

Create an ASP.NET Core app with user data protected by authorization

12/18/2018 • 39 minutes to read •  +5

In this article

[Prerequisites](#)

[The starter and completed app](#)

[Secure user data](#)

[Create owner, manager, and administrator authorization handlers](#)

[Register the authorization handlers](#)

[Support authorization](#)

[Inject the authorization service into the views](#)

[Add or remove a user to a role](#)

[Differences between Challenge and Forbid](#)

[Test the completed app](#)

[Create the starter app](#)

By [Rick Anderson](#) and [Joe Audette](#)

This tutorial shows how to create an ASP.NET Core web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

- **Registered users** can view all the approved data and can edit/delete their own data.
- **Managers** can approve or reject contact data. Only approved contacts are visible to users.
- **Administrators** can approve/reject and edit/delete any data.

The images in this document don't exactly match the latest templates.

In the following image, user Rick (rick@example.com) is signed in. Rick can only view approved contacts and **Edit/Delete/Create New** links for his contacts. Only the last record, created by Rick, displays **Edit** and **Delete** links. Other users won't see the last record until a manager or administrator changes the status to "Approved".

Index - ContactManager

ContactManager Home About Contact Hello rick@example.com! Log off

Create New

| Address | City | Email | Name | State | Zip | Status | |
|----------------|---------|----------------------|-------------------|-------|-------|-----------|---|
| 5678 1st Ave W | Redmond | thorsten@example.com | Thorsten Weinrich | WA | 10999 | Approved | Details |
| 9012 State st | Redmond | yuhong@example.com | Yuhong Li | WA | 10999 | Approved | Details |
| 3456 Maple St | Redmond | jon@example.com | Jon Orton | WA | 10999 | Approved | Details |
| 123 N 456 E | GF | rick@example.com | Rick Anderson | MT | 59405 | Submitted | Edit Details Delete |

© 2017 - ContactManager

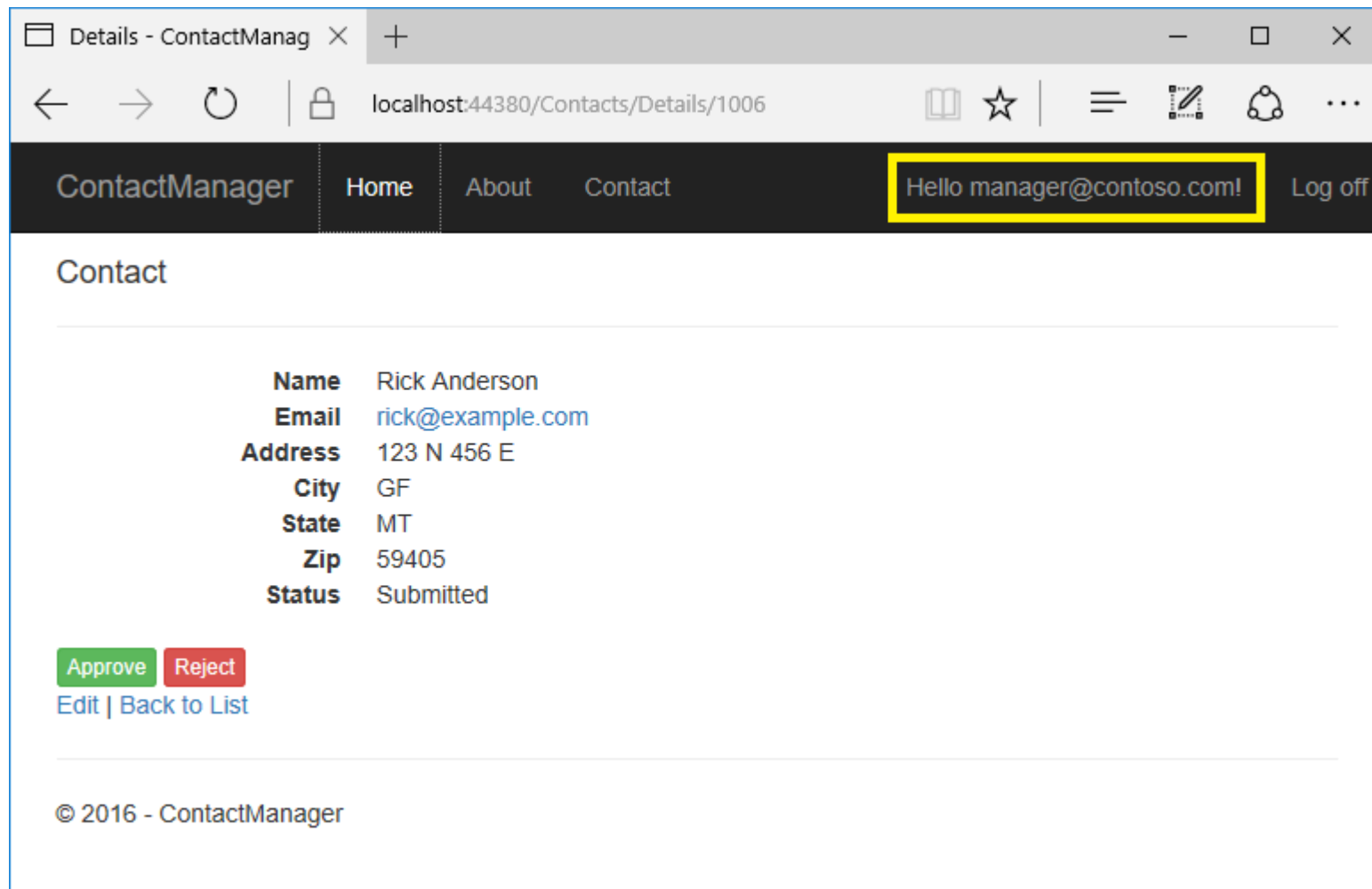
In the following image, `manager@contoso.com` is signed in and in the manager's role:

The screenshot shows a web browser window with the address `localhost:44380/Contacts`. The application header includes the title "ContactManager" and navigation links for "Home", "About", and "Contact". The user is logged in as "Hello manager@contoso.com!" with a "Log off" link. Below the header, there is a "Create New" link and a table of contacts. The table has columns for Address, City, Email, Name, State, Zip, Status, and a "Details" link. The "Rejected" status is highlighted with a red box, and the "Submitted" status is also highlighted with a red box.

| Address | City | Email | Name | State | Zip | Status | |
|----------------|---------|--------------------------------------|--------------------------|-------|-------|-----------|-------------------------|
| 1234 Main St | Redmond | debra@example.com | Debra Garcia | WA | 10999 | Rejected | Details |
| 5678 1st Ave W | Redmond | thorsten@example.com | Thorsten Weinrich | WA | 10999 | Approved | Details |
| 9012 State st | Redmond | yuhong@example.com | Yuhong Li | WA | 10999 | Approved | Details |
| 3456 Maple St | Redmond | jon@example.com | Jon Orton | WA | 10999 | Approved | Details |
| 7890 2nd Ave E | Redmond | diliana@example.com | Diliana Alexieva-Bosseva | WA | 10999 | Submitted | Details |
| 123 N 456 E | GF | rick@example.com | Rick Anderson | MT | 59405 | Approved | Details |

