option isn't intended to be a full-featured identity service provider or token server, but instead an alternative to the cookie option for clients that can't use cookies.

To use token-based authentication, set the `useCookies` query string parameter to `false` when calling the `/login` endpoint. Tokens use the *bearer* authentication scheme. Using the token returned from the call to `/login`, subsequent calls to protected endpoints should add the header `Authorization: Bearer <token>` where `<token>` is the access token. For more information, see Use the POST /login endpoint later in this article.

Log out

To provide a way for the user to log out, define a `/logout` endpoint like the following example:

C#

```csharp
app.MapPost("/logout", async (SignInManager<IdentityUser> signInManager,
    [FromBody] object empty) =>
{
    if (empty != null)
    {
        await signInManager.SignOutAsync();
        return Results.Ok();
    }
    return Results.Unauthorized();
})
.WithOpenApi()
.RequireAuthorization();
```

Provide an empty JSON object (`{}`) in the request body when calling this endpoint. The following code is an example of a call to the logout endpoint:

TypeScript

```typescript
public signOut() {
  return this.http.post('/logout', {}, {
    withCredentials: true,
    observe: 'response',
    responseType: 'text'
```

The `MapIdentityApi<TUser>` endpoints

The call to `MapIdentityApi<TUser>` adds the following endpoints to the app:

- POST /register
- POST /login
- POST /refresh
- GET /confirmEmail
- POST /resendConfirmationEmail
- POST /forgotPassword
- POST /resetPassword
- POST /manage/2fa
- GET /manage/info
- POST /manage/info

Use the `POST /register` endpoint

The request body must have Email and Password properties:

JSON

```json
{
  "email": "string",
  "password": "string",
}
```

For more information, see:

- Test registration earlier in this article.
- RegisterRequest.

Use the `POST /login` endpoint

In the request body, Email and Password are required. If two-factor authentication (2FA) is enabled, either TwoFactorCode or TwoFactorRecoveryCode is required. If 2FA isn't enabled, omit both `twoFactorCode` and `twoFactorRecoveryCode`. For more information, see Use the POST /manage/2fa endpoint later in this article.

Here's a request body example with 2FA not enabled:

JSON

```json
{
  "email": "string",
  "password": "string"
}
```

Here are request body examples with 2FA enabled:

- JSON

  ```json
  {
    "email": "string",
    "password": "string",
    "twoFactorCode": "string",
  }
  ```

- JSON

  ```json
  {
    "email": "string",
    "password": "string",
    "twoFactorRecoveryCode": "string"
  }
  ```

The endpoint expects a query string parameter:

- `useCookies` - Set to `true` for cookie-based authentication. Set to `false` or omit for token-based authentication.

For more information about cookie-based authentication, see Test login earlier in this article.

Token-based authentication

If `useCookies` is `false` or omitted, token-based authentication is enabled. The response body includes the following properties:

JSON

```json
{
  "tokenType": "string",
  "accessToken": "string",
  "expiresIn": 0,
  "refreshToken": "string"
}
```

For more information about these properties, see AccessTokenResponse.

Put the access token in a header to make authenticated requests, as shown in the following example

HTTP

```
Authorization: Bearer {access token}
```

When the access token is about to expire, call the /refresh endpoint.

Use the `POST /refresh` endpoint

For use only with token-based authentication. Gets a new access token without forcing the user to log in again. Call this endpoint when the access token is about to expire.

The request body contains only the RefreshToken. Here's a request body example:

```json
{
  "refreshToken": "string"
}
```

If the call is successful, the response body is a new AccessTokenResponse, as shown in the following example:

```json
{
  "tokenType": "string",
  "accessToken": "string",
  "expiresIn": 0,
  "refreshToken": "string"
}
```

Use the `GET /confirmEmail` endpoint

If Identity is set up for email confirmation, a successful call to the `/register` endpoint sends an email that contains a link to the `/confirmEmail` endpoint. The link contains the following query string parameters:

- `userId`
- `code`
- `changedEmail` - Included only if the user changed the email address during registration.

Identity provides default text for the confirmation email. By default, the email subject is "Confirm your email" and the email body looks like the following example:

HTTP

```
 Please confirm your account by <a href='https://contoso.com/confirmEmail?
userId={user ID}&code={generated code}&changedEmail={new email
address}'>clicking here</a>.
```

If the RequireConfirmedEmail property is set to `true`, the user can't log in until the email address is confirmed by clicking the link in the email. The `/confirmEmail` endpoint:

- Confirms the email address and enables the user to log in.
- Returns the text "Thank you for confirming your email." in the response body.

To set up Identity for email confirmation, add code in `Program.cs` to set `RequireConfirmedEmail` to `true` and add a class that implements IEmailSender to the DI container. For example:

C#

```
builder.Services.Configure<IdentityOptions>(options =>
{
    options.SignIn.RequireConfirmedEmail = true;
});

builder.Services.AddTransient<IEmailSender, EmailSender>();
```

For more information, see Account confirmation and password recovery in ASP.NET Core.

Identity provides default text for the other emails that need to be sent as well, such as for 2FA and password reset. To customize these emails, provide a custom implementation of the `IEmailSender` interface. In the preceding example, `EmailSender` is a class that implements `IEmailSender`. For more information, including an example of a class that implements `IEmailSender`, see Account confirmation and password recovery in ASP.NET Core.

Use the `POST /resendConfirmationEmail` endpoint

Sends an email only if the address is valid for a registered user.

The request body contains only the Email. Here's a request body example:

JSON

```
{
  "email": "string"
}
```

For more information, see Use the GET /confirmEmail endpoint earlier in this article.

Use the `POST /forgotPassword` endpoint

Generates an email that contains a password reset code. Send that code to /resetPassword with a new password.

The request body contains only the Email. Here's an example:

```JSON
{
  "email": "string"
}
```

For information about how to enable Identity to send emails, see Use the GET /confirmEmail endpoint.

Use the `POST /resetPassword` endpoint

Call this endpoint after getting a reset code by calling the `/forgotPassword` endpoint.

The request body requires Email, ResetCode, and NewPassword. Here's an example:

```JSON
{
  "email": "string",
  "resetCode": "string",
  "newPassword": "string"
}
```

Use the `POST /manage/2fa` endpoint

Configures two-factor authentication (2FA) for the user. When 2FA is enabled, successful login requires a code produced by an authenticator app in addition to the email address and password.

Enable 2FA

To enable 2FA for the currently authenticated user:

- Call the `/manage/2fa` endpoint, sending an empty JSON object (`{}`) in the request body.
- The response body provides the SharedKey along with some other properties that aren't needed at this point. The shared key is used to set up the authenticator app. Response body example:

  JSON

  ```json
  {
    "sharedKey": "string",
    "recoveryCodesLeft": 0,
    "recoveryCodes": null,
    "isTwoFactorEnabled": false,
    "isMachineRemembered": false
  }
  ```

- Use the shared key to get a Time-based one-time password (TOTP). For more information, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core.
- Call the `/manage/2fa` endpoint, sending the TOTP and `"enable": true` in the request body. For example:

  JSON

  ```json
  {
    "enable": true,
    "twoFactorCode": "string"
  }
  ```

- The response body confirms that IsTwoFactorEnabled is true and provides the RecoveryCodes. The recovery codes are used to log in when the authenticator app isn't available. Response body example after successfully enabling 2FA:

  JSON

  ```json
  {
    "sharedKey": "string",
    "recoveryCodesLeft": 10,
    "recoveryCodes": [
      "string",
      "string",
  ```

```
    "string",
    "string",
    "string",
    "string",
    "string",
    "string",
    "string",
    "string"
  ],
  "isTwoFactorEnabled": true,
  "isMachineRemembered": false
}
```

Log in with 2FA

Call the `/login` endpoint, sending the email address, password, and TOTP in the request body. For example:

```JSON
{
  "email": "string",
  "password": "string",
  "twoFactorCode": "string"
}
```

If the user doesn't have access to the authenticator app, log in by calling the `/login` endpoint with one of the recovery codes that was provided when 2FA was enabled. The request body looks like the following example:

```JSON
{
  "email": "string",
  "password": "string",
  "twoFactorRecoveryCode": "string"
}
```

Reset the recovery codes

To get a new collection of recovery codes, call this endpoint with ResetRecoveryCodes set to `true`. Here's a request body example:

JSON

```
{
  "resetRecoveryCodes": true
}
```

Reset the shared key

To get a new random shared key, call this endpoint with ResetSharedKey set to `true`. Here's a request body example:

```JSON
{
  "resetSharedKey": true
}
```

Resetting the key automatically disables the two-factor login requirement for the authenticated user until it's re-enabled by a later request.

