Jason De Oliveira
Michel Bruchet

ASP.NET Core 2.0

Wprowadzenie

Helion
Packt

Tytuł oryginału: Learning ASP.NET Core 2.0

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-4499-0

HELION SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/asp2wp.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/asp2wp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

9

ROZDZIAŁ

Zabezpieczanie aplikacji ASP.NET Core 2.0

W dzisiejszym świecie przy wzroście liczby przestępstw komputerowych i oszustw internetowych wszystkie nowoczesne aplikacje WWW wymagają implementacji silnych mechanizmów bezpieczeństwa w celu zapobiegania atakom i przywłaszczeniu tożsamości użytkowników.

Do tej pory skupialiśmy się na poznaniu metod tworzenia efektywnych aplikacji internetowych na platformie ASP.NET Core 2.0, nie myśląc wcale o uwierzytelnianiu i autoryzacji użytkowników ani ochronie danych, ale ponieważ aplikacja *Kółko i krzyżyk* staje się coraz bardziej skomplikowana, to przed końcowym, ogólnodostępnym wdrożeniem musimy się zająć sprawami bezpieczeństwa.

Tworzenie aplikacji internetowej bez myślenia o bezpieczeństwie jest wielkim niedociągnięciem i może doprowadzić do upadku nawet największe i najbardziej znane witryny. W przypadku naruszenia bezpieczeństwa i kradzieży danych osobowych zła reputacja i uderzenie w zaufanie użytkowników są olbrzymie i nikt nie chce więcej mieć do czynienia z takimi aplikacjami, a co gorsza — firmami.

To zagadnienie musi być traktowane bardzo poważnie. Należy pracować z firmami zajmującymi się bezpieczeństwem w celu weryfikacji kodu i przeprowadzania testów penetracyjnych, aby zapewnić zgodność z najlepszymi rozwiązaniami i wysokimi standardami bezpieczeństwa (na przykład OWASP10).

Na szczęście platforma ASP.NET Core 2.0 zawiera wszystko, co potrzebne, aby pomóc w tym ważnym — choć skomplikowanym — zagadnieniu. Większość wbudowanych w nią funkcji nie wymaga nawet zaawansowanej umiejętności programowania ani znajomości kwestii bezpieczeństwa. Zobaczysz, że dzięki użyciu frameworka Identity platformy ASP.NET Core 2.0 poznanie i implementacja bezpiecznych aplikacji są bardzo proste.

W tym rozdziale poruszymy następujące zagadnienia:

- wprowadzanie podstawowego uwierzytelniania za pomocą formularza;
- wprowadzanie uwierzytelniania za pośrednictwem zewnętrznego dostawcy;
- dodawanie mechanizmu resetowania zapomnianego hasła;
- praca z uwierzytelnianiem dwuskładnikowym;
- implementacja autoryzacji.

Implementacja uwierzytelniania

Uwierzytelnianie pozwala na identyfikację w aplikacji konkretnego użytkownika. Nie używa się go do zarządzania prawami dostępu użytkowników, bo to jest rola autoryzacji, ani do ochrony danych.

Istnieje kilka metod uwierzytelniania użytkowników aplikacji, takich jak:

- podstawowe uwierzytelnianie za pomocą formularza logowania z polami loginu i hasła;
- uwierzytelnianie przez **pojedyncze logowanie** (ang. *Single Sign-On* — **SSO**), w przypadku którego użytkownik uwierzytelnia się tylko raz do wszystkich aplikacji w ramach swojej firmy;
- uwierzytelnianie za pośrednictwem zewnętrznego dostawcy w sieciach społecznościowych (takich jak Facebook i LinkedIn);
- uwierzytelnianie za pomocą certyfikatów lub **infrastruktury klucza publicznego** (ang. *public key infrastructure* — **PKI**).

Platforma ASP.NET Core 2.0 obsługuje te wszystkie metody, ale w tym rozdziale skoncentrujemy się na uwierzytelnianiu za pomocą formularzy z loginem użytkownika i hasłem oraz na uwierzytelnianiu za pośrednictwem zewnętrznego dostawcy — serwisu Facebook.

W kolejnych przykładach przyjrzysz się użyciu tych metod do uwierzytelniania użytkowników aplikacji, a także bardziej zaawansowanym możliwościom, na przykład mechanizmom potwierdzania adresu e-mail oraz resetowania hasła.

Na koniec zobaczysz, jak w przypadku kluczowych aplikacji za pomocą funkcji wbudowanych w platformę ASP.NET Core 2.0 zaimplementować uwierzytelnianie dwuskładnikowe.

Przygotujmy implementację różnych mechanizmów uwierzytelniania w aplikacji *Kółko i krzyżyk*:

1. W klasie `Startup` zmień czas istnienia instancji `UserService`, `GameInvitationService` i `GameSessionService`:

```
services.AddTransient<IUserService, UserService>();
services.AddScoped<IGameInvitationService,
 GameInvitationService>();
services.AddScoped<IGameSessionService, GameSessionService>();
```

2. Zmień metodę `Configure` klasy `Startup`, dodając zaraz po oprogramowaniu pośredniczącym plików statycznych wywołanie oprogramowania pośredniczącego uwierzytelniania:

```
app.UseStaticFiles();
app.UseAuthentication();
```

3. Zmień klasę `UserModel`, aby dostosować ją do funkcji uwierzytelniania wybudowanych we framework Identity platformy ASP.NET Core 2.0, usuwając właściwości `Id` i `Email`, które ma już klasa `IdentityUser`:

```
public class UserModel : IdentityUser<Guid>
{
  [Display(Name = "FirstName")]
  [Required(ErrorMessage = "FirstNameRequired")]
  public string FirstName { get; set; }
  [Display(Name = "LastName")]
  [Required(ErrorMessage = "LastNameRequired")]
  public string LastName { get; set; }
  [Display(Name = "Password")]
  [Required(ErrorMessage = "PasswordRequired"),
   DataType(DataType.Password)]
  public string Password { get; set; }
  [NotMapped]
  public bool IsEmailConfirmed
  {
    get { return EmailConfirmed; }
  }
  public System.DateTime? EmailConfirmationDate { get; set; }
  public int Score { get; set; }
}
```

W rzeczywistych zastosowaniach należałoby usunąć również właściwość `Password`, jednak dla przejrzystości i w celach edukacyjnych w tym przykładzie ją pozostawimy.

4. Dodaj folder o nazwie *Managers*, a w nim klasę `ApplicationUserManager`:

```
public class ApplicationUserManager : UserManager<UserModel>
{
  private IUserStore<UserModel> _store;
  DbContextOptions<GameDbContext> _dbContextOptions;
  public ApplicationUserManager(
   DbContextOptions<GameDbContext> dbContextOptions,
   IUserStore<UserModel> store, IOptions<IdentityOptions>
   optionsAccessor, IPasswordHasher<UserModel> passwordHasher,
   IEnumerable<IUserValidator<UserModel>> userValidators,
   IEnumerable<IPasswordValidator<UserModel>>
   passwordValidators, ILookupNormalizer keyNormalizer,
   IdentityErrorDescriber errors, IServiceProvider services,
   ILogger<UserManager<UserModel>> logger) :
    base(store, optionsAccessor, passwordHasher,
```

```
    userValidators, passwordValidators, keyNormalizer,
    errors, services, logger)
{
  _store = store;
  _dbContextOptions = dbContextOptions;
}

public override async Task<UserModel> FindByEmailAsync(
 string email)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
    return await dbContext.Set<UserModel>().FirstOrDefaultAsync(
     x => x.Email == email);
  }
}

public override async Task<UserModel> FindByIdAsync(
 string userId)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
    Guid id = Guid.Parse(userId);
    return await dbContext.Set<UserModel>().FirstOrDefaultAsync(
     x => x.Id == id);
  }
}

public override async Task<IdentityResult>
 UpdateAsync(UserModel user)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
    var current =
      await dbContext.Set<UserModel>().FirstOrDefaultAsync(x => x.Id == user.Id);
    current.AccessFailedCount = user.AccessFailedCount;
    current.ConcurrencyStamp = user.ConcurrencyStamp;
    current.Email = user.Email;
    current.EmailConfirmationDate = user.EmailConfirmationDate;
    current.EmailConfirmed = user.EmailConfirmed;
    current.FirstName = user.FirstName;
    current.LastName = user.LastName;
    current.LockoutEnabled = user.LockoutEnabled;
    current.NormalizedEmail = user.NormalizedEmail;
    current.NormalizedUserName = user.NormalizedUserName;
    current.PhoneNumber = user.PhoneNumber;
    current.PhoneNumberConfirmed = user.PhoneNumberConfirmed;
    current.Score = user.Score;
    current.SecurityStamp = user.SecurityStamp;
    current.TwoFactorEnabled = user.TwoFactorEnabled;
    current.UserName = user.UserName;
    await dbContext.SaveChangesAsync();
```

```
        return IdentityResult.Success;
      }
    }

    public override async Task<IdentityResult>
     ConfirmEmailAsync(UserModel user, string token)
    {
      var isValide = await base.VerifyUserTokenAsync(user,
       Options.Tokens.EmailConfirmationTokenProvider, ConfirmEmailTokenPurpose, token);
      if (isValide)
      {
        using (var dbContext = new GameDbContext(_dbContextOptions))
        {
          var current = await dbContext.UserModels.FindAsync(user.Id);
          current.EmailConfirmationDate = DateTime.Now;
          current.EmailConfirmed = true;
          await dbContext.SaveChangesAsync();
          return IdentityResult.Success;
        }
      }
      return IdentityResult.Failed();
    }
  }
```

5. W klasie `Startup` zarejestruj instancję `ApplicationUserManager`:

```
services.AddTransient<ApplicationUserManager>();
```

6. Dostosuj usługę `UserService` do współpracy z klasą `ApplicationUserManager`, dodaj dwie metody `GetEmailConfirmationCode` i `ConfirmEmail` oraz uaktualnij interfejs `IUserService`:

```
public class UserService
{
  private ILogger<UserService> _logger;
  private ApplicationUserManager _userManager;
  public UserService(ApplicationUserManager userManager, ILogger<UserService> logger)
  {
    _userManager = userManager;
    _logger = logger;

    var emailTokenProvider = new EmailTokenProvider<UserModel>();
    _userManager.RegisterTokenProvider("Default", emailTokenProvider);
  }

  public async Task<bool> ConfirmEmail(string email, string code)
  {
    var start = DateTime.Now;
    _logger.LogTrace($"Potwierdzenie adresu e-mail {email}");

    var stopwatch = new Stopwatch();
    stopwatch.Start();

    try
```

```
    {
      var user = await _userManager.FindByEmailAsync(email);

      if (user == null)
        return false;

      var result = await _userManager.ConfirmEmailAsync(user, code);
      return result.Succeeded;
    }
    catch (Exception ex)
    {
      _logger.LogError($"Nie można potwierdzić adresu e-mail {email} - {ex}");
      return false;
    }
    finally
    {
      stopwatch.Stop();
      _logger.LogTrace($"Potwierdzenie adresu e-mail użytkownika ukończono w ciągu
      ↪{stopwatch.Elapsed}");
    }
  }

  public async Task<string> GetEmailConfirmationCode(UserModel user)
  {
    return
     await _userManager.GenerateEmailConfirmationTokenAsync(user);
  }

  public async Task<bool> RegisterUser(UserModel userModel)
  {
    var start = DateTime.Now;
    _logger.LogTrace($"Rozpoczęcie rejestracji użytkownika {userModel.Email} -
    ↪{start}");

    var stopwatch = new Stopwatch();
    stopwatch.Start();

    try
    {
      userModel.UserName = userModel.Email;
      var result = await _userManager.CreateAsync(userModel, userModel.Password);
      return result == IdentityResult.Success;
    }
    catch (Exception ex)
    {
      _logger.LogError($"Nie można zarejestrować użytkownika {userModel.Email}
       ↪- {ex}");
      return false;
    }
    finally
    {
      stopwatch.Stop();
```

```
        _logger.LogTrace($"Rozpoczęcie rejestracji użytkownika {userModel.Email}
        ↪zakończono o {DateTime.Now} - upłynęło(y){stopwatch.Elapsed.TotalSeconds}
        ↪sekund(y)");
      }
    }

    public async Task<UserModel> GetUserByEmail(string email)
    {
      return await _userManager.FindByEmailAsync(email);
    }

    public async Task<bool> IsUserExisting(string email)
    {
      return (await _userManager.FindByEmailAsync(email)) != null;
    }

    public async Task<IEnumerable<UserModel>> GetTopUsers(int numberOfUsers)
    {
      return await _userManager.Users.OrderByDescending(x => x.Score).ToListAsync();
    }

    public async Task UpdateUser(UserModel userModel)
    {
      await _userManager.UpdateAsync(userModel);
    }
  }
```

Powinieneś również poprawić klasę `UserServiceTest`, żeby działała z nowym konstruktorem. W tym celu musisz także utworzyć atrapę klasy `UserManager` i przekazać ją do konstruktora. W chwili obecnej możesz po prostu zakomentować ten test i zaktualizować go później. Ale nie zapomnij tego zrobić!