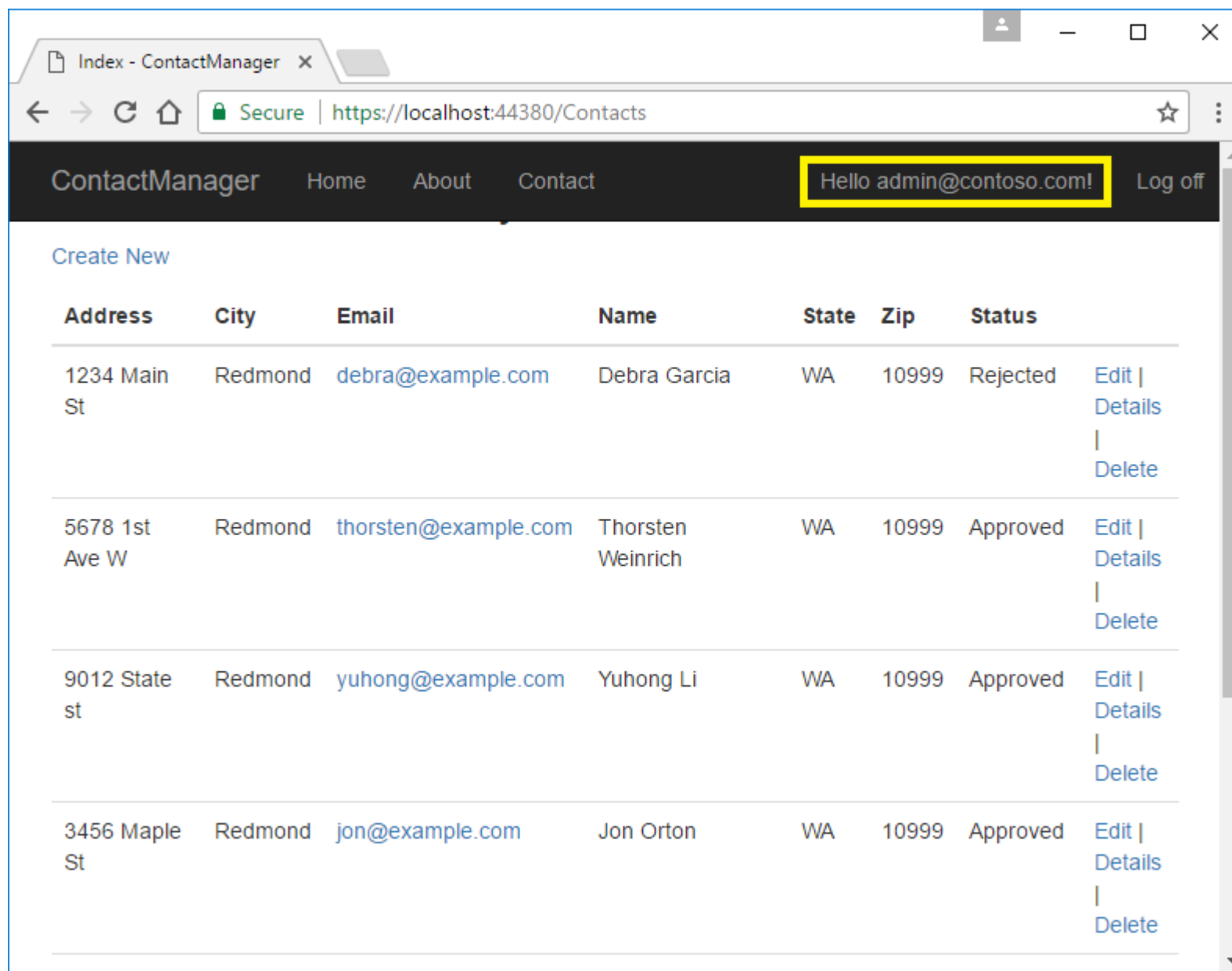
© 2016 - ContactManager

The following image shows the managers details view of a contact:



The **Approve** and **Reject** buttons are only displayed for managers and administrators.

In the following image, `admin@contoso.com` is signed in and in the administrator's role:



Index - ContactManager x

Secure | <https://localhost:44380/Contacts>

ContactManager Home About Contact Hello admin@contoso.com! Log off

Create New

| Address | City | Email | Name | State | Zip | Status | |
|----------------|---------|--|-------------------|-------|-------|----------|---|
| 1234 Main St | Redmond | debra@example.com | Debra Garcia | WA | 10999 | Rejected | Edit Details Delete |
| 5678 1st Ave W | Redmond | thorsten@example.com | Thorsten Weinrich | WA | 10999 | Approved | Edit Details Delete |
| 9012 State st | Redmond | yuhong@example.com | Yuhong Li | WA | 10999 | Approved | Edit Details Delete |
| 3456 Maple St | Redmond | jon@example.com | Jon Orton | WA | 10999 | Approved | Edit Details Delete |

The administrator has all privileges. She can read/edit/delete any contact and change the status of contacts.

The app was created by [scaffolding](#) the following contact model:

C#

 Copy

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

The sample contains the following authorization handlers:

- `ContactIsOwnerAuthorizationHandler`: Ensures that a user can only edit their data.
- `ContactManagerAuthorizationHandler`: Allows managers to approve or reject contacts.
- `ContactAdministratorsAuthorizationHandler`: Allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This tutorial is advanced. You should be familiar with:

- [ASP.NET Core](#)
- [Authentication](#)
- [Account Confirmation and Password Recovery](#)
- [Authorization](#)
- [Entity Framework Core](#)

The starter and completed app

[Download](#) the [completed](#) app. [Test](#) the completed app so you become familiar with its security features.