Forget the machine

To clear the cookie "remember me flag" if present, call this endpoint with ForgetMachine set to true. Here's a request body example:

```JSON
{
  "forgetMachine": true
}
```

This endpoint has no impact on token-based authentication.

Use the `GET /manage/info` endpoint

Gets email address and email confirmation status of the logged-in user. Claims were omitted from this endpoint for security reasons. If claims are needed, use the server-side APIs to set up an endpoint for claims. Or instead of sharing all of the users' claims, provide a validation endpoint that accepts a claim and responds whether the user has it.

The request doesn't require any parameters. The response body includes the Email and IsEmailConfirmed properties, as in the following example:

JSON

```
{
  "email": "string",
  "isEmailConfirmed": true
}
```

Use the `POST /manage/info` endpoint

Updates the email address and password of the logged-in user. Send NewEmail, NewPassword, and OldPassword in the request body, as shown in the following example:

```JSON
{
  "newEmail": "string",
  "newPassword": "string",
  "oldPassword": "string"
}
```

Here's an example of the response body:

```JSON
{
  "email": "string",
  "isEmailConfirmed": false
}
```

See also

For more information, see the following resources:

- Choose an identity management solution
- Identity management solutions for .NET web apps
- Simple authorization in ASP.NET Core
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Create an ASP.NET Core app with user data protected by authorization
- Account confirmation and password recovery in ASP.NET Core
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Sample Web API backend for SPAs The .http file shows token-based authentication. For example:
  - Doesn't set `useCookies`

  - Uses the Authorization header to pass the token
  - Shows refresh to extend session without forcing the user to login again
- Sample Angular app that uses Identity to secure a Web API backend

Scaffold Identity in ASP.NET Core projects

Article • 07/23/2024

By Rick Anderson

Blazor Identity scaffolding

ASP.NET Core Identity scaffolding adds ASP.NET Core Identity to Blazor Web Apps and Blazor Server apps. After the scaffolder adds the Identity Razor components to the app, you can customize the components to suit your app's requirements.

Although the scaffolder generates the necessary C# code to scaffold Identity into the app, you must update the project's database with an Entity Framework (EF) Core database migration to complete the process. This article explains the steps required to migrate a database.

Inspect the changes after running the Identity scaffolder. We recommend using GitHub or another source control system that shows file changes with a revert changes feature.

Services are required when using two-factor authentication (2FA), account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually.

Razor Pages and MVC Identity scaffolding

ASP.NET Core provides ASP.NET Core Identity as a Razor class library (RCL). Applications that include Identity can apply the scaffolder to selectively add the source code contained in the Identity RCL. You might want to generate source code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Customized Identity code overrides the default implementation provided by the Identity RCL. To gain full control of the UI and not use the default RCL, see the Create full Identity UI source section.

Applications that do **not** include authentication can apply the scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

Although the scaffolder generates most of the necessary code, you need to update your project to complete the process. This document explains the steps needed to complete an Identity scaffolding update.

We recommend using a source control system that shows file differences and allows you to back out of changes. Inspect the changes after running the Identity scaffolder.

Services are required when using Two Factor Authentication, Account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually. For example, see Require Email Confirmation.

Typically, apps created with individual accounts should ***not*** create a new data context.

Scaffold Identity into a Blazor project

This section applies to Blazor Web Apps and Blazor Server apps.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Blazor Identity** in the center pane. Select the **Add** button.
- In the **Add Blazor Identity** dialog:
  - Select or add with the plus (+) button the database context class (**DbContext class**).
  - Select the database provider (**Database provider**), which defaults to SQL Server.
  - Select or add with the plus (+) button the user class (**User class**).
  - Select the **Add** button.

The generated Identity database code requires EF Core Migrations. The following steps explain how to create and apply a migration to the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

Client-side Blazor apps (Standalone Blazor WebAssembly)

Client-side Blazor apps (Standalone Blazor WebAssembly) use their own Identity UI approaches and can't use ASP.NET Core Identity scaffolding.

For more information, see the Blazor Security and Identity articles.

Scaffold Identity into a Razor project without existing authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Migrations, UseAuthentication, and layout

The generated Identity database code requires Entity Framework (EF) Core Migrations. If a migration to generate the Identity schema hasn't been created and applied to the database, create a migration and update the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

If the Identity schema has already been created but not applied to the database, only the command to update the database must be executed:

Visual Studio

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

You can confirm the application of an Identity schema with the following command. The output of the command includes an "`applied`" column to show which migrations are applied to the database.

Visual Studio

In the Visual Studio **Package Manager Console**, execute Get-Migration:

PowerShell

```powershell
Get-Migration
```

If more than one database context exists, specify the context with the `-Context` parameter.

Layout changes

Optional: Add the login partial (`_LoginPartial`) to the layout file:

CSHTML

```cshtml
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRPnoAuth2Auth</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/WebRPnoAuth2Auth.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">WebRPnoAuth2Auth</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
```

```
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - WebRPnoAuth2Auth - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

Scaffold Identity into a Razor project with authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation).

Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the **+** button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the **+** button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Scaffold Identity into an MVC project without existing authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the **+** button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the **+** button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Optional: Add the login partial (`_LoginPartial`) to the `Views/Shared/_Layout.cshtml` file:

CSHTML

```cshtml
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRPnoAuth2Auth</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/WebRPnoAuth2Auth.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">WebRPnoAuth2Auth</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - WebRPnoAuth2Auth - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The generated Identity database code requires Entity Framework (EF) Core Migrations. If a migration to generate the Identity schema hasn't been created and applied to the database, create a migration and update the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

If the Identity schema has already been created but not applied to the database, only the command to update the database must be executed:

Visual Studio

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

You can confirm the application of an Identity schema with the following command. The output of the command includes an "`applied`" column to show which migrations are applied to the database.

Visual Studio

In the Visual Studio **Package Manager Console**, execute Get-Migration:

PowerShell

```powershell
Get-Migration
```

If more than one database context exists, specify the context with the `-Context` parameter.

Add `MapRazorPages` to `Program.cs` as shown in the following highlighted code:

C#

```csharp
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebMVCauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
```

```
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.MapRazorPages();

app.Run();
```

Scaffold Identity into an MVC project with authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Create full Identity UI source

To maintain full control of the Identity UI, run the Identity scaffolder and select **Override all files**.

Password configuration

If PasswordOptions are configured in `Startup.ConfigureServices`, [StringLength] attribute configuration might be required for the `Password` property in scaffolded Identity pages. `InputModel` `Password` properties are found in the following files:

- `Areas/Identity/Pages/Account/Register.cshtml.cs`
- `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

Disable a page

This section shows how to disable the register page but the approach can be used to disable any page.

To disable user registration:

- Scaffold Identity. Include Account.Register, Account.Login, and Account.RegisterConfirmation. For example:

  .NET CLI

  ```
  dotnet aspnet-codegenerator identity -dc
  RPauth.Data.ApplicationDbContext --files
  "Account.Register;Account.Login;Account.RegisterConfirmation"
  ```

- Update `Areas/Identity/Pages/Account/Register.cshtml.cs` so users can't register from this endpoint:

  C#

  ```
  public class RegisterModel : PageModel
  {
      public IActionResult OnGet()
      {
          return RedirectToPage("Login");
      }

      public IActionResult OnPost()
      {
          return RedirectToPage("Login");
      }
  }
  ```

- Update `Areas/Identity/Pages/Account/Register.cshtml` to be consistent with the preceding changes:

  CSHTML

  ```
  @page
  @model RegisterModel
  @{
      ViewData["Title"] = "Go to Login";
  }

  <h1>@ViewData["Title"]</h1>

  <li class="nav-item">
  ```

```
    <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Login">Login</a>
</li>
```

- Comment out or remove the registration link from `Areas/Identity/Pages/Account/Login.cshtml`

CSHTML

```
@*
<p>
    <a asp-page="./Register" asp-route-
returnUrl="@Model.ReturnUrl">Register as a new user</a>
</p>
*@
```

- Update the `Areas/Identity/Pages/Account/RegisterConfirmation` page.
  - Remove the code and links from the cshtml file.
  - Remove the confirmation code from the `PageModel`:

C#

```
[AllowAnonymous]
  public class RegisterConfirmationModel : PageModel
  {
      public IActionResult OnGet()
      {
          return Page();
      }
  }
```

Use another app to add users

Provide a mechanism to add users outside the web app. Options to add users include:

- A dedicated admin web app.
- A console app.

The following code outlines one approach to adding users:

- A list of users is read into memory.
- A strong unique password is generated for each user.
- The user is added to the Identity database.
- The user is notified and told to change the password.