7. Popraw metodę `EmailConfirmation` klasy `UserRegistrationController`, używając do pobrania kodu potwierdzenia adresu e-mail dodanej wcześniej metody `GetEmailConfirmationCode`:

```
var urlAction = new UrlActionContext
{
  Action = "ConfirmEmail",
  Controller = "UserRegistration",
  Values = new { email, code =
   await _userService.GetEmailConfirmationCode(user) },
  Protocol = Request.Scheme,
  Host = Request.Host.ToString()
};
```

8. Popraw metodę `ConfirmEmail` klasy `UserRegistrationController`; w celu zakończenia potwierdzania adresu e-mail musi ona wywoływać metodę `ConfirmEmail` klasy `UserService`:

```
[HttpGet]
public async Task<IActionResult> ConfirmEmail(string email,
 string code)
{
  var confirmed = await _userService.ConfirmEmail(email, code);

  if (!confirmed)
    return BadRequest();

  return RedirectToAction("Index", "Home");
}
```

9. Do folderu *Models* dodaj klasę `RoleModel`; niech dziedziczy po klasie generycznej `IdentityRole< Guid>`, ponieważ będzie używana przez funkcje uwierzytelnienia frameworka Identity platformy ASP.NET Core 2.0:

```
public class RoleModel : IdentityRole<Guid>
{
  public RoleModel()
  {
  }

  public RoleModel(string roleName) : base(roleName)
  {
  }
}
```

10. Popraw klasę `GameDbContext`, dodając instancję `DbSet` z modelami ról:

```
public DbSet<RoleModel> RoleModels { get; set; }
```

11. W klasie `Startup` zarejestruj usługę uwierzytelniania i usługę tożsamości z użyciem dodanej wcześniej klasy `RoleModel`:

```
services.AddIdentity<UserModel, RoleModel>(options =>
{
  options.Password.RequiredLength = 1;
  options.Password.RequiredUniqueChars = 0;
  options.Password.RequireNonAlphanumeric = false;
  options.Password.RequireUppercase = false;
  options.SignIn.RequireConfirmedEmail = false;
}).AddEntityFrameworkStores<GameDbContext>()
.AddDefaultTokenProviders();

services.AddAuthentication(options => {
  options.DefaultScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultSignInScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
}).AddCookie();
```

12. Zmień klasę `CommunicationMiddleware`, usuwając z niej prywatną składową `_userService` i odpowiednio poprawiając konstruktor:

```
public CommunicationMiddleware(RequestDelegate next)
{
  _next = next;
}
```

13. W klasie `CommunicationMiddleware` popraw obie metody `ProcessEmailConfirmation`; aby mogły współpracować z frameworkiem Identity platformy ASP.NET Core 2.0, muszą być asynchroniczne:

```
private async Task ProcessEmailConfirmation(HttpContext
 context, WebSocket currentSocket, CancellationToken ct,
 string email)
{
  var userService =
    context.RequestServices.GetRequiredService<IUserService>();
  var user = await userService.GetUserByEmail(email);
  while (!ct.IsCancellationRequested &&
         !currentSocket.CloseStatus.HasValue &&
          user?.IsEmailConfirmed == false)
  {
    await SendStringAsync(currentSocket,
     "WaitEmailConfirmation", ct);
    await Task.Delay(500);
    user = await userService.GetUserByEmail(email);
  }

  if (user.IsEmailConfirmed)
  {
    await SendStringAsync(currentSocket, "OK", ct);
  }
}

private async Task ProcessEmailConfirmation(HttpContext context)
{
  var userService =
    context.RequestServices.GetRequiredService<IUserService>();
  var email = context.Request.Query["email"];

  UserModel user = await userService.GetUserByEmail(email);

  if (string.IsNullOrEmpty(email))
  {
    await context.Response.WriteAsync("Nieprawidłowe żądanie:
    ↪Wymagany jest adres e-mail");
  }
  else if ((await
   userService.GetUserByEmail(email)).IsEmailConfirmed)
  {
    await context.Response.WriteAsync("OK");
  }
}
```

14. W klasie `GameInvitationService` zmień konstruktor publiczny na statyczny.
15. Z klasy `Startup` usuń następującą rejestrację instancji `DbContextOptions`; w następnym kroku zostanie zmieniona na inną:

```
var dbContextOptionsbuilder =
  new DbContextOptionsBuilder<GameDbContext>()
   .UseSqlServer(connectionString);
services.AddSingleton(dbContextOptionsbuilder.Options);
```

16. Dodaj nową rejestrację instancji `DbContextOptions` w klasie `Startup`:

```
services.AddScoped(typeof(DbContextOptions<GameDbContext>),
(serviceProvider) =>
{
  return new DbContextOptionsBuilder<GameDbContext>()
   .UseSqlServer(connectionString).Options;
});
```

17. Na końcu metody `Configure` klasy `Startup` zmień kod dokonujący migracji bazy danych:

```
var provider = app.ApplicationServices;
var scopeFactory =
  provider.GetRequiredService<IServiceScopeFactory>();
using (var scope = scopeFactory.CreateScope())
using (var context =
  scope.ServiceProvider.GetRequiredService<GameDbContext>())
{
  context.Database.Migrate();
}
```

18. Zmień metodę `Index` klasy `GameInvitationController`:

```
...
var invitation =
  gameInvitationService.Add(gameInvitationModel).Result;
return RedirectToAction("GameInvitationConfirmation",
 new { id = invitation.Id });
...
```

19. Zmień metodę `ConfirmGameInvitation` klasy `GameInvitationController`, dodając do istniejącej rejestracji użytkownika dodatkowe pola:

```
await _userService.RegisterUser(new UserModel
{
  Email = gameInvitation.EmailTo,
  EmailConfirmationDate = DateTime.Now,
  EmailConfirmed = true,
  FirstName = "",
  LastName = "",
  Password = "Azerty123!",
  UserName = gameInvitation.EmailTo
});
```

Automatyczne tworzenie i rejestracja zaproszonego użytkownika są tylko tymczasowym obejściem wprowadzonym po to, by uprościć przykładową aplikację. W rzeczywistych warunkach należy inaczej obsłużyć ten przypadek i zamienić tymczasowe obejście na prawdziwe rozwiązanie.

20. W klasie `GameSessionService` zmień metody `CreateGameSession` i `AddTurn` oraz ponownie wyodrębnij interfejs `IGameSessionService`:

```
public async Task<GameSessionModel> CreateGameSession(
 Guid invitationId, UserModel invitedBy,
 UserModel invitedPlayer)
{
  var session = new GameSessionModel
  {
    User1 = invitedBy,
    User2 = invitedPlayer,
    Id = invitationId,
    ActiveUser = invitedBy
  };
  _sessions.Add(session);
  return session;
}

public async Task<GameSessionModel> AddTurn(Guid id,
 UserModel user, int x, int y)
{
  List<Models.TurnModel> turns;
  var gameSession = _sessions.FirstOrDefault(session =>
   session.Id == id);
  if (gameSession.Turns != null && gameSession.Turns.Any())
    turns = new List<Models.TurnModel>(gameSession.Turns);
  else
    turns = new List<TurnModel>();

  turns.Add(new TurnModel
  {
    User = user,
    X = x,
    Y = y,
    IconNumber = user.Email == gameSession.User1?.Email ? "1" : "2"
  });

  gameSession.Turns = turns;
  gameSession.TurnNumber = gameSession.TurnNumber + 1;
  if (gameSession.User1?.Email == user.Email)
    gameSession.ActiveUser = gameSession.User2;
  else
    gameSession.ActiveUser = gameSession.User1;

  gameSession.TurnFinished = true;
  _sessions = new ConcurrentBag<GameSessionModel>
```

```
    (_sessions.Where(u => u.Id != id))
  {
    gameSession
  };
  return gameSession;
}
```

21. Zmień metodę `Index` klasy `GameSessionController`:

```
public async Task<IActionResult> Index(Guid id)
{
  var session = await _gameSessionService.GetGameSession(id);
  var userService =
    HttpContext.RequestServices.GetService<IUserService>();

  if (session == null)
  {
    var gameInvitationService =
      Request.HttpContext.RequestServices.GetService
      <IGameInvitationService>();
    var invitation = await gameInvitationService.Get(id);

    var invitedPlayer =
      await userService.GetUserByEmail(invitation.EmailTo);
    var invitedBy =
      await userService.GetUserByEmail(invitation.InvitedBy);

    session =
      await _gameSessionService.CreateGameSession(
       invitation.Id, invitedBy, invitedPlayer);
  }
  return View(session);
}
```

22. Popraw metodę `SetPosition` klasy `GameSessionController`, przekazując w niej zamiast właściwości `turn.User.Email` obiekt klasy `turn.User`:

```
gameSession = await _gameSessionService.AddTurn(gameSession.Id,
 turn.User, turn.X, turn.Y);
```

23. Zmień metodę `OnModelCreating` klasy `GameDbContext`, dodając klucz obcy `WinnerId`:

```
...
modelBuilder.Entity(typeof(GameSessionModel))
  .HasOne(typeof(UserModel), "Winner")
  .WithMany()
  .HasForeignKey("WinnerId").OnDelete(DeleteBehavior.Restrict);
...
```

24. Popraw metodę `GameInvitationConfirmation` klasy `GameInvitationController`; aby mogła współpracować z frameworkiem Identity platformy ASP.NET Core 2.0, musi być asynchroniczna:

```
[HttpGet]
public async Task<IActionResult> GameInvitationConfirmation(
 Guid id, [FromServices]IGameInvitationService
```

```
 gameInvitationService)
{
  return await Task.Run(() =>
  {
    var gameInvitation = gameInvitationService.Get(id).Result;
    return View(gameInvitation);
  });
}
```

1. Popraw metody `Index` i `SetCulture` klasy `HomeController`; aby mogły współpracować z frameworkiem Identity platformy ASP.NET Core 2.0, muszą być asynchroniczne:

```
public async Task<IActionResult> Index()
{
  return await Task.Run(() =>
  {
    var culture =
      Request.HttpContext.Session.GetString("culture");
    ViewBag.Language = culture;
    return View();
  });
}

public async Task<IActionResult> SetCulture(string culture)
{
  return await Task.Run(() =>
  {
    Request.HttpContext.Session.SetString("culture", culture);
    return RedirectToAction("Index");
  });
}
```

1. Popraw metodę `Index` klasy `UserRegistrationController`; aby mogła współpracować z frameworkiem Identity platformy ASP.NET Core 2.0, musi być asynchroniczna:

```
public async Task<IActionResult> Index()
{
  return await Task.Run(() =>
  {
    return View();
  });
}
```

27. Otwórz konsolę menedżera pakietów i wykonaj polecenie `Add-Migration IdentityDb`.
28. Zaktualizuj bazę danych, wykonując w konsoli menedżera pakietów polecenie `Update-Database`.
29. Uruchom aplikację i zarejestruj użytkownika, a potem sprawdź, czy wszystko działa, jak powinno.

> Aby przejść pomyślnie rejestrację użytkownika, musisz teraz użyć złożonego hasła w stylu `Azerty123!`, ponieważ zaimplementowałeś funkcje wchodzące w skład frameworka Identity platformy ASP.NET Core 2.0, które wymagają złożonych haseł.

Wprowadzanie podstawowego uwierzytelnienia za pomocą formularza

Wspaniale! Zarejestrowałeś oprogramowanie pośredniczące uwierzytelniania i przygotowałeś bazę danych. Kolejnym krokiem będzie implementacja w aplikacji *Kółko i krzyżyk* podstawowego uwierzytelniania.