The starter app

[Download](#) the [starter](#) app.

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact.

Secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET [Identity](#) user ID to ensure users can edit their data, but not other users data. Add `ownerID` and `ContactStatus` to the `Contact` model:

| C# |  Copy |
|---|--|
| <pre>public class Contact { public int ContactId { get; set; } // user ID from AspNetUser table. public string OwnerID { get; set; } public string Name { get; set; } public string Address { get; set; } }</pre> | |

```
public string City { get; set; }

public string State { get; set; }
public string Zip { get; set; }
[DataType(DataType.EmailAddress)]
public string Email { get; set; }

public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

ownerID is the user's ID from the `AspNetUser` table in the [Identity](#) database. The `status` field determines if a contact is viewable by general users.

Create a new migration and update the database:

.NET Core CLI

 Copy

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Add Role services to Identity

Append [AddRoles](#) to add Role services:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{

```

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        Configuration.GetConnectionString("DefaultConnection")));
services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Require authenticated users

Set the default authentication policy to require users to be authenticated:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddRazorPages();

    services.AddControllers(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    });
}
```

```
});
```

You can opt out of authentication at the Razor Page, controller, or action method level with the `[AllowAnonymous]` attribute. Setting the default authentication policy to require users to be authenticated protects newly added Razor Pages and controllers. Having authentication required by default is more secure than relying on new controllers and Razor Pages to include the `[Authorize]` attribute.

Add [AllowAnonymous](#) to the Index and Privacy pages so anonymous users can get information about the site before they register.

C#

 Copy

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;


namespace ContactManager.Pages
{
    [AllowAnonymous]
    public class IndexModel : PageModel
    {
        private readonly ILogger<IndexModel> _logger;

        public IndexModel(ILogger<IndexModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

Configure the test account

The `SeedData` class creates two accounts: administrator and manager. Use the [Secret Manager tool](#) to set a password for these accounts. Set the password from the project directory (the directory containing *Program.cs*):

| | |
|--|--|
| .NET Core CLI |  Copy |
| <pre>dotnet user-secrets set SeedUserPW <PW></pre> | |

If a strong password is not specified, an exception is thrown when `SeedData.Initialize` is called.

Update `Main` to use the test password:

| | |
|--|--|
| C# |  Copy |
| <pre>public class Program { public static void Main(string[] args) { var host = CreateHostBuilder(args).Build(); using (var scope = host.Services.CreateScope()) { var services = scope.ServiceProvider; try { var context = services.GetRequiredService<ApplicationDbContext>(); context.Database.Migrate(); // requires using Microsoft.Extensions.Configuration; var config = host.Services.GetRequiredService<IConfiguration>(); // Set password with the Secret Manager tool. // dotnet user-secrets set SeedUserPW <pw></pre> | |

```
        var testUserPw = config["SeedUserPW"];

        SeedData.Initialize(services, testUserPw).Wait();
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
```

Create the test accounts and update the contacts

Update the `Initialize` method in the `SeedData` class to create the test accounts:

C#

 Copy

```
public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
{
    using (var context = new ApplicationDbContext(
        serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
    }
```



```
// dotnet user-secrets set SeedUserPW <pw>

// The admin user can do anything

var adminID = await EnsureUser(serviceProvider, testUserPw, "admin@contoso.com");
await EnsureRole(serviceProvider, adminID, Constants.ContactAdministratorsRole);

// allowed user can create and edit contacts that they create
var managerID = await EnsureUser(serviceProvider, testUserPw, "manager@contoso.com");
await EnsureRole(serviceProvider, managerID, Constants.ContactManagersRole);

SeedDB(context, adminID);
}
}

private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
                                             string testUserPw, string UserName)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser {
            UserName = UserName,
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, testUserPw);
    }

    if (user == null)
    {
        throw new Exception("The password is probably not strong enough!");
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider,
```

```
string uid, string role)
```

```
{
    IdentityResult IR = null;
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByIdAsync(uid);

    if (user == null)
    {
        throw new Exception("The testUserPw password was probably not strong enough!");
    }

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

Add the administrator user ID and `contactStatus` to the contacts. Make one of the contacts "Submitted" and one "Rejected". Add the user ID and status to all the contacts. Only one contact is shown:

C#

 Copy

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
```

```
11 if (context.Contact.Any())
{
    return; // DB has been seeded
}

context.Contact.AddRange(
    new Contact
    {
        Name = "Debra Garcia",
        Address = "1234 Main St",
        City = "Redmond",
        State = "WA",
        Zip = "10999",
        Email = "debra@example.com",
        Status = ContactStatus.Approved,
        OwnerID = adminID
    },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The `ContactIsOwnerAuthorizationHandler` verifies that the user acting on a resource owns the resource.