C#

```
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<AppDbCntx>();
                context.Database.Migrate();

                var config =
host.Services.GetRequiredService<IConfiguration>();
                var userList =
config.GetSection("userList").Get<List<string>>();

                SeedData.Initialize(services, userList).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>
();
                logger.LogError(ex, "An error occurred adding users.");
            }
        }

        host.Run();
    }

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

The following code outlines adding a user:

C#

```
public static async Task Initialize(IServiceProvider serviceProvider,
                                    List<string> userList)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();
```

```
    foreach (var userName in userList)
    {
        var userPassword = GenerateSecurePassword();
        var userId = await EnsureUser(userManager, userName, userPassword);

        NotifyUser(userName, userPassword);
    }
}

private static async Task<string> EnsureUser(UserManager<IdentityUser>
userManager,
                                             string userName, string
userPassword)
{
    var user = await userManager.FindByNameAsync(userName);

    if (user == null)
    {
        user = new IdentityUser(userName)
        {
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, userPassword);
    }

    return user.Id;
}
```

A similar approach can be followed for production scenarios.

Prevent publish of static Identity assets

To prevent publishing static Identity assets to the web root, see Introduction to Identity on ASP.NET Core.

Add, download, and delete custom user data to Identity in an ASP.NET Core project

Article • 05/31/2024

By Rick Anderson

This article shows how to:

- Add custom user data to an ASP.NET Core web app.
- Mark the custom user data model with the PersonalDataAttribute attribute so it's automatically available for download and deletion. Making the data able to be downloaded and deleted helps meet GDPR requirements.

The project sample is created from a Razor Pages web app, but the instructions are similar for an ASP.NET Core MVC web app.

View or download sample code (how to download)

Prerequisites

.NET 6.0 SDK

Create a Razor web app

Visual Studio

- From the Visual Studio **File** menu, select **New** > **Project**. Name the project **WebApp1** if you want to it match the namespace of the download sample code.
- Select **ASP.NET Core Web Application** > **OK**
- Select **Web Application** > **OK**
- Build and run the project.

Run the Identity scaffolder

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, the following options:
  - Select the existing layout file `~/Pages/Shared/_Layout.cshtml`
  - Select the following files to override:
    - **Account/Register**
    - **Account/Manage/Index**
  - Select the **+** button to create a new **Data context class**. Accept the type (**WebApp1.Models.WebApp1Context** if the project is named **WebApp1**).
  - Select the **+** button to create a new **User class**. Accept the type (**WebApp1User** if the project is named **WebApp1**) > **Add**.
- Select **Add**.

Follow the instruction in Migrations, UseAuthentication, and layout to perform the following steps:

- Create a migration and update the database.
- Add `UseAuthentication` to Program.cs
- Add `<partial name="_LoginPartial" />` to the layout file.
- Test the app:
  - Register a user
  - Select the new user name (next to the **Logout** link). You might need to expand the window or select the navigation bar icon to show the user name and other links.
  - Select the **Personal Data** tab.
  - Select the **Download** button and examined the `PersonalData.json` file.
  - Test the **Delete** button, which deletes the logged on user.

Add custom user data to the Identity DB

Update the `IdentityUser` derived class with custom properties. If you named the project WebApp1, the file is named `Areas/Identity/Data/WebApp1User.cs`. Update the file with the following code:

C#

```
using Microsoft.AspNetCore.Identity;

namespace WebApp1.Areas.Identity.Data;

public class WebApp1User : IdentityUser
{
    [PersonalData]
    public string? Name { get; set; }
    [PersonalData]
    public DateTime DOB { get; set; }
}
```

Properties with the PersonalData attribute are:

- Deleted when the `Areas/Identity/Pages/Account/Manage/DeletePersonalData.cshtml` Razor Page calls `UserManager.Delete`.
- Included in the downloaded data by the `Areas/Identity/Pages/Account/Manage/DownloadPersonalData.cshtml` Razor Page.

Update the `Account/Manage/Index.cshtml` page

Update the `InputModel` in `Areas/Identity/Pages/Account/Manage/Index.cshtml.cs` with the following highlighted code:

```C#
public class IndexModel : PageModel
{
    private readonly UserManager<WebApp1User> _userManager;
    private readonly SignInManager<WebApp1User> _signInManager;

    public IndexModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    /// <summary>
    ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
    ///     directly from your code. This API may change or be removed in
future releases.
    /// </summary>
    public string Username { get; set; }
    // Remaining API warnings ommited.

    [TempData]
    public string StatusMessage { get; set; }

    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Phone]
        [Display(Name = "Phone number")]
        public string PhoneNumber { get; set; }
    }

    private async Task LoadAsync(WebApp1User user)
    {
        var userName = await _userManager.GetUserNameAsync(user);
        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

        Username = userName;

        Input = new InputModel
        {
            Name = user.Name,
            DOB = user.DOB,
            PhoneNumber = phoneNumber
        };
    }

    public async Task<IActionResult> OnGetAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID
'{_userManager.GetUserId(User)}'.");
        }

        await LoadAsync(user);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID
'{_userManager.GetUserId(User)}'.");
        }

        if (!ModelState.IsValid)
        {
            await LoadAsync(user);
            return Page();
        }

        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);
        if (Input.PhoneNumber != phoneNumber)
        {
            var setPhoneResult = await
_userManager.SetPhoneNumberAsync(user, Input.PhoneNumber);
            if (!setPhoneResult.Succeeded)
            {
                StatusMessage = "Unexpected error when trying to set phone
number.";
                return RedirectToPage();
            }
        }

        if (Input.Name != user.Name)
        {
            user.Name = Input.Name;
        }

        if (Input.DOB != user.DOB)
        {
            user.DOB = Input.DOB;
        }

        await _userManager.UpdateAsync(user);
        await _signInManager.RefreshSignInAsync(user);
        StatusMessage = "Your profile has been updated";
        return RedirectToPage();
    }
}
```

Update the `Areas/Identity/Pages/Account/Manage/Index.cshtml` with the following highlighted markup:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Profile";
    ViewData["ActivePage"] = ManageNavPages.Index;
}

<h3>@ViewData["Title"]</h3>
<partial name="_StatusMessage" for="StatusMessage" />
<div class="row">
    <div class="col-md-6">
        <form id="profile-form" method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-floating">
                <input asp-for="Username" class="form-control" disabled />
                <label asp-for="Username" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.Name" class="form-control" />
                <label asp-for="Input.Name" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.DOB" class="form-control" />
                <label asp-for="Input.DOB" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.PhoneNumber" class="form-control" />
                <label asp-for="Input.PhoneNumber" class="form-label">
</label>
                <span asp-validation-for="Input.PhoneNumber" class="text-
danger"></span>
            </div>
            <button id="update-profile-button" type="submit" class="w-100
btn btn-lg btn-primary">Save</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Update the `Account/Register.cshtml` page

Update the `InputModel` in `Areas/Identity/Pages/Account/Register.cshtml.cs` with the following highlighted code:

```C#
    public class RegisterModel : PageModel
    {
        private readonly SignInManager<WebApp1User> _signInManager;
        private readonly UserManager<WebApp1User> _userManager;
        private readonly IUserStore<WebApp1User> _userStore;
        private readonly IUserEmailStore<WebApp1User> _emailStore;
        private readonly ILogger<RegisterModel> _logger;
        private readonly IEmailSender _emailSender;

        public RegisterModel(
            UserManager<WebApp1User> userManager,
            IUserStore<WebApp1User> userStore,
            SignInManager<WebApp1User> signInManager,
            ILogger<RegisterModel> logger,
            IEmailSender emailSender)
        {
            _userManager = userManager;
            _userStore = userStore;
            _emailStore = GetEmailStore();
            _signInManager = signInManager;
            _logger = logger;
            _emailSender = emailSender;
        }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        [BindProperty]
        public InputModel Input { get; set; }

        // Remaining API warnings ommited.
        public string ReturnUrl { get; set; }

        public IList<AuthenticationScheme> ExternalLogins { get; set; }

        public class InputModel
        {
            [Required]
            [DataType(DataType.Text)]
            [Display(Name = "Full name")]
            public string Name { get; set; }

            [Required]
            [Display(Name = "Birth Date")]
            [DataType(DataType.Date)]
            public DateTime DOB { get; set; }

            [Required]
            [EmailAddress]
            [Display(Name = "Email")]
            public string Email { get; set; }