Poniższy przykład demonstruje, jak zmodyfikować rejestrację użytkowników oraz jak w celu ich uwierzytelniania dodać prosty formularz logowania z polami nazwy użytkownika oraz hasła:

1. Do folderu *Models* dodaj klasę `LoginModel`:

```
public class LoginModel
{
  [Required]
  public string UserName { get; set; }
  [Required]
  public string Password { get; set; }
  public string ReturnUrl { get; set; }
}
```

2. Do folderu *Views* dodaj folder *Account*, a w nim plik o nazwie *Login.cshtml*, który będzie zawierał widok logowania:

```
@model TicTacToe.Models.LoginModel
<div class="container">
  <div id="loginbox" style="margin-top:50px;" class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Logowanie</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div style="display:none" id="login-alert"
         class="alert alert-danger col-sm-12"></div>
        <form id="loginform" class="form-horizontal"
         role="form" asp-action="Login" asp-controller="Account">
                          <input type="hidden" asp-for="ReturnUrl" />
          <div asp-validation-summary="ModelOnly"
           class="text-danger"></div>
          <div style="margin-bottom: 25px" class="input-group">
            <span class="input-group-addon"><i class="glyphicon
             glyphicon-user"></i></span>
            <input type="text" class="form-control"
             asp-for="UserName" value="" placeholder="nazwa użytkownika
             ↪lub adres e-mail">
          </div>
          <div style="margin-bottom: 25px" class="input-group">
            <span class="input-group-addon"><i class="glyphicon
             glyphicon-lock"></i></span>
            <input type="password" class="form-control"
```

```
            asp-for="Password" placeholder="hasło">
          </div>
          <div style="margin-top:10px" class="form-group">
            <div class="col-sm-12 controls">
              <button type="submit" id="btn-login" href="#"
               class="btn btn-success">Zaloguj się</button>
            </div>
          </div>
          <div class="form-group">
            <div class="col-md-12 control">
              <div style="border-top: 1px solid#888;
               padding-top:15px; font-size:85%">
                Nie masz konta?
                <a asp-action="Index"
                 asp-controller="UserRegistration">Załóż je tutaj
                </a>
              </div>
            </div>
          </div>
        </form>
      </div>
    </div>
  </div>
</div>
```

3. W klasie `UserService` dodaj prywatne pole `SignInManager` i zaktualizuj konstruktor:

```
...
private SignInManager<UserModel> _signInManager;
public UserService(ApplicationUserManager userManager,
 ILogger<UserService> logger, SignInManager<UserModel>
 signInManager)
{
  ...
  _signInManager = signInManager;
  ...
}
...
```

4. Do klasy `UserService` dodaj dwie metody o nazwach `SignInUser` i `SignOutUser` oraz zaktualizuj interfejs `IUserService`:

```
public async Task<SignInResult> SignInUser(
 LoginModel loginModel, HttpContext httpContext)
{
  var start = DateTime.Now;
  _logger.LogTrace($"logowanie użytkownika {loginModel.UserName}");

  var stopwatch = new Stopwatch();
  stopwatch.Start();

  try
  {
    var user =
      await _userManager.FindByNameAsync(loginModel.UserName);
```

```
        var isValid =
          await _signInManager.CheckPasswordSignInAsync(user,
           loginModel.Password, true);
        if (!isValid.Succeeded)
        {
          return SignInResult.Failed;
        }

        if (!await _userManager.IsEmailConfirmedAsync(user))
        {
          return SignInResult.NotAllowed;
        }

        var identity = new ClaimsIdentity(
          CookieAuthenticationDefaults.AuthenticationScheme);
        identity.AddClaim(new Claim(
          ClaimTypes.Name, loginModel.UserName));
        identity.AddClaim(new Claim(
          ClaimTypes.GivenName, user.FirstName));
        identity.AddClaim(new Claim(
          ClaimTypes.Surname, user.LastName));
        identity.AddClaim(new Claim(
          "displayName", $"{user.FirstName} {user.LastName}"));
        if (!string.IsNullOrEmpty(user.PhoneNumber))
        {
          identity.AddClaim(new Claim(ClaimTypes.HomePhone,
           user.PhoneNumber));
        }

        identity.AddClaim(new Claim("Score",
         user.Score.ToString()));

        await httpContext.SignInAsync(
         CookieAuthenticationDefaults.AuthenticationScheme,
         new ClaimsPrincipal(identity),
         new AuthenticationProperties { IsPersistent = false });

        return isValid;
      }
      catch (Exception ex)
      {
        _logger.LogError($"nie można zalogować użytkownika {loginModel.UserName}
        ↪- {ex}");
        throw ex;
      }
      finally
      {
        stopwatch.Stop();
        _logger.LogTrace($"logowanie użytkownika {loginModel.UserName} ukończono
         ↪w czasie {stopwatch.Elapsed}");
      }
    }
```

```
public async Task SignOutUser(HttpContext httpContext)
{
  await _signInManager.SignOutAsync();
  await httpContext.SignOutAsync(new AuthenticationProperties {
   IsPersistent = false });
  return;
}
```

5. Do folderu *Controllers* dodaj klasę `AccountController` i zaimplementuj trzy nowe metody do obsługi uwierzytelniania użytkownika:

```
public class AccountController : Controller
{
  private IUserService _userService;
  public AccountController(IUserService userService)
  {
    _userService = userService;
  }

  public async Task<IActionResult> Login(string returnUrl)
  {
    return await Task.Run(() =>
    {
      var loginModel = new LoginModel { ReturnUrl = returnUrl };
      return View(loginModel);
    });
  }

  [HttpPost]
  public async Task<IActionResult> Login(LoginModel loginModel)
  {
    if (ModelState.IsValid)
    {
      var result = await _userService.SignInUser(loginModel,
       HttpContext);
      if (result.Succeeded)
      {
        if (!string.IsNullOrEmpty(loginModel.ReturnUrl))
            return Redirect(loginModel.ReturnUrl);
        else
            return RedirectToAction("Index", "Home");
      }
      else
        ModelState.AddModelError("", result.IsLockedOut ?
         "Użytkownik jest zablokowany" : "Użytkownik nie ma prawa dostępu");
    }
    return View();
  }

  public IActionResult Logout()
  {
    _userService.SignOutUser(HttpContext).Wait();
    HttpContext.Session.Clear();
```

```
      return RedirectToAction("Index", "Home");
    }
  }
```

6. Zmień metodę CheckGameSessionIsFinished klasy GameSessionController:

```
[HttpGet("/restapi/v1/CheckGameSessionIsFinished/{sessionId}")]
public async Task<IActionResult> CheckGameSessionIsFinished(
 Guid sessionId)
{
  if (sessionId != Guid.Empty)
  {
    var session =
      await _gameSessionService.GetGameSession(sessionId);
    if (session != null)
    {
      if (session.Turns.Count() == 9)
        return Ok("Gra zakończyła się remisem.");

      var userTurns = session.Turns.Where(
        x => x.User.Id == session.User1.Id).ToList();
      var user1Won = CheckIfUserHasWon(session.User1?.Email,
        userTurns);
      if (user1Won)
      {
        return Ok($"{session.User1.Email} wygrał grę.");
      }
      else
      {
        userTurns = session.Turns.Where(
          x => x.User.Id == session.User2.Id).ToList();
        var user2Won = CheckIfUserHasWon(session.User2?.Email,
         userTurns);

        if (user2Won)
           return Ok($"{session.User2.Email} wygrał grę.");
        else
           return Ok("");
      }
    }
    else
    {
      return NotFound($"Nie można odnaleźć rozgrywki {sessionId}.");
    }
  }
  else
  {
    return BadRequest("Identyfikator SessionId jest pusty.");
  }
}
```

7. Zmień blok kodu na początku pliku *Views/Shared/_Menu.cshtml*:

```
@using Microsoft.AspNetCore.Http;
@{
  var email = User?.Identity?.Name ??
   Context.Session.GetString("email");
  var displayName = User.Claims.FirstOrDefault(
   x => x.Type == "displayName")?.Value ??
   Context.Session.GetString("displayName");
}
```

8. Zmień plik *Views/Shared/_Menu.cshtml*, aby uwierzytelnionym użytkownikom pokazać element nazwy wyświetlanej, a nieuwierzytelnionym element logowania; w tym celu zmień zawartość ostatniego znacznika `<li>`:

```
<li>
  @if (!string.IsNullOrEmpty(email))
  {
    Html.RenderPartial("_Account",
     new TicTacToe.Models.AccountModel { Email = email,
     DisplayName = displayName });
  }
  else
  {
    <a asp-area="" asp-controller="Account"
     asp-action="Login">Logowanie</a>
  }
</li>
```

9. Popraw plik *Views/Shared/_Account.cshtml*, zmieniając linki *Wyloguj się* i *Zobacz szczegóły*:

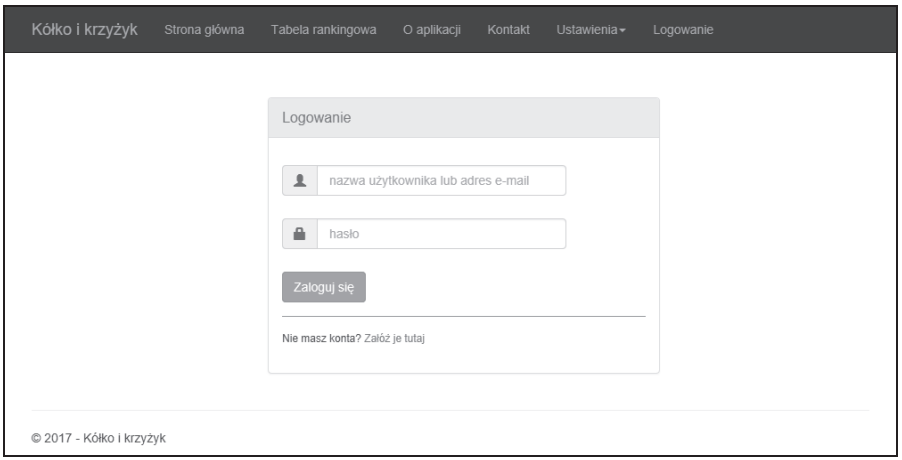
```
<a class="btn btn-danger btn-block" asp-controller="Account"
asp-action="Logout" asp-area="">Wyloguj się</a>
<a class="btn btn-default btn-block" asp-action="Index"
asp-controller="Home" asp-area="Account">Zobacz szczegóły</a>
```

10. Przejdź do folderu *Views\Shared\Components\GameSession* i zmień plik *default.cshtml*, aby poprawić prezentację wizualną:

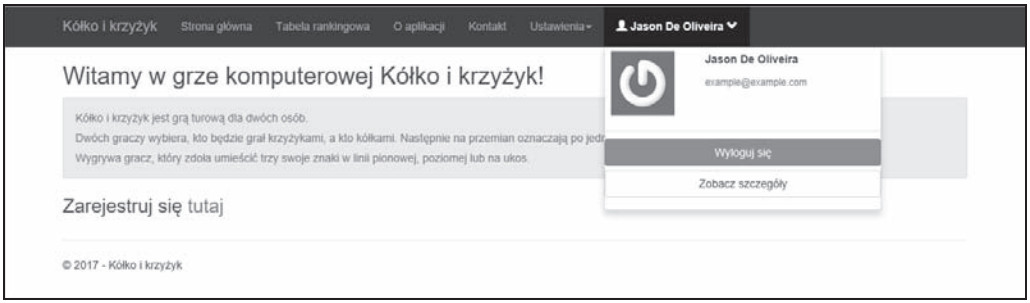
```
@using Microsoft.AspNetCore.Http
@model TicTacToe.Models.GameSessionModel
@{
  var email = Context.Session.GetString("email");
}
<div id="gameBoard">
  <table>
    @for (int rows = 0; rows < 3; rows++)
    {
      <tr style="height:150px;">
        @for (int columns = 0; columns < 3; columns++)
        {
          <td style="width:150px; border:1px solid #808080;
           text-align:center; vertical-align:middle"
           id="@($"c_{rows}_{columns}")">
            @{
```

```
                var position = Model.Turns?.FirstOrDefault(
                 turn => turn.X == columns && turn.Y == rows);
                if (position != null)
                {
                  if (position.User == Model.User1)
                  {
                    <i class="glyphicon glyphicon-unchecked"></i>
                  }
                  else
                  {
                    <i class="glyphicon glyphicon-remove-circle"></i>
                  }
                }
                else
                {
                  <a class="btn btn-default btn-SetPosition"
                   style="width:150px; min-height:150px;"
                   data-X="@columns" data-Y="@rows">
                     
                   </a>
                 }
              }
            </td>
          }
        </tr>
      }
    </table>
  </div>
  <div class="alert" id="divAlertWaitTurn">
    <i class="glyphicon glyphicon-alert">Proszę czekać, aż
       drugi użytkownik skończy kolejkę.</i>
  </div>
```

11. Uruchom aplikację, kliknij pozycję *Zaloguj się* w głównym menu i zaloguj się na konto istniejącego użytkownika (albo zarejestruj użytkownika, jeśli tego wcześniej nie zrobiłeś):



12. Kliknij przycisk *Wyloguj się*. Powinieneś zostać wylogowany i przekierowany z powrotem na stronę główną:



Wprowadzanie uwierzytelniania za pośrednictwem zewnętrznego dostawcy

W kolejnym punkcie zaprezentujemy uwierzytelnianie za pośrednictwem zewnętrznego dostawcy, używając do tego serwisu Facebook.

Oto przegląd przepływu sterowania w takim wypadku:

1. Użytkownik klika specjalny przycisk logowania do zewnętrznego dostawcy.
2. Odpowiedni kontroler otrzymuje żądanie wskazujące, który dostawca jest potrzebny, a następnie do tego dostawcy kierowane jest pytanie (ang. *challenge*).
3. Zewnętrzny dostawca wysyła do aplikacji zwrotne żądanie HTTP (typu POST lub GET) z nazwą dostawcy, kluczem oraz pewnymi oświadczeniami użytkownika.
4. Aplikacja sprawdza, czy te oświadczenia pasują do któregoś z wewnętrznych użytkowników.
5. Jeśli oświadczenia nie pasują do żadnego użytkownika wewnętrznego, użytkownik jest przekierowywany do odpowiedniego formularza rejestracji albo jest odrzucany.

> Etapy implementacji dla wszystkich zewnętrznych dostawców, którzy wspierają interfejs OWIN i framework Identity platformy ASP.NET Core 2.0, są takie same. Możesz nawet stworzyć własnego dostawcę i zintegrować go w ten sam sposób.

