryToken()
    <div class="form-group">
        <div>Czy na pewno chcesz usunąć wybranego użytkownika?</div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Usuń" class="btn btn-danger" />
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="button" value="Anuluj"
onclick="location.href='@Url.Action("EditUser","Admin")'" class="btn btn-default"
style="float:left" />
        </div>
    </div>
}

```

Następnie dodajemy kod metody post DeleteUser:

```
[HttpPost]
[Authorize(Roles = "Admin")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteUser(string userid)
{
    if (AdmUsrRole == "Admin")
    {
        return RedirectToAction("Index", "Admin", new { Message =
ManageMessageId.HighRankedUser });
    }
    userid = context.Users.Where(x => x.UserName == AdmUsrName).Select(x =>
x.Id).FirstOrDefault();
    var user = await UserManager.FindByIdAsync(userid);
    var userClaims = await UserManager.GetClaimsAsync(user.Id);
    var userRoles = await UserManager.GetRolesAsync(user.Id);
    var userLogins = await UserManager.GetLoginsAsync(user.Id);
    foreach (var claim in userClaims)
    {
        await UserManager.RemoveClaimAsync(user.Id, claim);
    }
    string[] roles = new string[userRoles.Count];
    userRoles.CopyTo(roles, 0);
    await UserManager.RemoveFromRolesAsync(user.Id, roles);
    foreach (var log in userLogins)
    {
        await UserManager.RemoveLoginAsync(user.Id, new
UserLoginInfo(log.LoginProvider, log.ProviderKey));
    }
    await UserManager.DeleteAsync(user);
    return RedirectToAction("Index", "Admin", new { Message =
ManageMessageId.UserDeleted });
}
```

9. Aktualizujemy metodę Index o obsługę komunikatów:

```
[HttpGet]
[Authorize(Roles = "Admin")]
public async Task<ActionResult> Index(AdminUserViewModel model, ManageMessageId? message =
null)
{
    ViewBag.StatusMessage = message == ManageMessageId.UserDeleted ? " Konto użytkownika
zostało pomyślnie usunięte."
: message == ManageMessageId.UserUpdated ? "Konto użytkownika
zostało zaaktualizowane."
: "";
    ViewBag.ErrorMessage = message == ManageMessageId.Error ? "Błąd."
: message == ManageMessageId.HighRankedUser ? "Admin nie może
zostać usunięty."
: "";
    await ShowUserDetails(model);
    return View();
}
```

W widoku Admin/Index dodajemy linijki, odpowiedzialne za wyświetlanie komunikatu:

```
<p class="text-success">@ViewBag.StatusMessage</p>
<p class="text-error">@ViewBag.ErrorMessage</p>
```

10. Możemy uruchomić aplikację, zalogować się na konto administratora i przetestować dodane metody. Możliwe jest usunięcie użytkownika z aplikacji oraz zmiana rangi. Nie można edytować danych użytkownika gdyż przechowujemy tylko e-mail i hasło użytkownika.

Część III

W tej części stworzymy kilka przykładowych metod wraz z widokami w celu zaprezentowania jak działa system rang.

1. Dodajemy nowy kontroler o nazwie ExampleController.
2. Podmieniamy ciała klasy kontrolera na kod z kilkoma metodami potrzebujących różnych uprawnień do oglądania ich.

```
public class ExampleController : Controller
{
    // GET: Example
    public ActionResult Index()
    {
        ViewBag.StatusMessage = "To jest strona dostępna dla wszystkich.";
        return View();
    }

    [Authorize(Roles = "Teacher")]
    public ActionResult OnlyTeachers()
    {
        ViewBag.StatusMessage = "To jest strona dostępna tylko dla
nauczycieli.";
        return View();
    }

    [Authorize(Roles = "Student")]
    public ActionResult OnlyStudents()
    {
        ViewBag.StatusMessage = "To jest strona dostępna tylko dla
studentów.";
        return View();
    }

    [Authorize(Roles = "Admin")]
    public ActionResult OnlyAdmin()
    {
        ViewBag.StatusMessage = "To jest strona dostępna tylko dla
administratora.";
        return View();
    }

    [Authorize(Roles = "Student,Teacher")]
    public ActionResult StudentsAndTeachers()
    {
        ViewBag.StatusMessage = "To jest strona dostępna dla studentów i
nauczycieli.";
        return View();
    }
}
```

3. Dodajemy widok dla metody Index. Wybieramy Empty(bez modelu) i dodajemy do niego:


```
@ViewBag.StatusMessage
<hr />
To są odnośniki do stworzonych metod:
<ul>
    <li>@Html.ActionLink("Teacher", "OnlyTeachers", "Example")</li>
    <li>@Html.ActionLink("Student", "OnlyStudents", "Example")</li>
    <li>@Html.ActionLink("Admin", "OnlyAdmin", "Example")</li>
    <li>@Html.ActionLink("Students And Teachers", "StudentsAndTeachers", "Example")</li>
</ul>
```

4. Następnie tworzymy w takim sam sposób widoki do pozostałych 4 metod, dodając do nich jedynie:

```
@ViewBag.StatusMessage
```

Jest to miejsce w którym będzie wyświetlony komunikat pochodzący od metody.

5. Ostatnim krokiem jest dodanie odnośnika w Layout naszego projektu. Obok innych linków dodajemy:

```
<li>@Html.ActionLink("Example", "Index", "Example")</li>
```

6. Uruchamiamy aplikację i sprawdzamy co się stanie przy przejściu do poszczególnych metod. Należy to sprawdzić z kont z różnymi uprawnieniami.