C#

 Copy

```
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
```

```
UserManager<IdentityUser> _userManager;

public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
    userManager)
{
    _userManager = userManager;
}

protected override Task
    HandleRequirementAsync(AuthorizationHandlerContext context,
        OperationAuthorizationRequirement requirement,
        Contact resource)
{
    if (context.User == null || resource == null)
    {
        return Task.CompletedTask;
    }

    // If not asking for CRUD permission, return.

    if (requirement.Name != Constants.CreateOperationName &&
        requirement.Name != Constants.ReadOperationName &&
        requirement.Name != Constants.UpdateOperationName &&
        requirement.Name != Constants.DeleteOperationName )
    {
        return Task.CompletedTask;
    }

    if (resource.OwnerID == _userManager.GetUserId(context.User))
    {
        context.Succeed(requirement);
    }

    return Task.CompletedTask;
}
}
```

The `ContactIsOwnerAuthorizationHandler` calls [context.Succeed](#) if the current authenticated user is the contact owner. Authorization handlers generally:

- Return `context.Succeed` when the requirements are met.
- Return `Task.CompletedTask` when requirements aren't met. `Task.CompletedTask` is not success or failure—it allows other authorization handlers to run.

If you need to explicitly fail, return [context.Fail](#).

The app allows contact owners to edit/delete/create their own data. `ContactIsOwnerAuthorizationHandler` doesn't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` verifies the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

C#

 Copy

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
```

```
        OperationAuthorizationRequirement requirement,  
        Contact resource)  
{  
    if (context.User == null || resource == null)  
    {  
        return Task.CompletedTask;  
    }  
  
    // If not asking for approval/reject, return.  
    if (requirement.Name != Constants.ApproveOperationName &&  
        requirement.Name != Constants.RejectOperationName)  
    {  
        return Task.CompletedTask;  
    }  
  
    // Managers can approve or reject.  
    if (context.User.IsInRole(Constants.ContactManagersRole))  
    {  
        context.Succeed(requirement);  
    }  
  
    return Task.CompletedTask;  
}  
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` verifies the user acting on the resource is an administrator. Administrator can do all operations.

C#

 Copy

```
using System.Threading.Tasks;
```

```
using ContactManager.Models;

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            OperationAuthorizationRequirement requirement,
            Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for [dependency injection](#) using [AddScoped](#). The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core [Identity](#), which is built on Entity Framework Core. Register the handlers with the service collection so they're available to the `ContactsController` through [dependency injection](#). Add the

following code to the end of `ConfigureServices`:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddRazorPages();

    services.AddControllers(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    });
    // Authorization handlers.
    services.AddScoped<IAuthorizationHandler,
        ContactIsOwnerAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
        ContactAdministratorsAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
        ContactManagerAuthorizationHandler>();
}
```


`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They're singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

Support authorization

In this section, you update the Razor Pages and add an operations requirements class.

Review the contact operations requirements class

Review the `ContactOperations` class. This class contains the requirements the app supports:

C#

 Copy

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
            new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
            new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
            new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
            new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
            new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
            new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }
}
```

```
public class Constants
{
    public static readonly string CreateOperationName = "Create";
    public static readonly string ReadOperationName = "Read";
    public static readonly string UpdateOperationName = "Update";
    public static readonly string DeleteOperationName = "Delete";
    public static readonly string ApproveOperationName = "Approve";
    public static readonly string RejectOperationName = "Reject";

    public static readonly string ContactAdministratorsRole =
        "ContactAdministrators";
    public static readonly string ContactManagersRole = "ContactManagers";
}
```

Create a base class for the Contacts Razor Pages

Create a base class that contains the services used in the contacts Razor Pages. The base class puts the initialization code in one location:

C#

 Copy

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }
    }
}
```


```
public DI_BasePageModel(  
    ApplicationDbContext context,  
    IAuthorizationService authorizationService,  
    UserManager<IdentityUser> userManager) : base()  
{  
    Context = context;  
    UserManager = userManager;  
    AuthorizationService = authorizationService;  
}  
}
```

The preceding code:

- Adds the `IAuthorizationService` service to access to the authorization handlers.
- Adds the Identity `UserManager` service.
- Add the `ApplicationDbContext`.