Zaimplementujemy teraz uwierzytelnianie za pośrednictwem zewnętrznego dostawcy na przykładzie serwisu Facebook:

6. Zmień formularz logowania, dodając bezpośrednio po standardowym przycisku logowania przycisk *Zaloguj się przez Facebook*:

```
<a id="btn-fblogin" asp-action="ExternalLogin"
 asp-controller="Account" asp-route-Provider="Facebook"
 class="btn btn-primary">Zaloguj się przez Facebook</a>
```

13. Zmień klasę `UserService` oraz interfejs `IUserService`, dodając trzy metody o nazwach `GetExternalAuthenticationProperties`, `GetExternalLoginInfoAsync` i `ExternalLogin`
```
↪SignInAsync:
```

```
public async Task<AuthenticationProperties>
 GetExternalAuthenticationProperties(string provider,
 string redirectUrl)
{
  return await Task.FromResult(
   _signInManager.ConfigureExternalAuthenticationProperties(
   provider, redirectUrl));
}

public async Task<ExternalLoginInfo> GetExternalLoginInfoAsync()
{
  return await _signInManager.GetExternalLoginInfoAsync();
}

public async Task<SignInResult> ExternalLoginSignInAsync(
 string loginProvider, string providerKey, bool isPersistent)
{
  _logger.LogInformation($"Logowanie użytkownika przez zewnętrzny serwis
{loginProvider} - {providerKey}");
  return await _signInManager.ExternalLoginSignInAsync(
   loginProvider, providerKey, isPersistent);
}
```

14. Zmień klasę `AccountController`, dodając dwie metody o nazwach `ExternalLogin` i `ExternalLoginCallBack`:

```
[AllowAnonymous]
public async Task<ActionResult> ExternalLogin(string provider,
 string ReturnUrl)
{
  var redirectUrl = Url.Action(nameof(ExternalLoginCallBack),
   "Account", new { ReturnUrl = ReturnUrl }, Request.Scheme,
    Request.Host.ToString());
  var properties =
    await _userService.GetExternalAuthenticationProperties(
     provider, redirectUrl);
  ViewBag.ReturnUrl = redirectUrl;
  return Challenge(properties, provider);
}

[AllowAnonymous]
public async Task<IActionResult> ExternalLoginCallBack(
 string returnUrl, string remoteError = null)
{
  if (remoteError != null)
  {
    ModelState.AddModelError(string.Empty, $"Błąd zewnętrznego dostawcy:
    ↪{remoteError}");
    ViewBag.ReturnUrl = returnUrl;
```

```
      return View("Login");
    }

    var info = await _userService.GetExternalLoginInfoAsync();
    if (info == null)
    {
      return RedirectToAction("Login",
       new { ReturnUrl = returnUrl });
    }

    var result =
     await _userService.ExternalLoginSignInAsync(
      info.LoginProvider, info.ProviderKey, isPersistent: false);
    if (result.Succeeded)
    {
      if (!string.IsNullOrEmpty(returnUrl))
        return Redirect(returnUrl);
      else
        return RedirectToAction("Index", "Home");
    }
    if (result.IsLockedOut)
    {
      return View("Lockout");
    }
    else
    {
      return View("NotFound");
    }
  }
```

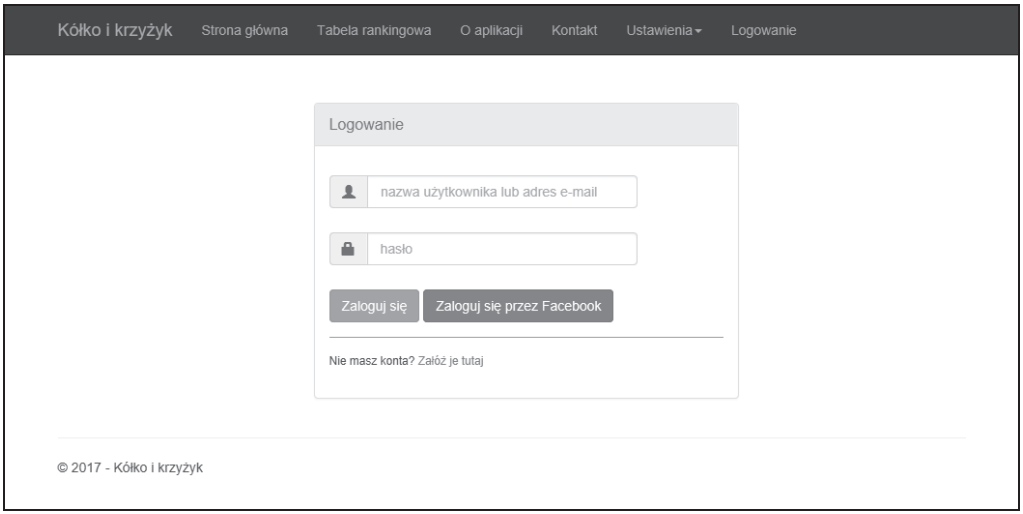
1. W klasie `Startup` zarejestruj oprogramowanie pośredniczące serwisu Facebook:

```
services.AddAuthentication(options => {
  options.DefaultScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultSignInScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
}).AddCookie().AddFacebook(facebook =>
{
  facebook.AppId = "123";
  facebook.AppSecret = "123";
  facebook.ClientId = "123";
  facebook.ClientSecret = "123";
});
```

Zanim będziesz mógł uwierzytelniać logowania za pomocą serwisu Facebook, musisz zaktualizować konfigurację jego oprogramowania pośredniczącego oraz zarejestrować aplikację na portalu programistów serwisu Facebook.

Więcej informacji (w języku angielskim) dostępnych jest na stronie *http://developer.facebook.com*.

16. Uruchom aplikację, kliknij przycisk *Zaloguj się przez Facebook*, wprowadź swoje dane uwierzytelniające do tego serwisu i sprawdź, czy wszystko działa tak, jak powinno:



Praca z uwierzytelnianiem dwuskładnikowym

Standardowe mechanizmy bezpieczeństwa, które widziałeś wcześniej, wymagają tylko prostej nazwy użytkownika i hasła, co w znacznym stopniu ułatwia cyberprzestępcom uzyskanie dostępu do poufnych informacji, w tym szczegółowych danych osobistych i finansowych, w drodze złamania hasła albo przejęcia danych uwierzytelniających użytkownika [za pomocą wiadomości e-mail, szperania po sieci (ang. *network sniffing*) itp.]. Te dane bywają potem używane do popełniania oszustw internetowych i kradzieży tożsamości.

Uwierzytelnianie dwuskładnikowe wprowadza dodatkowy poziom zabezpieczeń, ponieważ wymaga nie tylko nazwy użytkownika i hasła, ale także kodu zabezpieczającego (wygenerowanego przez fizyczne urządzenie, program itd.), który może podać jedynie użytkownik. To bardzo utrudnia dostęp potencjalnym intruzom i zabezpiecza przed kradzieżą tożsamości i danych.

Wszystkie większe strony internetowe zapewniają opcję uwierzytelniania dwuskładnikowego, dodajmy więc ją również do aplikacji *Kółko i krzyżyk*:

1. Dodaj model `TwoFactorCodeModel` do folderu *Models*:

```
public class TwoFactorCodeModel
{
  [Key]
  public long Id { get; set; }
  public Guid UserId { get; set; }
  [ForeignKey("UserId")]
  public UserModel User { get; set; }
```

```
  public string TokenProvider { get; set; }
  public string TokenCode { get; set; }
}
```

2. Dodaj model `TwoFactorEmailModel` do folderu *Models*:

```
public class TwoFactorEmailModel
{
  public string DisplayName { get; set; }
  public string Email { get; set; }
  public string ActionUrl { get; set; }
}
```

1. Zarejestruj model `TwoFactorCodeModel` w kontekście bazy `GameDbContext`, dodając instancję odpowiedniego typu `DbSet`:

```
public DbSet<TwoFactorCodeModel> TwoFactorCodeModels { get; set; }
```

4. Otwórz konsolę menedżera pakietów NuGet i wykonaj polecenie `Add-Migration AddTwoFactorCode`, a potem zaktualizuj bazę danych poleceniem `Update-Database`.
5. Do klasy `ApplicationUserManager` dodaj trzy metody o nazwach `SetTwoFactorEnabledAsync`, `GenerateTwoFactorTokenAsync` i `VerifyTwoFactorTokenAsync`:

```
public override async Task<IdentityResult>
 SetTwoFactorEnabledAsync(UserModel user, bool enabled)
{
  try
  {
    using (var db = new GameDbContext(_dbContextOptions))
    {
      var current = await db.UserModels.FindAsync(user.Id);
      current.TwoFactorEnabled = enabled;
      await db.SaveChangesAsync();
      return IdentityResult.Success;
    }
  }
  catch (Exception ex)
  {
    return IdentityResult.Failed(new IdentityError {
     Description = ex.ToString() });
  }
}

public override async Task<string>
 GenerateTwoFactorTokenAsync(UserModel user,
 string tokenProvider)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
    var emailTokenProvider = new EmailTokenProvider<UserModel>();
    var token = await emailTokenProvider.GenerateAsync(
     "TwoFactor", this, user);
    dbContext.TwoFactorCodeModels.Add(new TwoFactorCodeModel
    {
```

```
        TokenCode = token,
        TokenProvider = tokenProvider,
        UserId = user.Id
      });

      if (dbContext.ChangeTracker.HasChanges())
        await dbContext.SaveChangesAsync();

      return token;
    }
  }

  public override async Task<bool>
   VerifyTwoFactorTokenAsync(UserModel user,
   string tokenProvider, string token)
  {
    using (var dbContext = new GameDbContext(_dbContextOptions))
    {
      return await dbContext.TwoFactorCodeModels.AnyAsync(
       x => x.TokenProvider == tokenProvider &&
       x.TokenCode == token && x.UserId == user.Id);
    }
  }
```

1. Przejdź do folderu *Areas/Account/Views/Home* i zmień domyślny widok:

```
@model TicTacToe.Models.UserModel
@using Microsoft.AspNetCore.Identity
@inject UserManager<TicTacToe.Models.UserModel> UserManager
@{
  var isTwoFactor =
    UserManager.GetTwoFactorEnabledAsync(Model).Result;
  ViewData["Title"] = "Indeks";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<h3>Szczegóły konta</h3>
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-6 col-md-6">
      <div class="well well-sm">
        <div class="row">
          <div class="col-sm-6 col-md-4">
            <Gravatar email="@Model.Email"></Gravatar>
          </div>
          <div class="col-sm-6 col-md-8">
            <h4>@($"{Model.FirstName} {Model.LastName}")</h4>
             <p>
               <i class="glyphicon glyphicon-envelope">
               </i> <a href="mailto:@Model.Email">
                @Model.Email</a>
             </p>
             <p>
               <i class="glyphicon glyphicon-calendar">
```

```
                </i> @Model.EmailConfirmationDate
              </p>
              <p>
                <i class="glyphicon glyphicon-star">
                </i> @Model.Score
              </p>
              <p>
                <i class="glyphicon glyphicon-check"></i>
                 <text>Uwierzytelnianie dwuskładnikowe </text>
                @if (Model.TwoFactorEnabled)
                {
                  <a asp-action="DisableTwoFactor">Wyłącz</a>
                }
                else
                {
                  <a asp-action="EnableTwoFactor">Włącz</a>
                }
              </p>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
```

7. Do folderu *Areas/Account/Views* dodaj plik *_ViewImports.cshtml*:

```
@using TicTacToe
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
@addTagHelper *, TicTacToe
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

8. Zmień klasę `UserService` i interfejs `IUserService`, dodając dwie metody o nazwach `EnableTwoFactor` i `GetTwoFactorCode`:

```
public async Task<IdentityResult> EnableTwoFactor(string name,
 bool enabled)
{
  try
  {
    var user = await _userManager.FindByEmailAsync(name);
    user.TwoFactorEnabled = true;
    await _userManager.SetTwoFactorEnabledAsync(user, enabled);
    return IdentityResult.Success;
  }
  catch (Exception ex)
  {
    throw;
  }
}

public async Task<string> GetTwoFactorCode(string userName,
 string tokenProvider)
```

```
{
  var user = await GetUserByEmail(userName);
  return await _userManager.GenerateTwoFactorTokenAsync(user,
   tokenProvider);
}
```

9. Zmień metodę `SignInUser` klasy `UserService` tak, by obsługiwała uwierzytelnianie dwuskładnikowe, jeśli jest włączone:

```
public async Task<SignInResult> SignInUser(LoginModel
 loginModel, HttpContext httpContext)
{
  var start = DateTime.Now;
  _logger.LogTrace($"Logowanie użytkownika {loginModel.UserName}");
  var stopwatch = new Stopwatch();
  stopwatch.Start();
  try
  {
    var user =
      await _userManager.FindByNameAsync(loginModel.UserName);
    var isValid =
      await _signInManager.CheckPasswordSignInAsync(user,
       loginModel.Password, true);
    if (!isValid.Succeeded)
    {
      return SignInResult.Failed;
    }
    if (!await _userManager.IsEmailConfirmedAsync(user))
    {
      return SignInResult.NotAllowed;
    }
    if (await _userManager.GetTwoFactorEnabledAsync(user))
    {
      return SignInResult.TwoFactorRequired;
    }
    var identity = new ClaimsIdentity(
      CookieAuthenticationDefaults.AuthenticationScheme);
    identity.AddClaim(new Claim(ClaimTypes.Name,
      loginModel.UserName));
    identity.AddClaim(new Claim(ClaimTypes.GivenName,
      user.FirstName));
    identity.AddClaim(new Claim(ClaimTypes.Surname,
      user.LastName));
    identity.AddClaim(new Claim("displayName",
      $"{user.FirstName} {user.LastName}"));
    if (!string.IsNullOrEmpty(user.PhoneNumber))
    {
      identity.AddClaim(new Claim(ClaimTypes.HomePhone,
       user.PhoneNumber));
    }
    identity.AddClaim(new Claim("Score",
     user.Score.ToString()));
    await httpContext.SignInAsync(
     CookieAuthenticationDefaults.AuthenticationScheme,
```

```
        new ClaimsPrincipal(identity),
        new AuthenticationProperties { IsPersistent = false });
      return isValid;
    }
    catch (Exception ex)
    {
      _logger.LogError($"Nie można zalogować użytkownika {loginModel.UserName}
      ↪- {ex}");
      throw ex;
    }
    finally
    {
      stopwatch.Stop();
      _logger.LogTrace($"Logowanie użytkownika {loginModel.UserName} ukończono
      ↪w czasie {stopwatch.Elapsed}");
    }
  }
```

10. Przejdź do folderu *Areas/Account/Controllers* i zmień klasę `HomeController`. Popraw jej metodę `Index` i dodaj dwie nowe o nazwach `EnableTwoFactor` i `DisableTwoFactor`:

```
[Authorize]
public async Task<IActionResult> Index()
{
  var user =
    await _userService.GetUserByEmail(User.Identity.Name);
  return View(user);
}

[Authorize]
public IActionResult EnableTwoFactor()
{
  _userService.EnableTwoFactor(User.Identity.Name, true);
  return RedirectToAction("Index");
}

[Authorize]
public IActionResult DisableTwoFactor()
{
  _userService.EnableTwoFactor(User.Identity.Name, false);
  return RedirectToAction("Index");
}
```

> Atrybut `[Authorize]` objaśnimy w dalszej części rozdziału. Używa się go do wprowadzenia ograniczeń dostępu do zasobów.

11. Do folderu *Models* dodaj klasę `ValidateTwoFactorModel`:

```
public class ValidateTwoFactorModel
{
  public string UserName { get; set; }
  public string Code { get; set; }
}
```

12. Do klasy `AccountController` dodaj metodę `SendEmailTwoFactor`:

```
private async Task SendEmailTwoFactor(string UserName)
{
  var user = await _userService.GetUserByEmail(UserName);
  var urlAction = new UrlActionContext
  {
    Action = "ValidateTwoFactor",
    Controller = "Account",
    Values = new { email = UserName,
    code = await _userService.GetTwoFactorCode(
     user.UserName, "Email") },
    Protocol = Request.Scheme,
    Host = Request.Host.ToString()
  };

  var TwoFactorEmailModel = new TwoFactorEmailModel
  {
    DisplayName = $"{user.FirstName} {user.LastName}",
     Email = UserName,
     ActionUrl = Url.Action(urlAction)
  };
  var emailRenderService =
    HttpContext.RequestServices.GetService
     <IEmailTemplateRenderService>();
  var emailService =
    HttpContext.RequestServices.GetService
     <IEmailService>();
  var message =
    await emailRenderService.RenderTemplate(
     "EmailTemplates/TwoFactorEmail", TwoFactorEmailModel,
      Request.Host.ToString());
  try
  {
    emailService.SendEmail(UserName, "Kod zabezpieczający aplikacji Kółko
    ↪i krzyżyk", message).Wait();
  }
  catch
  {
  }
}
```

Aby wywołać metodę `RequestServices.GetService<T>();`, tak jak w poprzednich przykładach, należy dodać instrukcję `using Microsoft.Extensions.DependencyInjection;`.

13. Zmień metodę `Login` klasy `AccountController`:

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel loginModel)
{
  if (ModelState.IsValid)
```

```
    {
      var result = await _userService.SignInUser(loginModel,
       HttpContext);
      if (result.Succeeded)
      {
        if (!string.IsNullOrEmpty(loginModel.ReturnUrl))
          return Redirect(loginModel.ReturnUrl);
        else
          return RedirectToAction("Index", "Home");
      }
      else if (result.RequiresTwoFactor)
      {
        await SendEmailTwoFactor(loginModel.UserName);
        return RedirectToAction("ValidateTwoFactor");
      }
      else
        ModelState.AddModelError("", result.IsLockedOut ? "Użytkownik jest
        ↪zablokowany" : "Użytkownik nie jest upoważniony");
    }

    return View();
  }
```

14. Do folderu *Views/Account* dodaj widok o nazwie *ValidateTwoFactor*:

```
@model TicTacToe.Models.ValidateTwoFactorModel
@{
  ViewData["Title"] = "Weryfikacja dwuskładnikowa";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<div class="container">
  <div id="loginbox" style="margin-top:50px;" class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Weryfikacja kodu zabezpieczającego</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div class="text-center">
          <form asp-controller="Account"
           asp-action="ValidateTwoFactor" method="post">
            <div asp-validation-summary="All"></div>
            <div style="margin-bottom: 25px" class="input-group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-envelope
               color-blue"></i></span>
              <input id="email" asp-for="UserName"
               placeholder="adres e-mail"
               class="form-control" type="email">
            </div>
            <div style="margin-bottom: 25px" class="input-group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-lock
```

```
                color-blue"></i></span>
              <input id="Code" asp-for="Code"
                placeholder="Wprowadź kod" class="form-control">
            </div>
            <div style="margin-bottom: 25px" class="input-group">
              <input name="submit"
                class="btn btn-lg btn-primary btn-block"
                value="Zweryfikuj kod" type="submit">
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
```

15. Do folderu *Views/EmailTemplates* dodaj widok o nazwie *TwoFactorEmail*:

```
@model TicTacToe.Models.TwoFactorEmailModel
@{
  ViewData["Title"] = "Widok";
  Layout = "_LayoutEmail";
}
<h1>Witaj, @Model.DisplayName</h1>
Chcesz otrzymać kod zabezpieczający? Aby kontynuować, kliknij <a
 href="@Model.ActionUrl">tutaj</a>.
```

16. Zmień klasę `UserService` i interfejs `IUserService`, dodając metodę `ValidateTwoFactor`:

```
public async Task<bool> ValidateTwoFactor(string userName,
 string tokenProvider, string token, HttpContext httpContext)
{
  var user = await GetUserByEmail(userName);
  if (await _userManager.VerifyTwoFactorTokenAsync(user,
   tokenProvider, token))
  {
    var identity =
      new ClaimsIdentity(
        CookieAuthenticationDefaults.AuthenticationScheme);
    identity.AddClaim(new Claim(ClaimTypes.Name,
     user.UserName));
    identity.AddClaim(new Claim(ClaimTypes.GivenName,
     user.FirstName));
    identity.AddClaim(new Claim(ClaimTypes.Surname,
     user.LastName));
    identity.AddClaim(new Claim("displayName",
     $"{user.FirstName} {user.LastName}"));

    if (!string.IsNullOrEmpty(user.PhoneNumber))
    {
      identity.AddClaim(new Claim(ClaimTypes.HomePhone,
       user.PhoneNumber));
    }
```

```
      identity.AddClaim(new Claim("Score",
       user.Score.ToString()));
      await httpContext.SignInAsync(
       CookieAuthenticationDefaults.AuthenticationScheme,
       new ClaimsPrincipal(identity),
       new AuthenticationProperties { IsPersistent = false });

      return true;
    }
    return false;
  }
```

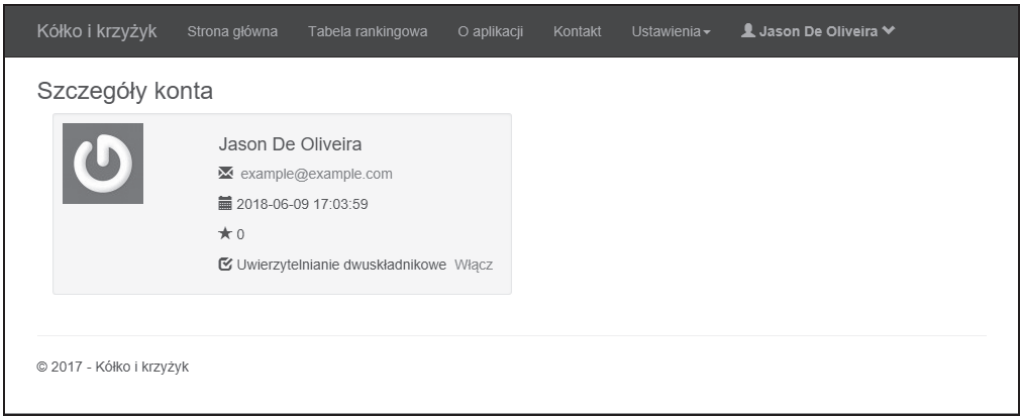
17. Zmień klasę `AccountController`, dodając dwie nowe metody do weryfikacji uwierzytelniania dwuskładnikowego:

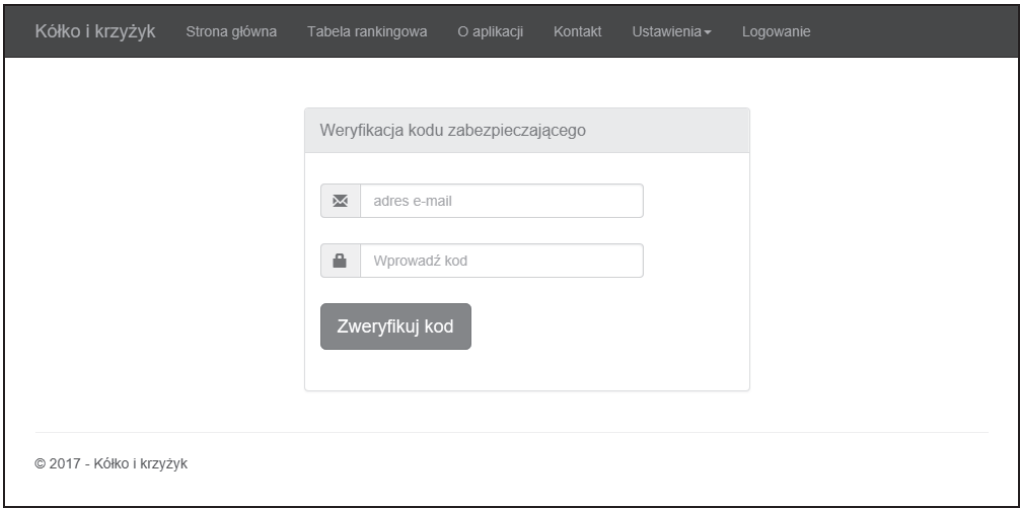
```
public async Task<IActionResult> ValidateTwoFactor(
 string email, string code)
{
  return await Task.Run(() =>
  {
    return View(new ValidateTwoFactorModel { Code = code,
     UserName = email });
  });
}

[HttpPost]
public async Task<IActionResult> ValidateTwoFactor(
 ValidateTwoFactorModel validateTwoFactorModel)
{
  if (ModelState.IsValid)
  {
    await _userService.ValidateTwoFactor(
     validateTwoFactorModel.UserName, "Email",
     validateTwoFactorModel.Code, HttpContext);
    return RedirectToAction("Index", "Home");
  }

  return View();
}
```

1. Uruchom aplikację, zaloguj się jako istniejący użytkownik i przejdź do strony szczegółów konta. Włącz uwierzytelnianie dwuskładnikowe (być może wcześniej będziesz musiał odtworzyć bazę danych i zarejestrować użytkownika) (patrz pierwszy rysunek na następnej stronie).