            [Required]
            [StringLength(100, ErrorMessage = "The {0} must be at least {2}
and at max {1} characters long.", MinimumLength = 6)]
            [DataType(DataType.Password)]
            [Display(Name = "Password")]
            public string Password { get; set; }
            [DataType(DataType.Password)]
            [Display(Name = "Confirm password")]
            [Compare("Password", ErrorMessage = "The password and
confirmation password do not match.")]
            public string ConfirmPassword { get; set; }
        }


        public async Task OnGetAsync(string returnUrl = null)
        {
            ReturnUrl = returnUrl;
            ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
        }

        public async Task<IActionResult> OnPostAsync(string returnUrl =
null)
        {
            returnUrl ??= Url.Content("~/");
            ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
            if (ModelState.IsValid)
            {
                var user = CreateUser();

                user.Name = Input.Name;
                user.DOB = Input.DOB;

                await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
                await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);
                var result = await _userManager.CreateAsync(user,
Input.Password);

                if (result.Succeeded)
                {
                    _logger.LogInformation("User created a new account with
password.");

                    var userId = await _userManager.GetUserIdAsync(user);
                    var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                    code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                    var callbackUrl = Url.Page(
                        "/Account/ConfirmEmail",
                        pageHandler: null,
                        values: new { area = "Identity", userId = userId,
code = code, returnUrl = returnUrl },
                        protocol: Request.Scheme);

                    await _emailSender.SendEmailAsync(Input.Email, "Confirm
your email",
                        $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                    if (_userManager.Options.SignIn.RequireConfirmedAccount)
                    {
                        return RedirectToPage("RegisterConfirmation", new {
email = Input.Email, returnUrl = returnUrl });
                    }
                    else
                    {
                        await _signInManager.SignInAsync(user, isPersistent:
false);
                        return LocalRedirect(returnUrl);
                    }
                }
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError(string.Empty,
error.Description);
                }
            }

            // If we got this far, something failed, redisplay form
            return Page();
        }

        private WebApp1User CreateUser()
        {
            try
            {
                return Activator.CreateInstance<WebApp1User>();
            }
            catch
            {
                throw new InvalidOperationException($"Can't create an
instance of '{nameof(WebApp1User)}'. " +
                    $"Ensure that '{nameof(WebApp1User)}' is not an abstract
class and has a parameterless constructor, or alternatively " +
                    $"override the register page in
/Areas/Identity/Pages/Account/Register.cshtml");
            }
        }

        private IUserEmailStore<WebApp1User> GetEmailStore()
        {
            if (!_userManager.SupportsUserEmail)
            {
                throw new NotSupportedException("The default UI requires a
user store with email support.");
            }
            return (IUserEmailStore<WebApp1User>)_userStore;
        }
    }
}
```

Update the `Areas/Identity/Pages/Account/Register.cshtml` with the following highlighted markup:

CSHTML

```cshtml
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl"
method="post">
            <h2>Create a new account.</h2>
            <hr />
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>

            <div class="form-floating">
                <input asp-for="Input.Name" class="form-control" />
                <label asp-for="Input.Name"></label>
                <span asp-validation-for="Input.Name" class="text-danger">
</span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.DOB" class="form-control" />
                <label asp-for="Input.DOB"></label>
                <span asp-validation-for="Input.DOB" class="text-danger">
</span>
            </div>

            <div class="form-floating">
                <input asp-for="Input.Email" class="form-control"
autocomplete="username" aria-required="true" />
                <label asp-for="Input.Email"></label>
                <span asp-validation-for="Input.Email" class="text-danger">
</span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.Password" class="form-control"
autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.Password"></label>
                <span asp-validation-for="Input.Password" class="text-
danger"></span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.ConfirmPassword" class="form-control"
autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.ConfirmPassword"></label>
                <span asp-validation-for="Input.ConfirmPassword"
```

```
class="text-danger"></span>
            </div>
            <button id="registerSubmit" type="submit" class="w-100 btn btn-
lg btn-primary">Register</button>
        </form>
    </div>
    <div class="col-md-6 col-md-offset-2">
        <section>
            <h3>Use another service to register.</h3>
            <hr />
            @{
                if ((Model.ExternalLogins?.Count ?? 0) == 0)
                {
                    <div>
                        <p>
                            There are no external authentication services
configured. See this <a href="https://go.microsoft.com/fwlink/?
LinkID=532715">article
                            about setting up this ASP.NET application to
support logging in via external services</a>.
                        </p>
                    </div>
                }
                else
                {
                    <form id="external-account" asp-page="./ExternalLogin"
asp-route-returnUrl="@Model.ReturnUrl" method="post" class="form-
horizontal">
                        <div>
                            <p>
                                @foreach (var provider in
Model.ExternalLogins!)
                                {
                                    <button type="submit" class="btn btn-
primary" name="provider" value="@provider.Name" title="Log in using your
@provider.DisplayName account">@provider.DisplayName</button>
                                }
                            </p>
                        </div>
                    </form>
                }
            }
        </section>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Build the project.

Update the layout

See Layout changes for instructions to add sign-in and sign-out links to every page.

Add a migration for the custom user data

Visual Studio

In the Visual Studio **Package Manager Console**:

PowerShell

```powershell
Add-Migration CustomUserData
Update-Database
```

Test create, view, download, delete custom user data

Test the app:

- Register a new user.
- View the custom user data on the `/Identity/Account/Manage` page.
- Download and view the users personal data from the `/Identity/Account/Manage/PersonalData` page.

Identity model customization in ASP.NET Core

Article • 10/30/2024

By Arthur Vickers

ASP.NET Core Identity provides a framework for managing and storing user accounts in ASP.NET Core apps. Identity is added to your project when **Individual User Accounts** is selected as the authentication mechanism. By default, Identity makes use of an Entity Framework (EF) Core data model. This article describes how to customize the Identity model.

Identity and EF Core Migrations

Before examining the model, it's useful to understand how Identity works with EF Core Migrations to create and update a database. At the top level, the process is:

1. Define or update a data model in code.
2. Add a Migration to translate this model into changes that can be applied to the database.
3. Check that the Migration correctly represents your intentions.
4. Apply the Migration to update the database to be in sync with the model.
5. Repeat steps 1 through 4 to further refine the model and keep the database in sync.

Use one of the following approaches to add and apply Migrations:

- The **Package Manager Console** (PMC) window if using Visual Studio. For more information, see EF Core PMC tools.
- The .NET CLI if using the command line. For more information, see EF Core .NET command line tools.
- Clicking the **Apply Migrations** button on the error page when the app is run.

ASP.NET Core has a development-time error page handler. The handler can apply migrations when the app is run. Production apps typically generate SQL scripts from the migrations and deploy database changes as part of a controlled app and database deployment.

When a new app using Identity is created, steps 1 and 2 above have already been completed. That is, the initial data model already exists, and the initial migration has been added to the project. The initial migration still needs to be applied to the database. The initial migration can be applied via one of the following approaches:

- Run `Update-Database` in PMC.
- Run `dotnet ef database update` in a command shell.
- Click the **Apply Migrations** button on the error page when the app is run.

Repeat the preceding steps as changes are made to the model.

The Identity model

Entity types

The Identity model consists of the following entity types.

Expand table

Entity type	Description
`User`	Represents the user.
`Role`	Represents a role.
`UserClaim`	Represents a claim that a user possesses.
`UserToken`	Represents an authentication token for a user.
`UserLogin`	Associates a user with a login.
`RoleClaim`	Represents a claim that's granted to all users within a role.
`UserRole`	A join entity that associates users and roles.

Entity type relationships

The entity types are related to each other in the following ways:

- Each `User` can have many `UserClaims`.
- Each `User` can have many `UserLogins`.
- Each `User` can have many `UserTokens`.
- Each `Role` can have many associated `RoleClaims`.
- Each `User` can have many associated `Roles`, and each `Role` can be associated with many `Users`. This is a many-to-many relationship that requires a join table in the

database. The join table is represented by the `UserRole` entity.