Update the CreateModel

Update the create page model constructor to use the `DI_BasePageModel` base class:

| C# |  Copy |
|--|--|
| <pre>public class CreateModel : DI_BasePageModel { public CreateModel(ApplicationDbContext context, IAuthorizationService authorizationService, UserManager<IdentityUser> userManager) : base(context, authorizationService, userManager) { } }</pre> | |

Update the `CreateModel.OnPostAsync` method to:

- Add the user ID to the `Contact` model.
- Call the authorization handler to verify the user has permission to create contacts.

C#

 Copy

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    Contact.OwnerID = UserManager.GetUserId(User);

    // requires using ContactManager.Authorization;
    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, Contact,
        ContactOperations.Create);

    if (!isAuthorized.Succeeded)
    {
        return Forbid();
    }

    Context.Contact.Add(Contact);
    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Update the IndexModel

Update the `onGetAsync` method so only approved contacts are shown to general users:

C#

 Copy

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                        select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status == ContactStatus.Approved
                                           || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

Update the EditModel

Add an authorization handler to verify the user owns the contact. Because resource authorization is being validated, the `[Authorize]` attribute is not enough. The app doesn't have access to the resource when attributes are evaluated. Resource-based authorization must be imperative. Checks must be performed once the app has access to the resource, either by loading it in the page model or by loading it within the handler itself. You frequently access the resource by passing in the resource key.

C#

 Copy

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
            User, Contact,
            ContactOperations.Update);

        if (!isAuthorized.Succeeded)
```

```

    {
        return Forbid();
    }

    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Fetch Contact from DB to get OwnerID.
    var contact = await Context
        .Contact.AsNoTracking()
        .FirstOrDefaultAsync(m => m.ContactId == id);

    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, contact,
        ContactOperations.Update);

    if (!isAuthorized.Succeeded)
    {
        return Forbid();
    }

    Contact.OwnerID = contact.OwnerID;

    Context.Attach(Contact).State = EntityState.Modified;

    if (Contact.Status == ContactStatus.Approved)
    {

```

```
// If the contact is updated after approval,  
  
// and the user cannot approve,  
// set the status back to submitted so the update can be  
// checked and approved.  
var canApprove = await AuthorizationService.AuthorizeAsync(User,  
    Contact,  
    ContactOperations.Approve);  
  
if (!canApprove.Succeeded)  
{  
    Contact.Status = ContactStatus.Submitted;  
}  
}  
  
await Context.SaveChangesAsync();  
  
return RedirectToPage("./Index");  
}  
}
```

Update the DeleteModel

Update the delete page model to use the authorization handler to verify the user has delete permission on the contact.

C#

 Copy

```
public class DeleteModel : DI_BasePageModel  
{  
    public DeleteModel(  
        ApplicationDbContext context,  
        IAuthorizationService authorizationService,  
        UserManager<IdentityUser> userManager)  
        : base(context, authorizationService, userManager)  
    {  
    }  
}
```


[BindProperty]

```
public Contact Contact { get; set; }
```

```
public async Task<IActionResult> OnGetAsync(int id)
{
    Contact = await Context.Contact.FirstOrDefaultAsync(
        m => m.ContactId == id);

    if (Contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, Contact,
        ContactOperations.Delete);

    if (!isAuthorized.Succeeded)
    {
        return Forbid();
    }

    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var contact = await Context
        .Contact.AsNoTracking()
        .FirstOrDefaultAsync(m => m.ContactId == id);

    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, contact,
        ContactOperations.Delete);
}
```

```
1+ (!IsAuthorized.Succeeded)