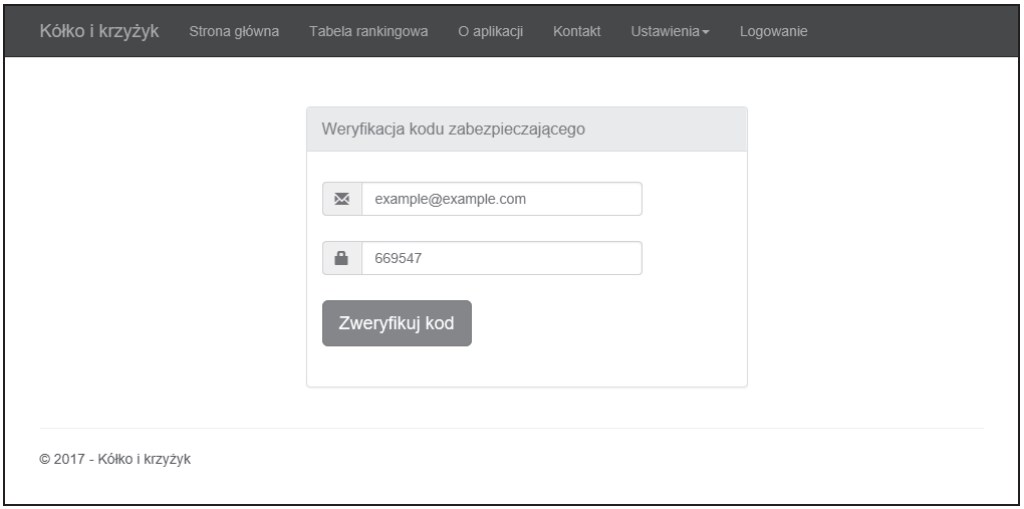
19. Wyloguj się z konta, przejdź do strony logowania i zaloguj się ponownie. Tym razem zostaniesz poproszony o wprowadzenie kodu uwierzytelniania dwuskładnikowego (patrz drugi rysunek na następnej stronie).

20. Otrzymasz wiadomość z **kodem uwierzytelniania dwuskładnikowego** (patrz trzeci rysunek na następnej stronie).





21. Kliknij link zawarty w wiadomości e-mail, a wszystko powinno zostać automatycznie wypełnione. Zaloguj się i sprawdź, czy wszystko działa tak, jak powinno:



Dodawanie mechanizmu resetowania zapomnianego hasła

Teraz, gdy już zobaczyłeś, jak dodać do aplikacji uwierzytelnianie, musisz pomyśleć o tym, w jaki sposób pomóc użytkownikom zresetować zapomniane przez nich hasła. Niektórzy użytkownicy z pewnością je zapomną, więc musisz mieć jakieś mechanizmy w pogotowiu.

Standardowym sposobem obsługi tego typu żądania jest wysłanie wiadomości e-mail z linkiem do zresetowania hasła. Użytkownik może wtedy zmienić hasło, a my nie ponosimy ryzyka wysyłania go jawnym tekstem w wiadomości e-mail. Wysyłanie hasła bezpośrednio na adres e-mail użytkownika jest niebezpieczne i powinno się tego unikać za wszelką cenę.

Zobaczysz teraz, jak do aplikacji *Kółko i krzyżyk* dodać funkcję resetowania hasła:

1. Zmień formularz logowania, dodając link *Zresetuj je tutaj* zaraz po *Załóż je tutaj*:

```
<div class="col-md-12 control">
  <div style="border-top: 1px solid#888; padding-top:15px;
   font-size:85%">
   Nie masz konta?
   <a asp-action="Index"
    asp-controller="UserRegistration">Załóż je tutaj</a>
</div>
<div style="font-size: 85%;">
  Nie pamiętasz hasła?
  <a asp-action="ForgotPassword">Zresetuj je tutaj</a></div>
</div>
```

2. Do folderu *Models* dodaj klasę `ResetPasswordEmailModel`:

```
public class ResetPasswordEmailModel
{
  public string DisplayName { get; set; }
```

```
  public string Email { get; set; }
  public string ActionUrl { get; set; }
}
```

3. Do klasy `AccountController` dodaj metodę `ForgotPassword`:

```
[HttpGet]
public async Task<IActionResult> ForgotPassword()
{
  return await Task.Run(() =>
  {
    return View();
  });
}
```

4. Do folderu *Models* dodaj klasę `ResetPasswordModel`:

```
public class ResetPasswordModel
{
  public string Token { get; set; }
  public string UserName { get; set; }
  public string Password { get; set; }
  public string ConfirmPassword { get; set; }
}
```

5. Do folderu *Views/Account* dodaj widok o nazwie `ForgotPassword`:

```
@model TicTacToe.Models.ResetPasswordModel
@{
  ViewData["Title"] = "Resetowanie hasła";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<div class="form-gap"></div>
<div class="container">
  <div class="row">
    <div class="col-md-4 col-md-offset-4">
      <div class="panel panel-default">
        <div class="panel-body">
          <div class="text-center">
            <h3><i class="fa fa-lock fa-4x"></i></h3>
            <h2 class="text-center">Nie pamiętasz hasła?</h2>
            <p>Tutaj możesz je zresetować.</p>
            <div class="panel-body">
              <form id="register-form" role="form"
               autocomplete="off" class="form"
               method="post" asp-controller="Account"
               asp-action="SendResetPassword">
                <div class="form-group">
                  <div class="input-group">
                    <span class="input-group-addon"><i
                     class="glyphicon glyphicon-envelope
                     color-blue"></i></span>
                    <input id="email" name="UserName"
                     placeholder="adres e-mail"
                     class="form-control" type="email">
```

```
                  </div>
                </div>
                <div class="form-group">
                  <input name="recover-submit"
                   class="btn btn-lg btn-primary btn-block"
                   value="Zresetuj hasło" type="submit">
                </div>
                <input type="hidden" class="hide"
                 name="token" id="token" value="">
              </form>

            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

6. Zmień klasę `UserService` i interfejs `IUserService`, dodając metodę `GetResetPasswordCode`:

```
public async Task<string> GetResetPasswordCode(UserModel user)
{
  return await _userManager.GeneratePasswordResetTokenAsync(user);
}
```

7. Do folderu *View/EmailTemplates* dodaj widok o nazwie *ResetPasswordEmail*:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "Widok";
  Layout = "_LayoutEmail";
}
<h1>Witaj, @Model.DisplayName</h1>
Chcesz zresetować hasło? Aby kontynuować, kliknij <a
 href="@Model.ActionUrl">tutaj</a>.
```

8. Do klasy `AccountController` dodaj metodę `SendResetPassword`:

```
[HttpPost]
public async Task<IActionResult> SendResetPassword(
 string UserName)
{
  var user = await _userService.GetUserByEmail(UserName);
  var urlAction = new UrlActionContext
  {
    Action = "ResetPassword",
    Controller = "Account",
    Values = new { email = UserName,
     code = await _userService.GetResetPasswordCode(user) },
    Protocol = Request.Scheme,
    Host = Request.Host.ToString()
  };

  var resetPasswordEmailModel = new ResetPasswordEmailModel
```

```
    {
      DisplayName = $"{user.FirstName} {user.LastName}",
      Email = UserName,
      ActionUrl = Url.Action(urlAction)
    };

    var emailRenderService =
      HttpContext.RequestServices.GetService
       <IEmailTemplateRenderService>();
    var emailService =
      HttpContext.RequestServices.GetService<IEmailService>();
    var message =
      await emailRenderService.RenderTemplate(
       "EmailTemplates/ResetPasswordEmail",
        resetPasswordEmailModel,
    Request.Host.ToString());

    try
    {
      emailService.SendEmail(UserName,
       "Resetowanie hasła w aplikacji Kółko i krzyżyk", message).Wait();
    }
    catch
    {
    }

    return View("ConfirmResetPasswordRequest",
     resetPasswordEmailModel);
  }
```

9. Do folderu *Views/Account* dodaj widok o nazwie *ConfirmResetPasswordRequest*:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "Potwierdzenie żądania zresetowania hasła";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
@section Desktop{<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<h1>@Localizer["Chcesz zresetować hasło. Wiadomość e-mail została wysłana na
↪adres {0}. Aby kontynuować, kliknij zawarty w niej link.", Model.Email]</h1>
```

10. Do klasy `AccountController` dodaj metodę `ResetPassword`:

```
public async Task<IActionResult> ResetPassword(string email,
 string code)
{
  var user = await _userService.GetUserByEmail(email);
  ViewBag.Code = code;
  return View(new ResetPasswordModel { Token = code,
   UserName = email });
}
```

11. Dodaj widok o nazwie *SendResetPassword* do folderu *Views/Account*:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "Wysyłanie resetowania hasła";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
@section Desktop{<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<h1>@Localizer["Chcesz zresetować hasło. Na adres {0} została wysłana wiadomość
↪e-mail. Aby kontynuować, kliknij link.",
 Model.Email]</h1>
```

12. Dodaj widok o nazwie *ResetPassword* do folderu *Views/Account*:

```
@model TicTacToe.Models.ResetPasswordModel
@{
  ViewData["Title"] = "Resetowanie hasła";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<div class="container">
  <div id="loginbox" style="margin-top:50px;" class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Zresetuj swoje hasło</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div class="text-center">
          <form asp-controller="Account"
           asp-action="ResetPassword" method="post">
            <input type="hidden" asp-for="Token" />
            <div asp-validation-summary="All"></div>
            <div style="margin-bottom: 25px" class="input-group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-envelope
               color-blue"></i></span>
              <input id="email" asp-for="UserName"
               placeholder="adres e-mail"
               class="form-control" type="email">
            </div>
            <div style="margin-bottom: 25px" class="input-group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-lock
               color-blue"></i></span>
              <input id="password" asp-for="Password"
               placeholder="Hasło"
               class="form-control" type="password">
            </div>
            <div style="margin-bottom: 25px" class="input-group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-lock
               color-blue"></i></span>
```

```
                <input id="confirmpassword"
                 asp-for="ConfirmPassword"
                 placeholder="Potwierdź hasło"
                 class="form-control" type="password">
              </div>
              <div style="margin-bottom: 25px" class="input-group">
                <input name="submit"
                 class="btn btn-lg btn-primary btn-block"
                 value="Zresetuj hasło" type="submit">
              </div>
            </form>
          </div>
        </div>
      </div>
    </div>
  </div>
```

13. Zmień klasę `UserService` oraz interfejs `IUserService`, dodając metodę o nazwie `ResetPassword`:

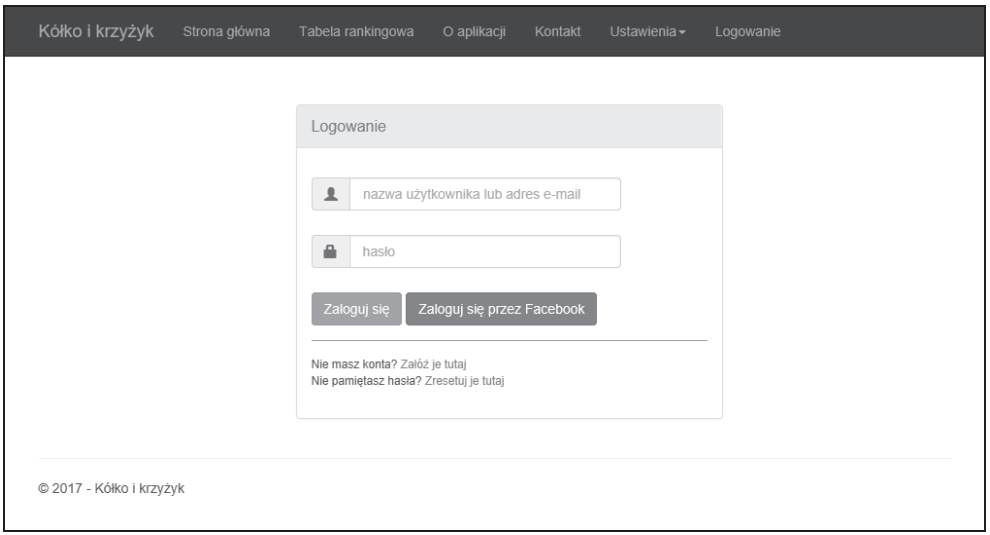
```
public async Task<IdentityResult> ResetPassword(
 string userName, string password, string token)
{
  var start = DateTime.Now;
  _logger.LogTrace($"Resetowanie hasła użytkownika {userName}");

  var stopwatch = new Stopwatch();
  stopwatch.Start();

  try
  {
    var user = await _userManager.FindByNameAsync(userName);
    var result = await _userManager.ResetPasswordAsync(user,
     token, password);
    return result;
  }
  catch (Exception ex)
  {
    _logger.LogError($"Nie można zresetować hasła użytkownika {userName} - {ex}");
    throw ex;
  }
  finally
  {
    stopwatch.Stop();
    _logger.LogTrace($"Resetowanie hasła użytkownika {userName} ukończono w ciągu
    ↪{stopwatch.Elapsed}");
  }
}
```

14. Dodaj metodę `ResetPassword` do klasy `AccountController`:

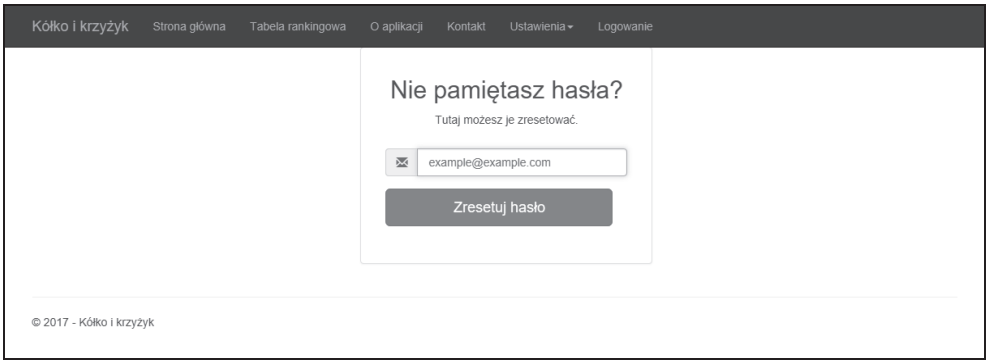
```
[HttpPost]
public async Task<IActionResult> ResetPassword(
 ResetPasswordModel reset)
```

```
{
  if (ModelState.IsValid)
  {
    var result =
      await _userService.ResetPassword(reset.UserName,
       reset.Password, reset.Token);
    if (result.Succeeded)
      return RedirectToAction("Login");
    else
      ModelState.AddModelError("", "Nie można zresetować Twojego hasła");
  }
  return View();
}
```