Default model configuration

Identity defines many *context classes* that inherit from DbContext to configure and use the model. This configuration is done using the EF Core Code First Fluent API in the OnModelCreating method of the context class. The default configuration is:

C#

```
builder.Entity<TUser>(b =>
{
    // Primary key
    b.HasKey(u => u.Id);

    // Indexes for "normalized" username and email, to allow efficient
lookups
    b.HasIndex(u =>
u.NormalizedUserName).HasName("UserNameIndex").IsUnique();
    b.HasIndex(u => u.NormalizedEmail).HasName("EmailIndex");

    // Maps to the AspNetUsers table
    b.ToTable("AspNetUsers");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(u => u.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.UserName).HasMaxLength(256);
    b.Property(u => u.NormalizedUserName).HasMaxLength(256);
    b.Property(u => u.Email).HasMaxLength(256);
    b.Property(u => u.NormalizedEmail).HasMaxLength(256);

    // The relationships between User and other entity types
    // Note that these relationships are configured with no navigation
properties

    // Each User can have many UserClaims
    b.HasMany<TUserClaim>().WithOne().HasForeignKey(uc =>
uc.UserId).IsRequired();

    // Each User can have many UserLogins
    b.HasMany<TUserLogin>().WithOne().HasForeignKey(ul =>
ul.UserId).IsRequired();

    // Each User can have many UserTokens
    b.HasMany<TUserToken>().WithOne().HasForeignKey(ut =>
ut.UserId).IsRequired();

    // Each User can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur =>
ur.UserId).IsRequired();
```

```
});

builder.Entity<TUserClaim>(b =>
{
    // Primary key
    b.HasKey(uc => uc.Id);

    // Maps to the AspNetUserClaims table
    b.ToTable("AspNetUserClaims");
});

builder.Entity<TUserLogin>(b =>
{
    // Composite primary key consisting of the LoginProvider and the key to
use
    // with that provider
    b.HasKey(l => new { l.LoginProvider, l.ProviderKey });

    // Limit the size of the composite key columns due to common DB
restrictions
    b.Property(l => l.LoginProvider).HasMaxLength(128);
    b.Property(l => l.ProviderKey).HasMaxLength(128);

    // Maps to the AspNetUserLogins table
    b.ToTable("AspNetUserLogins");
});

builder.Entity<TUserToken>(b =>
{
    // Composite primary key consisting of the UserId, LoginProvider and
Name
    b.HasKey(t => new { t.UserId, t.LoginProvider, t.Name });

    // Limit the size of the composite key columns due to common DB
restrictions
    b.Property(t => t.LoginProvider).HasMaxLength(maxKeyLength);
    b.Property(t => t.Name).HasMaxLength(maxKeyLength);

    // Maps to the AspNetUserTokens table
    b.ToTable("AspNetUserTokens");
});

builder.Entity<TRole>(b =>
{
    // Primary key
    b.HasKey(r => r.Id);

    // Index for "normalized" role name to allow efficient lookups
    b.HasIndex(r => r.NormalizedName).HasName("RoleNameIndex").IsUnique();

    // Maps to the AspNetRoles table
    b.ToTable("AspNetRoles");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(r => r.ConcurrencyStamp).IsConcurrencyToken();
    // Limit the size of columns to use efficient database types
    b.Property(u => u.Name).HasMaxLength(256);
    b.Property(u => u.NormalizedName).HasMaxLength(256);

    // The relationships between Role and other entity types
    // Note that these relationships are configured with no navigation
properties

    // Each Role can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur =>
ur.RoleId).IsRequired();

    // Each Role can have many associated RoleClaims
    b.HasMany<TRoleClaim>().WithOne().HasForeignKey(rc =>
rc.RoleId).IsRequired();
});

builder.Entity<TRoleClaim>(b =>
{
    // Primary key
    b.HasKey(rc => rc.Id);

    // Maps to the AspNetRoleClaims table
    b.ToTable("AspNetRoleClaims");
});

builder.Entity<TUserRole>(b =>
{
    // Primary key
    b.HasKey(r => new { r.UserId, r.RoleId });

    // Maps to the AspNetUserRoles table
    b.ToTable("AspNetUserRoles");
});
```

Model generic types

Identity defines default Common Language Runtime (CLR) types for each of the entity types listed above. These types are all prefixed with *Identity*:

- `IdentityUser`
- `IdentityRole`
- `IdentityUserClaim`
- `IdentityUserToken`
- `IdentityUserLogin`
- `IdentityRoleClaim`
- `IdentityUserRole`

Rather than using these types directly, the types can be used as base classes for the app's own types. The `DbContext` classes defined by Identity are generic, such that different CLR types can be used for one or more of the entity types in the model. These generic types also allow the `User` primary key (PK) data type to be changed.

When using Identity with support for roles, an IdentityDbContext class should be used. For example:

```C#
// Uses all the built-in Identity types
// Uses `string` as the key type
public class IdentityDbContext
    : IdentityDbContext<IdentityUser, IdentityRole, string>
{
}

// Uses the built-in Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityDbContext<TUser>
    : IdentityDbContext<TUser, IdentityRole, string>
        where TUser : IdentityUser
{
}

// Uses the built-in Identity types except with custom User and Role types
// The key type is defined by TKey
public class IdentityDbContext<TUser, TRole, TKey> : IdentityDbContext<
    TUser, TRole, TKey, IdentityUserClaim<TKey>, IdentityUserRole<TKey>,
    IdentityUserLogin<TKey>, IdentityRoleClaim<TKey>,
IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TRole : IdentityRole<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic
arguments
// The key type is defined by TKey
public abstract class IdentityDbContext<
    TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim,
TUserToken>
    : IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken>
         where TUser : IdentityUser<TKey>
         where TRole : IdentityRole<TKey>
         where TKey : IEquatable<TKey>
         where TUserClaim : IdentityUserClaim<TKey>
         where TUserRole : IdentityUserRole<TKey>
         where TUserLogin : IdentityUserLogin<TKey>
```

```
            where TRoleClaim : IdentityRoleClaim<TKey>
            where TUserToken : IdentityUserToken<TKey>
```

It's also possible to use Identity without roles (only claims), in which case an IdentityUserContext<TUser> class should be used:

```C#
// Uses the built-in non-role Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityUserContext<TUser>
    : IdentityUserContext<TUser, string>
        where TUser : IdentityUser
{
}

// Uses the built-in non-role Identity types except with a custom User type
// The key type is defined by TKey
public class IdentityUserContext<TUser, TKey> : IdentityUserContext<
    TUser, TKey, IdentityUserClaim<TKey>, IdentityUserLogin<TKey>,
    IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic
arguments, with no roles
// The key type is defined by TKey
public abstract class IdentityUserContext<
    TUser, TKey, TUserClaim, TUserLogin, TUserToken> : DbContext
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TUserToken : IdentityUserToken<TKey>
{
}
```

Customize the model

The starting point for model customization is to derive from the appropriate context type. See the Model generic types section. This context type is customarily called `ApplicationDbContext` and is created by the ASP.NET Core templates.

The context is used to configure the model in two ways:

- Supplying entity and key types for the generic type parameters.

- Overriding `OnModelCreating` to modify the mapping of these types.

When overriding `OnModelCreating`, `base.OnModelCreating` should be called first; the overriding configuration should be called next. EF Core generally has a last-one-wins policy for configuration. For example, if the `ToTable` method for an entity type is called first with one table name and then again later with a different table name, the table name in the second call is used.

NOTE: If the `DbContext` doesn't derive from `IdentityDbContext`, `AddEntityFrameworkStores` may not infer the correct POCO types for `TUserClaim`, `TUserLogin`, and `TUserToken`. If `AddEntityFrameworkStores` doesn't infer the correct POCO types, a workaround is to directly add the correct types via `services.AddScoped<IUser/RoleStore<TUser>` and `UserStore<...>>`.

Custom user data

Custom user data is supported by inheriting from `IdentityUser`. It's customary to name this type `ApplicationUser`:

C#

```csharp
public class ApplicationUser : IdentityUser
{
    public string CustomTag { get; set; }
}
```

Use the `ApplicationUser` type as a generic argument for the context:

C#

```csharp
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }
}
```

There's no need to override `OnModelCreating` in the `ApplicationDbContext` class. EF Core maps the `CustomTag` property by convention. However, the database needs to be updated to create a new `CustomTag` column. To create the column, add a migration, and then update the database as described in Identity and EF Core Migrations.

Update `Pages/Shared/_LoginPartial.cshtml` and replace `IdentityUser` with `ApplicationUser`:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@using WebApp1.Areas.Identity.Data
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
```

Update `Areas/Identity/IdentityHostingStartup.cs` or `Startup.ConfigureServices` and replace `IdentityUser` with `ApplicationUser`.

C#

```
services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

Calling AddDefaultIdentity is equivalent to the following code:

C#

```
services.AddAuthentication(o =>
{
    o.DefaultScheme = IdentityConstants.ApplicationScheme;
    o.DefaultSignInScheme = IdentityConstants.ExternalScheme;
})
.AddIdentityCookies(o => { });

services.AddIdentityCore<TUser>(o =>
{
    o.Stores.MaxLengthForKeys = 128;
    o.SignIn.RequireConfirmedAccount = true;
})
.AddDefaultUI()
.AddDefaultTokenProviders();
```

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to

AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`. For more information, see:

- Scaffold Identity
- Add, download, and delete custom user data to Identity

Change the primary key type

A change to the PK column's data type after the database has been created is problematic on many database systems. Changing the PK typically involves dropping and re-creating the table. Therefore, key types should be specified in the initial migration when the database is created.

Follow these steps to change the PK type:

1. If the database was created before the PK change, run `Drop-Database` (PMC) or `dotnet ef database drop` (.NET CLI) to delete it.

2. After confirming deletion of the database, remove the initial migration with `Remove-Migration` (PMC) or `dotnet ef migrations remove` (.NET CLI).