{
    return Forbid();
}

Context.Contact.Remove(contact);
await Context.SaveChangesAsync();

return RedirectToPage("./Index");
}
```

Inject the authorization service into the views

Currently, the UI shows edit and delete links for contacts the user can't modify.

Inject the authorization service in the *Pages/_ViewImports.cshtml* file so it's available to all views:

CSHTML

 Copy

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

The preceding markup adds several `using` statements.

Update the **Edit** and **Delete** links in *Pages/Contacts/Index.cshtml* so they're only rendered for users with the appropriate permissions:

permissions.

CSHTML

 Copy

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var contact in Model.Contacts)
        {
            <tr>
                <td>@contact.Name</td>
                <td>@contact.Address</td>
                <td>@contact.City</td>
                <td>@contact.State</td>
                <td>@contact.Zip</td>
                <td>@contact.Email</td>
                <td>@contact.Status</td>
            </tr>
        }
    </tbody>
</table>
```

```
        </th>

        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model.Contact)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Address)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.City)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.State)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
        </tr>
    }
</tbody>
</table>
```

```
</td>
<td>
    @Html.DisplayFor(modelItem => item.Status)
</td>
<td>
    @if ((await AuthorizationService.AuthorizeAsync(
        User, item,
        ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@item.ContactId">Edit</a>
        <text> | </text>
    }

    <a asp-page="./Details" asp-route-id="@item.ContactId">Details</a>


    @if ((await AuthorizationService.AuthorizeAsync(
        User, item,
        ContactOperations.Delete)).Succeeded)
    {
        <text> | </text>
        <a asp-page="./Delete" asp-route-id="@item.ContactId">Delete</a>
    }
</td>
</tr>
}
</tbody>
</table>
```

Warning

Hiding links from users that don't have permission to change data doesn't secure the app. Hiding links makes the app more user-friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The Razor Page or controller must enforce access checks to secure the data.

Update Details

Update the details view so managers can approve or reject contacts:

| CSHTML |  Copy |
|---|--|
| <pre>@*Precedng markup omitted for brevity.*@ <dt> @Html.DisplayNameFor(model => model.Contact.Email) </dt> <dd> @Html.DisplayFor(model => model.Contact.Email) </dd> <dt> @Html.DisplayNameFor(model => model.Contact.Status) </dt> <dd> @Html.DisplayFor(model => model.Contact.Status) </dd> </dl> </div> @if (Model.Contact.Status != ContactStatus.Approved) { @if ((await AuthorizationService.AuthorizeAsync(User, Model.Contact, ContactOperations.Approve)).Succeeded) { <form style="display:inline;" method="post"> <input type="hidden" name="id" value="@Model.Contact.ContactId" /> <input type="hidden" name="status" value="@ContactStatus.Approved" /> <button type="submit" class="btn btn-xs btn-success">Approve</button> </form> } } @if (Model.Contact.Status != ContactStatus.Rejected)</pre> | |

```
1
@if ((await AuthorizationService.AuthorizeAsync(
    User, Model.Contact, ContactOperations.Reject)).Succeeded)
{
    <form style="display:inline;" method="post">
        <input type="hidden" name="id" value="@Model.Contact.ContactId" />
        <input type="hidden" name="status" value="@ContactStatus.Rejected" />
        <button type="submit" class="btn btn-xs btn-danger">Reject</button>
    </form>
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
        User, Model.Contact,
        ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>
```

Update the details page model:

C#

 Copy

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }
```

```
public async Task<IActionResult> OnGetAsync(int id)
{
    Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

    if (Contact == null)
    {
        return NotFound();
    }

    var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
        User.IsInRole(Constants.ContactAdministratorsRole);

    var currentUserId = UserManager.GetUserId(User);

    if (!isAuthorized
        && currentUserId != Contact.OwnerID
        && Contact.Status != ContactStatus.Approved)
    {
        return Forbid();
    }

    return Page();
}

public async Task<IActionResult> OnPostAsync(int id, ContactStatus status)
{
    var contact = await Context.Contact.FirstOrDefaultAsync(
        m => m.ContactId == id);

    if (contact == null)
    {
        return NotFound();
    }

    var contactOperation = (status == ContactStatus.Approved)
        ? ContactOperations.Approve
        : ContactOperations.Reject;
```



```
var isAuthorized = await AuthorizationService.AuthorizeAsync(User, contact,
                                                    contactOperation);
if (!isAuthorized.Succeeded)
{
    return Forbid();
}
contact.Status = status;
Context.Contact.Update(contact);
await Context.SaveChangesAsync();

return RedirectToPage("./Index");
}
```

Add or remove a user to a role

See [this issue](#) for information on:

- Removing privileges from a user. For example, muting a user in a chat app.
- Adding privileges to a user.

Differences between Challenge and Forbid

This app sets the default policy to [require authenticated users](#). The following code allows anonymous users. Anonymous users are allowed to show the differences between Challenge vs Forbid.

| C# |  Copy |
|---|--|
| <pre>[AllowAnonymous] public class Details2Model : DI_BasePageModel { public Details2Model(</pre> | |