15. Uruchom aplikację i przejdź do strony logowania. Kliknij link *Zresetuj je tutaj*:

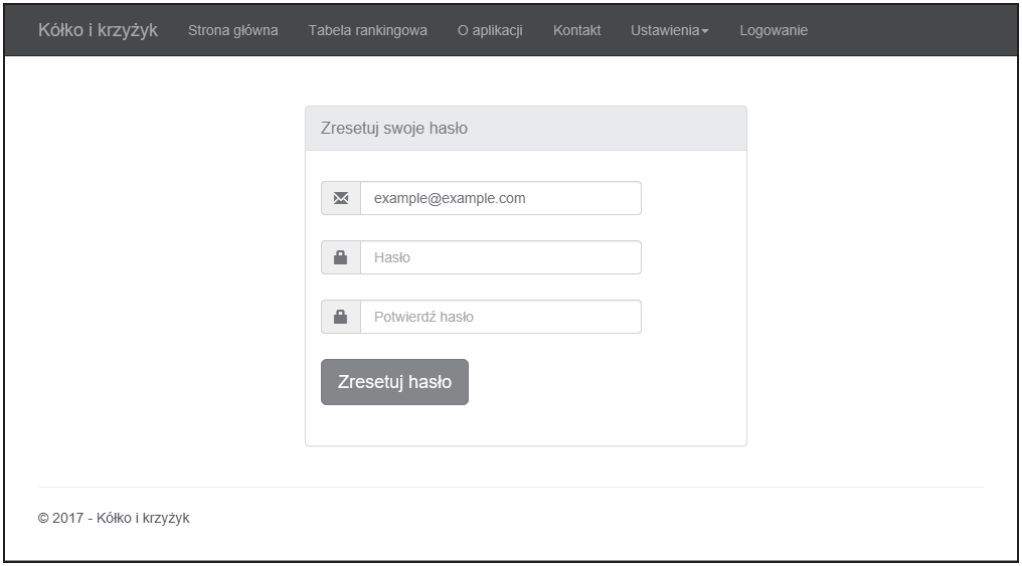


16. Na stronie *Nie pamiętasz hasła?* wprowadź adres e-mail istniejącego użytkownika; zostanie do niego wysłana wiadomość:



17. Otwórz wiadomość e-mail dotyczącą zresetowania hasła i kliknij zawarty w niej link:

18. Na stronie resetowania hasła wprowadź nowe hasło użytkownika i kliknij *Zresetuj hasło*. Zostaniesz automatycznie przekierowany na stronę logowania; zaloguj się tam nowym hasłem:



Implementacja autoryzacji

W dotychczasowej części rozdziału przyjrzałeś się, jak obsługiwać uwierzytelnianie użytkowników i jak postępować z ich logowaniem. W kolejnej części zobaczysz, jak zarządzać prawami dostępu w celu doprecyzowania, co komu wolno.

Najprostszą metodą autoryzacji jest użycie metaatrybutu `[Authorize]`, który zupełnie wyłącza dostęp anonimowy. W tym przypadku, aby uzyskać dostęp do zastrzeżonych zasobów, użytkownicy muszą być zalogowani.

Zobaczmy, jak zaimplementować ten sposób w aplikacji *Kółko i krzyżyk*:

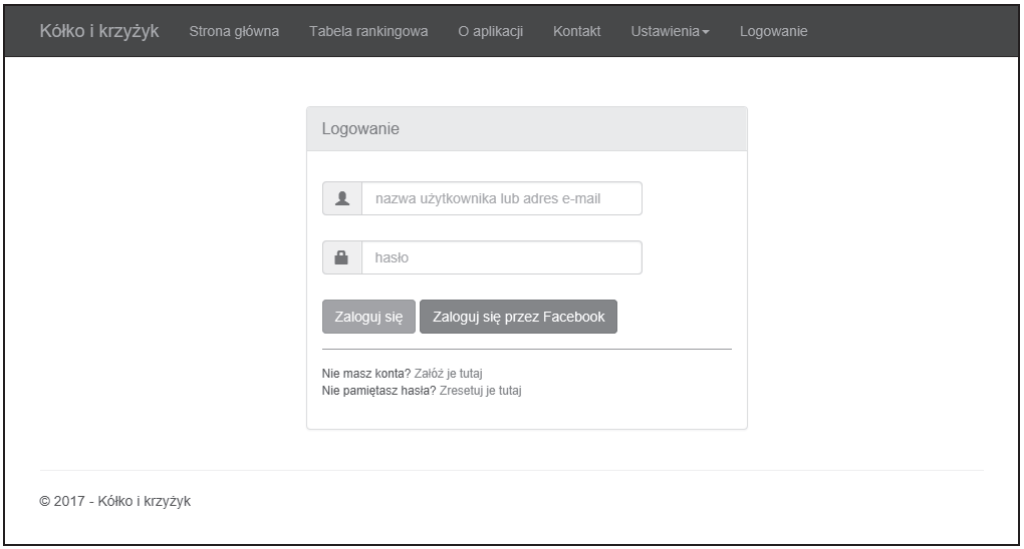
1. Do klasy `HomeController` dodaj metodę `SecuredPage` i zablokuj anonimowy dostęp do niej przez dodanie atrybutu `[Authorize]`:

```
[Authorize]
public async Task<IActionResult> SecuredPage()
{
  return await Task.Run(() =>
  {
    ViewBag.SecureWord = "Zabezpieczona strona";
    return View("SecuredPage");
  });
}
```

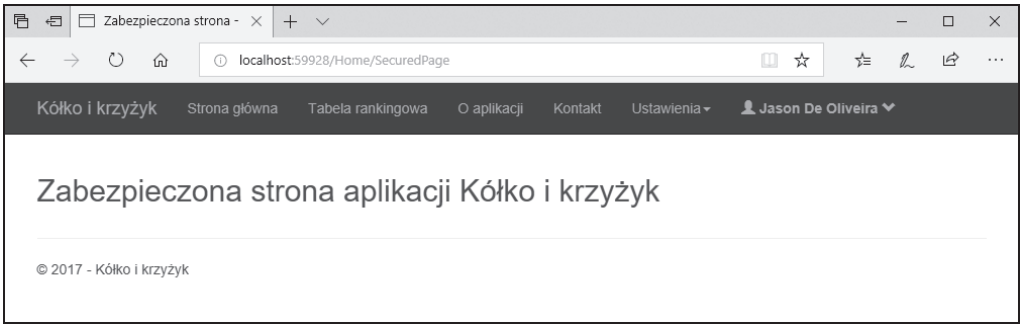
2. Do folderu *Views/Home* dodaj widok o nazwie *SecuredPage*:

```
@{
  ViewData["Title"] = "Zabezpieczona strona";
}
@section Desktop {<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<div class="row">
  <div class="col-lg-12">
    <h2>@ViewBag.SecureWord aplikacji Kółko i krzyżyk</h2>
  </div>
</div>
```

3. Spróbuj wejść bez zalogowania na zabezpieczoną stronę, wpisując ręcznie jej adres URL *http://<host>/Home/SecuredPage*; zostaniesz automatycznie przeniesiony na stronę logowania:



4. Wprowadź poprawne dane uwierzytelniające użytkownika i zaloguj się; zostaniesz automatycznie przeniesiony na zabezpieczoną stronę i będziesz mógł ją zobaczyć:



Kolejnym względnie popularnym podejściem jest użycie zabezpieczeń opartych na rolach (ang. *role-based security*), które dają bardziej zaawansowane możliwości. Jest to jedna z zalecanych metod zabezpieczania aplikacji internetowych ASP.NET Core 2.0.

Poniższy przykład wyjaśnia, jak to działa:

1. Do folderu *Models* dodaj klasę `UserRoleModel`, dziedziczącą po klasie generycznej `IdentityUserRole<long>`; będzie ona używana przez funkcje uwierzytelniania wbudowane we framework Identity platformy ASP.NET Core 2.0:

```
public class UserRoleModel : IdentityUserRole<Guid>
{
  [Key]
  public long Id { get; set; }
}
```

2. Zmień metodę `OnModelCreating` klasy `GameDbContext`:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  ...
  modelBuilder.Entity<IdentityUserRole<Guid>>()
   .ToTable("UserRoleModel")
   .HasKey(x => new { x.UserId, x.RoleId });
}
```

3. Otwórz konsolę menedżera pakietów NuGet i wykonaj polecenie `Add-Migration IdentityDb2`, a następnie `Update-Database`.
4. W klasie `UserService` zmodyfikuj konstruktor, aby tworzył dwie role o nazwach `Player` (gracz) i `Administrator`, jeśli takie jeszcze nie istnieją:

```
public UserService(RoleManager<RoleModel> roleManager,
 ApplicationUserManager userManager, ILogger<UserService>
 logger, SignInManager<UserModel> signInManager)
{
  ...
```

```
    if (!roleManager.RoleExistsAsync("Player").Result)
      roleManager.CreateAsync(new RoleModel {
      Name = "Player" }).Wait();

    if (!roleManager.RoleExistsAsync("Administrator").Result)
      roleManager.CreateAsync(new RoleModel {
      Name = "Administrator" }).Wait();
  }
```

1. Zmień metodę `RegisterUser` klasy `UserService` tak, by podczas rejestracji użytkownika nadawała mu rolę administratora albo gracza:

```
  ...
  try
  {
    userModel.UserName = userModel.Email;
    var result = await _userManager.CreateAsync(userModel,
     userModel.Password);
    if (result == IdentityResult.Success)
    {
      if(userModel.FirstName == "Jason")
        await _userManager.AddToRoleAsync(userModel,
         "Administrator");
      else
        await _userManager.AddToRoleAsync(userModel, "Player");
    }

    return result == IdentityResult.Success;
  }
  ...
```

> W tym przykładzie kod identyfikujący, czy użytkownik pełni rolę administratora, jest celowo bardzo prymitywny. W swoich aplikacjach powinieneś zaimplementować coś bardziej wyrafinowanego.

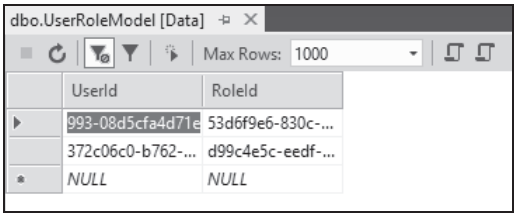
6. Uruchom aplikację i zarejestruj użytkownika, w okienku *SQL Server Object Explorer* otwórz tabelę *RoleModel* i przeanalizuj jej zawartość:

dbo.RoleModel [Data]

Max Rows: 1000

Id	ConcurrencyStamp	Name	NormalizedName
791e-08d5ce476ff9	29788856-107d-42f9-...	Player	PLAYER
7a231782-6fa4-...	45c267cd-f874-49f3-...	Administrator	ADMINISTRATOR
NULL	NULL	NULL	NULL

7. W okienku *SQL Server Object Explorer* otwórz tabelę *UserRoleModel* i przeanalizuj jej zawartość:



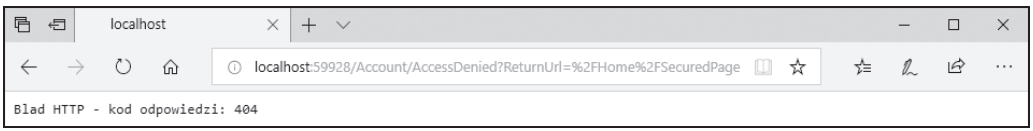
8. Zmień metodę `SignInUser` klasy `UserService` tak, by mapowała role do oświadczeń (ang. *claims*):