3. Update the `ApplicationDbContext` class to derive from IdentityDbContext<TUser,TRole,TKey>. Specify the new key type for `TKey`. For example, to use a `Guid` key type:

   C#

   ```
   public class ApplicationDbContext
       : IdentityDbContext<IdentityUser<Guid>, IdentityRole<Guid>, Guid>
   {
       public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
   options)
           : base(options)
       {
       }
   }
   ```

   In the preceding code, the generic classes IdentityUser<TKey> and IdentityRole<TKey> must be specified to use the new key type.

   `Startup.ConfigureServices` must be updated to use the generic user:

   C#

   ```
   services.AddDefaultIdentity<IdentityUser<Guid>>(options =>
   options.SignIn.RequireConfirmedAccount = true)
   ```

```
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

4. If a custom `ApplicationUser` class is being used, update the class to inherit from `IdentityUser`. For example:

C#

```
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationUser` class:

C#

```
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

C#

```
services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

The primary key's data type is inferred by analyzing the DbContext object.

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

5. If a custom `ApplicationRole` class is being used, update the class to inherit from `IdentityRole<TKey>`. For example:

```csharp
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationRole : IdentityRole<Guid>
{
    public string Description { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationRole` class. For example, the following class references a custom `ApplicationUser` and a custom `ApplicationRole`:

```csharp
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultUI()
            .AddDefaultTokenProviders();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

The primary key's data type is inferred by analyzing the DbContext object.

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

Add navigation properties

Changing the model configuration for relationships can be more difficult than making other changes. Care must be taken to replace the existing relationships rather than create new, additional relationships. In particular, the changed relationship must specify the same foreign key (FK) property as the existing relationship. For example, the relationship between `Users` and `UserClaims` is, by default, specified as follows:

C#

```
builder.Entity<TUser>(b =>
{
    // Each User can have many UserClaims
    b.HasMany<TUserClaim>()
     .WithOne()
     .HasForeignKey(uc => uc.UserId)
     .IsRequired();
});
```

The FK for this relationship is specified as the `UserClaim.UserId` property. `HasMany` and `WithOne` are called without arguments to create the relationship without navigation properties.

Add a navigation property to `ApplicationUser` that allows associated `UserClaims` to be referenced from the user:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set;
}
}
```

The `TKey` for `IdentityUserClaim<TKey>` is the type specified for the PK of users. In this case, `TKey` is `string` because the defaults are being used. It's **not** the PK type for the `UserClaim` entity type.

Now that the navigation property exists, it must be configured in `OnModelCreating`:

```C#
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();
        });
    }
}
```

Notice that relationship is configured exactly as it was before, only with a navigation property specified in the call to `HasMany`.

The navigation properties only exist in the EF model, not the database. Because the FK for the relationship hasn't changed, this kind of model change doesn't require the database to be updated. This can be checked by adding a migration after making the change. The `Up` and `Down` methods are empty.

Add all User navigation properties

Using the section above as guidance, the following example configures unidirectional navigation properties for all relationships on User:

```C#
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; 
}
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; 
}
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; 
}
    public virtual ICollection<IdentityUserRole<string>> UserRoles { get; 
set; }
}
```

```C#
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> 
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne()
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });
    }
}
```

Add User and Role navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships on User and Role:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set;
}
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set;
}
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set;
}
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}
```

C#

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        IdentityUserClaim<string>, ApplicationUserRole,
IdentityUserLogin<string>,
        IdentityRoleClaim<string>, IdentityUserToken<string>>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();
        });

    }
}
```

Notes:

- This example also includes the `UserRole` join entity, which is needed to navigate the many-to-many relationship from Users to Roles.
- Remember to change the types of the navigation properties to reflect that `Application{...}` types are now being used instead of `Identity{...}` types.

- Remember to use the `Application{...}` in the generic `ApplicationContext` definition.

Add all navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships on all entity types:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<ApplicationUserClaim> Claims { get; set; }
    public virtual ICollection<ApplicationUserLogin> Logins { get; set; }
    public virtual ICollection<ApplicationUserToken> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
    public virtual ICollection<ApplicationRoleClaim> RoleClaims { get; set;
}
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserClaim : IdentityUserClaim<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationUserLogin : IdentityUserLogin<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationRoleClaim : IdentityRoleClaim<string>
{
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserToken : IdentityUserToken<string>
{
    public virtual ApplicationUser User { get; set; }
}
```

C#

```csharp
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        ApplicationUserClaim, ApplicationUserRole, ApplicationUserLogin,
        ApplicationRoleClaim, ApplicationUserToken>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne(e => e.User)
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne(e => e.User)
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne(e => e.User)
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();

            // Each Role can have many associated RoleClaims
            b.HasMany(e => e.RoleClaims)
                .WithOne(e => e.Role)
                .HasForeignKey(rc => rc.RoleId)
                .IsRequired();
        });
    }
}
```

Use composite keys

The preceding sections demonstrated changing the type of key used in the Identity model. Changing the Identity key model to use composite keys isn't supported or recommended. Using a composite key with Identity involves changing how the Identity manager code interacts with the model. This customization is beyond the scope of this document.

Change table/column names and facets

To change the names of tables and columns, call `base.OnModelCreating`. Then, add configuration to override any of the defaults. For example, to change the name of all the Identity tables:

```C#
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.ToTable("MyUsers");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.ToTable("MyUserClaims");
    });

    modelBuilder.Entity<IdentityUserLogin<string>>(b =>
    {
        b.ToTable("MyUserLogins");
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.ToTable("MyUserTokens");
    });
    modelBuilder.Entity<IdentityRole>(b =>
    {
        b.ToTable("MyRoles");
    });

    modelBuilder.Entity<IdentityRoleClaim<string>>(b =>
    {
        b.ToTable("MyRoleClaims");
    });

    modelBuilder.Entity<IdentityUserRole<string>>(b =>
    {
        b.ToTable("MyUserRoles");
    });
}
```

These examples use the default Identity types. If using an app type such as `ApplicationUser`, configure that type instead of the default type.

The following example changes some column names:

C#

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

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(e => e.Email).HasColumnName("EMail");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.Property(e => e.ClaimType).HasColumnName("CType");
        b.Property(e => e.ClaimValue).HasColumnName("CValue");
    });
}
```

Some types of database columns can be configured with certain *facets* (for example, the maximum `string` length allowed). The following example sets column maximum lengths for several `string` properties in the model:

C#

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

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(u => u.UserName).HasMaxLength(128);
        b.Property(u => u.NormalizedUserName).HasMaxLength(128);
        b.Property(u => u.Email).HasMaxLength(128);
        b.Property(u => u.NormalizedEmail).HasMaxLength(128);
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.Property(t => t.LoginProvider).HasMaxLength(128);
        b.Property(t => t.Name).HasMaxLength(128);
    });
}
```

Map to a different schema

Schemas can behave differently across database providers. For SQL Server, the default is to create all tables in the *dbo* schema. The tables can be created in a different schema. For example:

C#

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

    modelBuilder.HasDefaultSchema("notdbo");
}
```

Lazy loading

In this section, support for lazy-loading proxies in the Identity model is added. Lazy-loading is useful since it allows navigation properties to be used without first ensuring they're loaded.

Entity types can be made suitable for lazy-loading in several ways, as described in the EF Core documentation. For simplicity, use lazy-loading proxies, which requires:

- Installation of the Microsoft.EntityFrameworkCore.Proxies package.
- A call to UseLazyLoadingProxies inside AddDbContext.
- Public entity types with `public virtual` navigation properties.

The following example demonstrates calling `UseLazyLoadingProxies` in `Startup.ConfigureServices`:

C#

```csharp
services
    .AddDbContext<ApplicationDbContext>(
        b => b.UseSqlServer(connectionString)
              .UseLazyLoadingProxies())
    .AddDefaultIdentity<ApplicationUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Refer to the preceding examples for guidance on adding navigation properties to the entity types.

⚠ Warning

This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Additional resources

- Scaffold Identity in ASP.NET Core projects

Configure ASP.NET Core Identity

Article • 03/12/2024

ASP.NET Core Identity uses default values for settings such as password policy, lockout, and cookie configuration. These settings can be overridden at application startup.

Identity options

The IdentityOptions class represents the options that can be used to configure the Identity system. IdentityOptions must be set **after** calling AddIdentity or AddDefaultIdentity.

Claims Identity

IdentityOptions.ClaimsIdentity specifies the ClaimsIdentityOptions with the properties shown in the following table.