```
ApplicationDbContext context,
IAuthorizationService authorizationService,
userManager<IdentityUser> userManager)
: base(context, authorizationService, userManager)
{
}

public Contact Contact { get; set; }

public async Task<IActionResult> OnGetAsync(int id)
{
    Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

    if (Contact == null)
    {
        return NotFound();
    }

    if (!User.Identity.IsAuthenticated)
    {
        return Challenge();
    }

    var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
        User.IsInRole(Constants.ContactAdministratorsRole);

    var currentUserId = UserManager.GetUserId(User);

    if (!isAuthorized
        && currentUserId != Contact.OwnerID
        && Contact.Status != ContactStatus.Approved)
    {
        return Forbid();
    }

    return Page();
}
}
```


In the preceding code:

- When the user is **not** authenticated, a `ChallengeResult` is returned. When a `ChallengeResult` is returned, the user is redirected to the sign-in page.
- When the user is authenticated, but not authorized, a `ForbidResult` is returned. When a `ForbidResult` is returned, the user is redirected to the access denied page.

Test the completed app

If you haven't already set a password for seeded user accounts, use the [Secret Manager tool](#) to set a password:

- Choose a strong password: Use eight or more characters and at least one upper-case character, number, and symbol. For example, `Passw0rd!` meets the strong password requirements.
- Execute the following command from the project's folder, where `<PW>` is the password:

| | |
|--|--|
| .NET Core CLI |  Copy |
| <pre>dotnet user-secrets set SeedUserPW <PW></pre> | |

If the app has contacts:

- Delete all of the records in the `Contact` table.
- Restart the app to seed the database.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate sessions). In one browser, register a new user (for example, `test@example.com`). Sign in to each browser with a different user. Verify the following operations:

- Registered users can view all of the approved contact data.


- Registered users can edit/delete their own data.
- Managers can approve/reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete all data.

| User | Seeded by the app | Options |
|---------------------|-------------------|--|
| test@example.com | No | Edit/delete the own data. |
| manager@contoso.com | Yes | Approve/reject and edit/delete own data. |
| admin@contoso.com | Yes | Approve/reject and edit/delete all data. |

Create a contact in the administrator's browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user can't perform these operations.

Create the starter app

- Create a Razor Pages app named "ContactManager"
 - Create the app with **Individual User Accounts**.
 - Name it "ContactManager" so the namespace matches the namespace used in the sample.
 - `-uld` specifies LocalDB instead of SQLite

| | |
|--|--|
| .NET Core CLI |  Copy |
| <pre>dotnet new webapp -o ContactManager -au Individual -uld</pre> | |

- Add *Models/Contact.cs*:

| | |
|----|--|
| C# |  Copy |
|----|--|

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

- Scaffold the Contact model.
- Create initial migration and update the database:

.NET Core CLI

 Copy

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet aspnet-codegenerator razorpage -m Contact -udl -dc ApplicationDbContext -outDir Pages\Contacts --
referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

If you experience a bug with the `dotnet aspnet-codegenerator razorpage` command, see [this GitHub issue](#).

- Update the **ContactManager** anchor in the *Pages/Shared/_Layout.cshtml* file:

CSHTML

 Copy

```
<a class="navbar-brand" asp-area="" asp-page="/Contacts/Index">ContactManager</a>
```

- Test the app by creating, editing, and deleting a contact

Seed the database

Add the [SeedData](#) class to the *Data* folder:

C#

 Copy

```
using ContactManager.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;
using System.Threading.Tasks;

// dotnet aspnet-codegenerator razorpage -m Contact -dc ApplicationDbContext -udl -outDir Pages\Contacts --
// referenceScriptLibraries

namespace ContactManager.Data
{
    public static class SeedData
    {
        public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
            {
                SeedDB(context, "0");
            }
        }

        public static void SeedDB(ApplicationDbContext context, string adminID)
        {
            if (context.Contact.Any())
            {
                return; // DB has been seeded
            }
        }
    }
}
```