```
...
identity.AddClaim(new Claim("Score", user.Score.ToString()));
var roles = await _userManager.GetRolesAsync(user);
identity.AddClaims(roles?.Select(r => new
 Claim(ClaimTypes.Role, r)));

await httpContext.SignInAsync(
 CookieAuthenticationDefaults.AuthenticationScheme,
  new ClaimsPrincipal(identity),
  new AuthenticationProperties { IsPersistent = false });
...
```

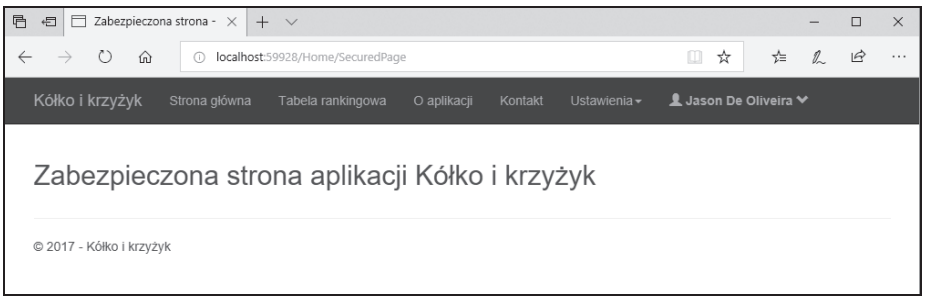
1. W metodzie `SecuredPage` klasy `HomeController` do zabezpieczenia dostępu użyj roli administratora, poprawiając atrybut `Authorize`:

```
[Authorize(Roles = "Administrator")]
```

10. Uruchom aplikację. Jeśli bez zalogowania wejdziesz na stronę *http://<host>/Home/SecuredPage*, zostaniesz przekierowany na stronę logowania. Zaloguj się jako użytkownik z rolą gracza, a zostaniesz przekierowany na stronę odmowy dostępu (która nie istnieje, stąd błąd 404), ponieważ użytkownik nie pełni roli administratora:



11. Wyloguj się, a następnie zaloguj jako użytkownik pełniący rolę administratora; zobaczysz teraz zabezpieczoną stronę, ponieważ masz odpowiednią rolę:



W kolejnym przykładzie przyjrzysz się, jak automatycznie zalogować zarejestrowanego użytkownika oraz jak uaktywnić uwierzytelnianie oparte na oświadczeniach (ang. *claims-based*) i na zasadach (ang. *policy-based*):

1. W klasie `UserService` zmień metodę `SignInUser` oraz dodaj nową o nazwie `SignIn`:

```
public async Task<SignInResult> SignInUser(LoginModel
 loginModel, HttpContext httpContext)
{
  var start = DateTime.Now;
  _logger.LogTrace($"Logowanie użytkownika {loginModel.UserName}");
  var stopwatch = new Stopwatch();
  stopwatch.Start();

  try
  {
    var user =
      await _userManager.FindByNameAsync(loginModel.UserName);
    var isValid =
      await _signInManager.CheckPasswordSignInAsync(user,
       loginModel.Password, true);

    if (!isValid.Succeeded)
    {
      return SignInResult.Failed;
    }

    if (!await _userManager.IsEmailConfirmedAsync(user))
    {
      return SignInResult.NotAllowed;
    }

    if (await _userManager.GetTwoFactorEnabledAsync(user))
    {
      return SignInResult.TwoFactorRequired;
    }

    await SignIn(httpContext, user);

    return isValid;
  }
  catch (Exception ex)
  {
    _logger.LogError($"Nie można zalogować użytkownika {loginModel.UserName}
    ↪- {ex}");
    throw ex;
  }
  finally
  {
    stopwatch.Stop();
    _logger.LogTrace($"Logowanie użytkownika {loginModel.UserName} ukończono
     ↪w czasie {stopwatch.Elapsed}");
  }
```

```
}

private async Task SignIn(HttpContext httpContext, UserModel user)
{
  var identity = new ClaimsIdentity(
    CookieAuthenticationDefaults.AuthenticationScheme);
  identity.AddClaim(new Claim(ClaimTypes.Name, user.UserName));
  identity.AddClaim(new Claim(ClaimTypes.GivenName,
    user.FirstName));
  identity.AddClaim(new Claim(ClaimTypes.Surname,
    user.LastName));
  identity.AddClaim(new Claim("displayName",
    $"{user.FirstName} {user.LastName}"));
  if (!string.IsNullOrEmpty(user.PhoneNumber))
  {
    identity.AddClaim(new Claim(ClaimTypes.HomePhone,
     user.PhoneNumber));
  }
  identity.AddClaim(new Claim("Score", user.Score.ToString()));

  var roles = await _userManager.GetRolesAsync(user);
  identity.AddClaims(roles?.Select(r =>
   new Claim(ClaimTypes.Role, r)));

  if (user.FirstName == "Jason")
  identity.AddClaim(new Claim("AccessLevel", "Administrator"));

  await httpContext.SignInAsync(
   CookieAuthenticationDefaults.AuthenticationScheme,
   new ClaimsPrincipal(identity),
   new AuthenticationProperties { IsPersistent = false });
}
```

W tym przykładzie kod identyfikujący, czy użytkownik ma uprawnienia administratora, jest celowo bardzo prymitywny. W swoich aplikacjach powinieneś zaimplementować coś bardziej wyrafinowanego.

2. Zmień metodę `RegisterUser` klasy `UserService`, dodając nowy parametr do automatycznego logowania użytkownika po rejestracji, i ponownie wyodrębnij interfejs `IUserService`:

```
public async Task<bool> RegisterUser(UserModel userModel,
 bool isOnline = false)
{
  ...
  if (result == IdentityResult.Success)
  {
    ...
    if (isOnline)
    {
      HttpContext httpContext =
        new HttpContextAccessor().HttpContext;
      await SignIn(httpContext, userModel);
```

```
      }
    }
    ...
  }
```

3. Popraw metodę `Index` klasy `UserRegistrationController`, aby automatycznie logowała nowo zarejestrowanego użytkownika:

```
...
await _userService.RegisterUser(userModel, true);
...
```

4. Popraw metodę `ConfirmGameInvitation` klasy `GameInvitationController`, aby automatycznie logowała nowo zarejestrowanego użytkownika:

```
...
await _userService.RegisterUser(new UserModel
{
  Email = gameInvitation.EmailTo,
  EmailConfirmationDate = DateTime.Now,
  EmailConfirmed = true,
  FirstName = "",
  LastName = "",
  Password = "Azerty123!",
  UserName = gameInvitation.EmailTo
}, true);
...
```

5. Do klasy `Startup`, tuż po konfiguracji oprogramowania pośredniczącego MVC, dodaj nową zasadę `AdministratorAccessLevelPolicy`:

```
services.AddAuthorization(options =>
{
  options.AddPolicy("AdministratorAccessLevelPolicy",
   policy => policy.RequireClaim("AccessLevel",
   "Administrator"));
});
```

6. Zmień metodę `SecuredPage` klasy `HomeController`, aby do zabezpieczenia dostępu używała zasad, a nie ról, poprawiając atrybut `Authorize`:

```
[Authorize(Policy = "AdministratorAccessLevelPolicy")]
```

Ponieważ w obrębie platformy ASP.NET Core 2.0 da się jednocześnie używać kilku rodzajów oprogramowania pośredniczącego uwierzytelniania (korzystających z plików cookie, elementu nośnego JWT itd.), może się pojawić konieczność ograniczenia dostępu tylko do jednego konkretnego oprogramowania pośredniczącego.

W takim przypadku przedstawiony wcześniej atrybut `Authorize` pozwala zdefiniować, które oprogramowanie pośredniczące może uwierzytelniać użytkowników.

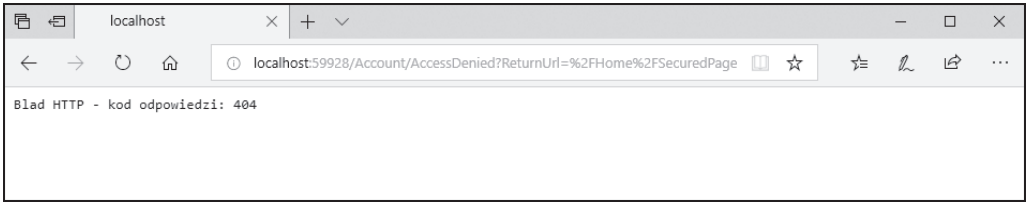
Oto przykład dopuszczający oprogramowanie pośredniczące uwierzytelniania korzystające z plików cookie i elementu nośnego JWT:

```
[Authorize(AuthenticationSchemes = "Cookie,Bearer",
  Policy = "AdministratorAccessLevelPolicy")]
```

7. Uruchom aplikację, zarejestruj użytkownika z poziomem dostępu `Administrator`, zaloguj się i wejdź na stronę *http://<host>/Home/SecuredPage*. Wszystko powinno działać tak jak wcześniej.

> Być może będziesz musiał wyczyścić pliki cookie i zalogować się ponownie, aby utworzyć nowy token uwierzytelniania z wymaganymi oświadczeniami.

8. Spróbuj wejść na zabezpieczoną stronę jako użytkownik bez wymaganego poziomu dostępu; tak jak wcześniej, zostaniesz przekierowany na adres *http://<host>/Account/AccessDenied?ReturnUrl=%2FHome%2FSecuredPage*:



9. Wyloguj się, a następnie zaloguj jako użytkownik z poziomem dostępu `Administrator`; teraz zobaczysz zabezpieczoną stronę, bo użytkownik ma wymagany poziom dostępu.

Podsumowanie

W tym rozdziale nauczyłeś się zabezpieczać aplikacje ASP.NET Core 2.0, w tym zarządzać uwierzytelnianiem i autoryzacją użytkowników.

W próbnej aplikacji wprowadziłeś podstawowe zabezpieczenie za pomocą formularzy oraz bardziej zaawansowane uwierzytelnienie za pośrednictwem zewnętrznego dostawcy — serwisu Facebook. To powinno dać Ci orientację, jak podejść do tych ważnych zagadnień we własnych aplikacjach.

Nauczyłeś się ponadto, jak dodawać standardowy mechanizm resetowania haseł, ponieważ użytkownicy wciąż je zapominają, a na tego typu żądania trzeba odpowiadać tak bezpiecznie, jak się tylko da.

Omówiliśmy nawet uwierzytelnianie dwuskładnikowe, które zapewnia jeszcze wyższy poziom bezpieczeństwa w kluczowych aplikacjach.

Na koniec przyjrzałeś się, jak na różne sposoby radzić sobie z autoryzacją (podstawową, opartą na rolach oraz na zasadach), więc będziesz mógł zdecydować, które podejście najbardziej pasuje do Twoich konkretnych zastosowań.

W następnym rozdziale omówimy różne opcje hostingu i wdrażania aplikacji internetowych ASP.NET Core 2.0.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

ASP.NET Core 2.0. Podejmij wyzwanie!

Framework ASP.NET Core 2.0 stanowi odpowiedź Microsoftu na potrzeby programistów, które z kolei zmieniają się wraz z rynkiem IT. Klienci wymagają od aplikacji zgodności z różnymi standardami, wysokiej efektywności i skalowalności, a czas wyprodukowania i wdrożenia nowego rozwiązania ma być jak najkrótszy. Do takiej pracy potrzeba narzędzi o odpowiedniej produktywności, rozszerzalności i elastyczności. Dzięki uwzględnieniu tych wyśrubowanych kryteriów Microsoft stworzył platformę ASP.NET Core pozwalającą na tworzenie, kompilację i uruchamianie aplikacji w dowolnym środowisku. Można też korzystać z zewnętrznych bibliotek i z najbardziej aktualnych wzorców projektowych. Opanowanie tego złożonego narzędzia pozwoli zająć programiście znakomitą pozycję wyjściową do tworzenia wydajnych i nowoczesnych aplikacji internetowych.

Ta książka jest przeznaczona dla programistów chcących budować nowoczesne aplikacje internetowe na platformie ASP.NET Core 2.0. W przystępny i zrozumiały sposób, na praktycznych przykładach wyjaśniono tu możliwości ASP.NET Core 2.0. Większość kluczowych funkcji została opisana z wykorzystaniem zwięzłych przykładów. Dzięki jasnym instrukcjom krok po kroku możliwe jest niemal natychmiastowe rozpoczęcie programowania. W książce omówiono tworzenie responsywnych aplikacji internetowych, stosowanie w praktyce modelu MVC, wdrażanie aplikacji z wykorzystaniem technologii chmury, a także monitorowanie pracy oprogramowania w środowisku produkcyjnym i reagowanie na pojawiające się problemy.

W tej książce między innymi:

- funkcjonalność i ograniczenia ASP.NET Core 2.0 oraz struktura i koncepcja aplikacji
- przygotowanie i konfiguracja środowiska pracy
- tworzenie aplikacji MVC i aplikacji Web API
- praca z bazą danych z użyciem zaawansowanych funkcji programu Entity Framework Core 2
- zabezpieczanie aplikacji i jej testowanie

Jason De Oliveira jest dyrektorem ds. technicznych (CTO) w firmie programistycznej MEGA International w Paryżu. Ma ogromną wiedzę i doświadczenie w dziedzinie architektury oprogramowania i architektury korporacyjnej. Chętnie zabiera głos na konferencjach, pisze fachowe książki i artykuły. Od ponad sześciu lat otrzymuje nagrodę MVP C#/.NET Microsoftu. **Michel Bruchet** jest architektem aplikacji w MEGA International. Od ponad dwudziestu lat kieruje złożonymi projektami IT. Jest również uważany za mózg firmy technologicznej Ingenius Solution.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
INFORMATYKA W NAJLEPSZYM WYDANIU

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-4499-0
9 788328 344990
Cena: 69,00 zł
Packt