Expand table

Property	Description	Default
RoleClaimType	Gets or sets the claim type used for a role claim.	ClaimTypes.Role
SecurityStampClaimType	Gets or sets the claim type used for the security stamp claim.	`AspNet.Identity.SecurityStamp`
UserIdClaimType	Gets or sets the claim type used for the user identifier claim.	ClaimTypes.NameIdentifier
UserNameClaimType	Gets or sets the claim type used for the user name claim.	ClaimTypes.Name

Lockout

Lockout is set in the PasswordSignInAsync method:

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set
lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
             Input.Password, Input.RememberMe,
             lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl =
returnUrl, RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

The preceding code is based on the Login Identity template.

Lockout options are set in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>

options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.Configure<IdentityOptions>(options =>
{
    // Default Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code sets the IdentityOptions LockoutOptions with default values.

A successful authentication resets the failed access attempts count and resets the clock.

IdentityOptions.Lockout specifies the LockoutOptions with the properties shown in the table.

Expand table

Property	Description	Default
AllowedForNewUsers	Determines if a new user can be locked out.	`true`

Property	Description	Default
DefaultLockoutTimeSpan	The amount of time a user is locked out when a lockout occurs.	5 minutes
MaxFailedAccessAttempts	The number of failed access attempts until a user is locked out, if lockout is enabled.	5

Password

By default, Identity requires that passwords contain an uppercase character, lowercase character, a digit, and a non-alphanumeric character. Passwords must be at least six characters long.

Passwords are configured with:

- PasswordOptions in `Program.cs`.
- [StringLength] attributes of `Password` properties if Identity is scaffolded into the app. `InputModel` `Password` properties are found in the following files:
  - `Areas/Identity/Pages/Account/Register.cshtml.cs`
  - `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

C#

```csharp
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                options.SignIn.RequireConfirmedAccount =
true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
});

var app = builder.Build();

// Remaining code removed for brevity.
```

IdentityOptions.Password specifies the PasswordOptions with the properties shown in the table.

Expand table

Property	Description	Default
RequireDigit	Requires a number between 0-9 in the password.	`true`
RequiredLength	The minimum length of the password.	6
RequireLowercase	Requires a lowercase character in the password.	`true`
RequireNonAlphanumeric	Requires a non-alphanumeric character in the password.	`true`
RequiredUniqueChars	Only applies to ASP.NET Core 2.0 or later. Requires the number of distinct characters in the password.	1
RequireUppercase	Requires an uppercase character in the password.	`true`

Sign-in

The following code sets `SignIn` settings (to default values):

```C#
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default SignIn settings.
    options.SignIn.RequireConfirmedEmail = false;
    options.SignIn.RequireConfirmedPhoneNumber = false;
});
```

IdentityOptions.SignIn specifies the SignInOptions with the properties shown in the table.

Expand table

Property	Description	Default
RequireConfirmedEmail	Requires a confirmed email to sign in.	`false`
RequireConfirmedPhoneNumber	Requires a confirmed phone number to sign in.	`false`

Tokens

IdentityOptions.Tokens specifies the TokenOptions with the properties shown in the table.

Expand table

Property	Description
AuthenticatorTokenProvider	Gets or sets the `AuthenticatorTokenProvider` used to validate two-factor sign-ins with an authenticator.
ChangeEmailTokenProvider	Gets or sets the `ChangeEmailTokenProvider` used to generate tokens used in email change confirmation emails.
ChangePhoneNumberTokenProvider	Gets or sets the `ChangePhoneNumberTokenProvider` used to generate tokens used when changing phone numbers.
EmailConfirmationTokenProvider	Gets or sets the token provider used to generate tokens used in account confirmation emails.
PasswordResetTokenProvider	Gets or sets the IUserTwoFactorTokenProvider<TUser> used to generate tokens used in password reset emails.
ProviderMap	Used to construct a User Token Provider with the key used as the provider's name.

User

C#

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default User settings.
    options.User.AllowedUserNameCharacters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;

});
```

IdentityOptions.User specifies the UserOptions with the properties shown in the table.

Expand table

Property	Description	Default
AllowedUserNameCharacters	Allowed characters in the username.	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 -._@+
RequireUniqueEmail	Requires each user to have a unique email.	`false`

Cookie settings

Configure the app's cookie in `Program.cs`. ConfigureApplicationCookie must be called **after** calling `AddIdentity` or `AddDefaultIdentity`.

```C#
builder.Services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Identity/Account/Login";
    // ReturnUrlParameter requires
    //using Microsoft.AspNetCore.Authentication.Cookies;
    options.ReturnUrlParameter =
CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

For more information, see CookieAuthenticationOptions.

Password Hasher options

PasswordHasherOptions gets and sets options for password hashing.

Expand table

Option	Description
CompatibilityMode	The compatibility mode used when hashing new passwords. Defaults to IdentityV3. The first byte of a hashed password, called a *format marker*, specifies the version of the hashing algorithm used to hash the password. When verifying a password against a hash, the VerifyHashedPassword method selects the correct algorithm based on the first byte. A client is able to authenticate regardless of which version of the algorithm was used to hash the password. Setting the compatibility mode affects the hashing of *new passwords*.
IterationCount	The number of iterations used when hashing passwords using PBKDF2. This value is only used when the CompatibilityMode is set to IdentityV3. The value must be a positive integer and defaults to `100000`.

In the following example, the IterationCount is set to `12000` in `Program.cs`:

C#

```csharp
// using Microsoft.AspNetCore.Identity;

builder.Services.Configure<PasswordHasherOptions>(option =>
{
    option.IterationCount = 12000;
});
```

Globally require all users to be authenticated

For information on how to globally require all users to be authenticated, see Require authenticated users.

ISecurityStampValidator and SignOut everywhere

Apps need to react to events involving security sensitive actions by regenerating the users ClaimsPrincipal. For example, the `ClaimsPrincipal` should be regenerated when joining a role, changing the password, or other security sensitive events. Identity uses the ISecurityStampValidator interface to regenerate the `ClaimsPrincipal`. The default implementation of Identity registers a SecurityStampValidator with the main application cookie and the two-factor cookie. The validator hooks into the OnValidatePrincipal event of each cookie to call into Identity to verify that the user's security stamp claim is unchanged from what's stored in the cookie. The validator calls in at regular intervals. The call interval is a tradeoff between hitting the datastore too frequently and not often enough. Checking with a long interval results in stale claims. Call `userManager.UpdateSecurityStampAsync(user)` to force existing cookies to be invalided the next time they are checked. Most of the Identity UI account and manage pages call `userManager.UpdateSecurityStampAsync(user)` after changing the password or adding a login. Apps can call `userManager.UpdateSecurityStampAsync(user)` to implement a sign out everywhere action.

Changing the validation interval is shown in the following highlighted code:

C#

```csharp
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebClaimsPrincipal.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("'DefaultConnection' not
found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

// Force Identity's security stamp to be validated every minute.
builder.Services.Configure<SecurityStampValidatorOptions>(o =>
                   o.ValidationInterval = TimeSpan.FromMinutes(1));

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Custom storage providers for ASP.NET Core Identity

Article • 10/30/2024

By Steve Smith

ASP.NET Core Identity is an extensible system which enables you to create a custom storage provider and connect it to your app. This topic describes how to create a customized storage provider for ASP.NET Core Identity. It covers the important concepts for creating your own storage provider, but isn't a step-by-step walk through. See Identity model customization to customize an Identity model.

Introduction

By default, the ASP.NET Core Identity system stores user information in a SQL Server database using Entity Framework Core. For many apps, this approach works well. However, you may prefer to use a different persistence mechanism or data schema. For example:

- You use Azure Table Storage or another data store.
- Your database tables have a different structure.
- You may wish to use a different data access approach, such as Dapper.

In each of these cases, you can write a customized provider for your storage mechanism and plug that provider into your app.

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

When using the .NET CLI, add `-au Individual`:

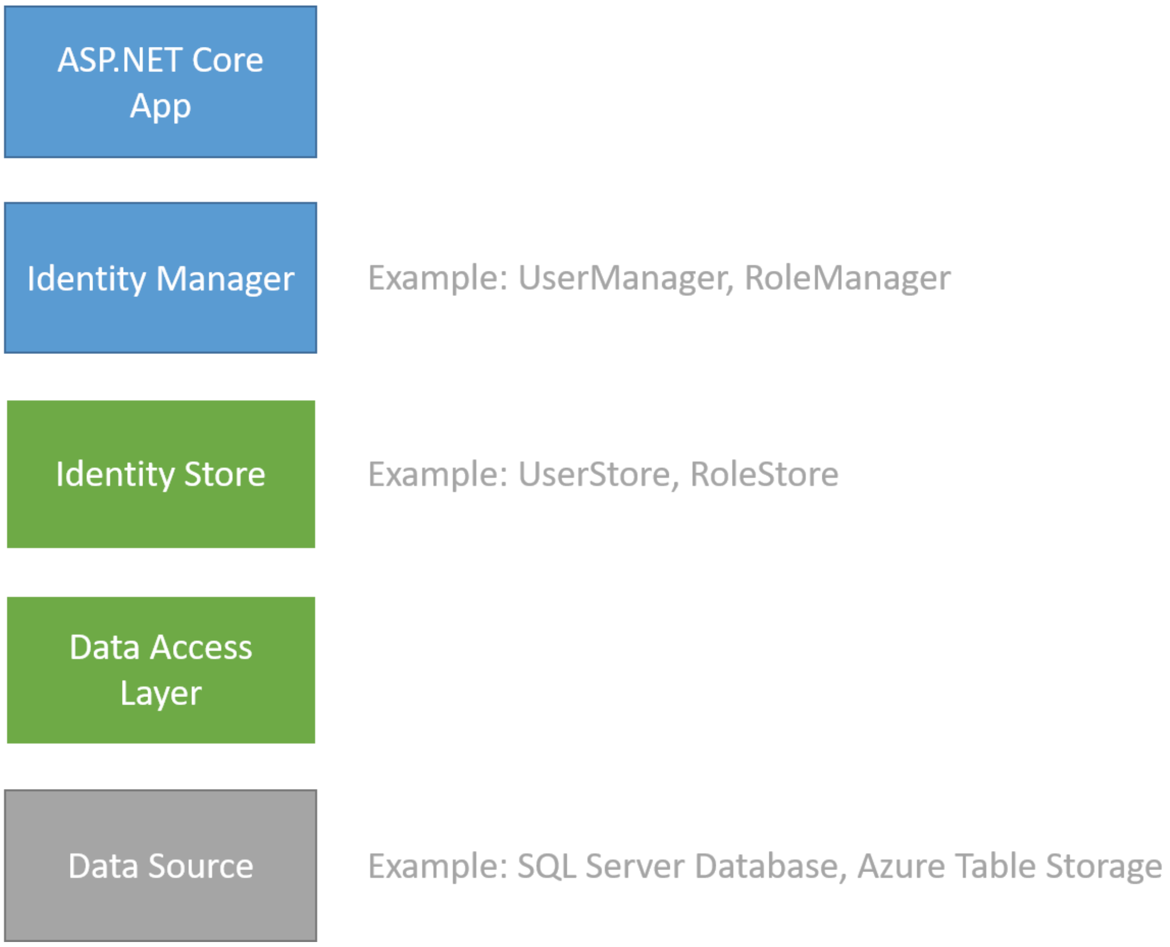
.NET CLI

```
dotnet new mvc -au Individual
```

The ASP.NET Core Identity architecture

ASP.NET Core Identity consists of classes called managers and stores. *Managers* are high-level classes which an app developer uses to perform operations, such as creating an Identity user. *Stores* are lower-level classes that specify how entities, such as users and roles, are persisted. Stores follow the repository pattern and are closely coupled with the persistence mechanism. Managers are decoupled from stores, which means you can replace the persistence mechanism without changing your application code (except for configuration).

The following diagram shows how a web app interacts with the managers, while stores interact with the data access layer.


ASP.NET Core App
Identity Manager
Example: UserManager, RoleManager
Identity Store
Example: UserStore, RoleStore
Data Access Layer
Data Source
Example: SQL Server Database, Azure Table Storage

To create a custom storage provider, create the data source, the data access layer, and the store classes that interact with this data access layer (the green and grey boxes in the diagram above). You don't need to customize the managers or your app code that interacts with them (the blue boxes above).

When creating a new instance of `UserManager` or `RoleManager` you provide the type of the user class and pass an instance of the store class as an argument. This approach enables you to plug your customized classes into ASP.NET Core.

Reconfigure app to use new storage provider shows how to instantiate `UserManager` and `RoleManager` with a customized store.

ASP.NET Core Identity stores data types

ASP.NET Core Identity data types are detailed in the following sections:

Users

Registered users of your web site. The IdentityUser type may be extended or used as an example for your own custom type. You don't need to inherit from a particular type to implement your own custom identity storage solution.

User Claims

A set of statements (or Claims) about the user that represent the user's identity. Can enable greater expression of the user's identity than can be achieved through roles.

User Logins

Information about the external authentication provider (like Facebook or a Microsoft account) to use when logging in a user. Example

Roles

Authorization groups for your site. Includes the role Id and role name (like "Admin" or "Employee"). Example

The data access layer

This topic assumes you are familiar with the persistence mechanism that you are going to use and how to create entities for that mechanism. This topic doesn't provide details about how to create the repositories or data access classes; it provides some suggestions about design decisions when working with ASP.NET Core Identity.

You have a lot of freedom when designing the data access layer for a customized store provider. You only need to create persistence mechanisms for features that you intend to use in your app. For example, if you are not using roles in your app, you don't need to create storage for roles or user role associations. Your technology and existing infrastructure may require a structure that's very different from the default implementation of ASP.NET Core Identity. In your data access layer, you provide the logic to work with the structure of your storage implementation.

The data access layer provides the logic to save the data from ASP.NET Core Identity to a data source. The data access layer for your customized storage provider might include the following classes to store user and role information.

Context class

Encapsulates the information to connect to your persistence mechanism and execute queries. Several data classes require an instance of this class, typically provided through dependency injection. Example.

User Storage

Stores and retrieves user information (such as user name and password hash). Example

Role Storage

Stores and retrieves role information (such as the role name). Example

UserClaims Storage

Stores and retrieves user claim information (such as the claim type and value). Example

UserLogins Storage

Stores and retrieves user login information (such as an external authentication provider). Example

UserRole Storage

Stores and retrieves which roles are assigned to which users. Example

TIP: Only implement the classes you intend to use in your app.

In the data access classes, provide code to perform data operations for your persistence mechanism. For example, within a custom provider, you might have the following code to create a new user in the *store* class:

C#

```csharp
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

The implementation logic for creating the user is in the `_usersTable.CreateAsync` method, shown below.

Customize the user class

When implementing a storage provider, create a user class which is equivalent to the IdentityUser class.

At a minimum, your user class must include an `Id` and a `UserName` property.

The `IdentityUser` class defines the properties that the `UserManager` calls when performing requested operations. The default type of the `Id` property is a string, but you can inherit from `IdentityUser<TKey, TUserClaim, TUserRole, TUserLogin, TUserToken>` and specify a different type. The framework expects the storage implementation to handle data type conversions.

Customize the user store

Create a `UserStore` class that provides the methods for all data operations on the user. This class is equivalent to the UserStore<TUser> class. In your `UserStore` class, implement `IUserStore<TUser>` and the optional interfaces required. You select which optional interfaces to implement based on the functionality provided in your app.

Optional interfaces

- IUserRoleStore
- IUserClaimStore
- IUserPasswordStore
- IUserSecurityStampStore
- IUserEmailStore
- IUserPhoneNumberStore
- IQueryableUserStore
- IUserLoginStore
- IUserTwoFactorStore
- IUserLockoutStore

The optional interfaces inherit from `IUserStore<TUser>`. You can see a partially implemented sample user store in the sample app.

Within the `UserStore` class, you use the data access classes that you created to perform operations. These are passed in using dependency injection. For example, in the SQL Server with Dapper implementation, the `UserStore` class has the `CreateAsync` method which uses an instance of `DapperUsersTable` to insert a new record:

```csharp
public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id,
user.Email, user.EmailConfirmed, user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could
not insert user {user.Email}." });
}
```

Interfaces to implement when customizing user store

- **IUserStore**
  The IUserStore<TUser> interface is the only interface you must implement in the user store. It defines methods for creating, updating, deleting, and retrieving users.
- **IUserClaimStore**
  The IUserClaimStore<TUser> interface defines the methods you implement to enable user claims. It contains methods for adding, removing and retrieving user claims.
- **IUserLoginStore**
  The IUserLoginStore<TUser> defines the methods you implement to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.
- **IUserRoleStore**
  The IUserRoleStore<TUser> interface defines the methods you implement to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.
- **IUserPasswordStore**
  The IUserPasswordStore<TUser> interface defines the methods you implement to

persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.

- **IUserSecurityStampStore**
  The IUserSecurityStampStore<TUser> interface defines the methods you implement to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.
- **IUserTwoFactorStore**
  The IUserTwoFactorStore<TUser> interface defines the methods you implement to support two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.
- **IUserPhoneNumberStore**
  The IUserPhoneNumberStore<TUser> interface defines the methods you implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.
- **IUserEmailStore**
  The IUserEmailStore<TUser> interface defines the methods you implement to store user email addresses. It contains methods for getting and setting the email address and whether the email is confirmed.
- **IUserLockoutStore**
  The IUserLockoutStore<TUser> interface defines the methods you implement to store information about locking an account. It contains methods for tracking failed access attempts and lockouts.
- **IQueryableUserStore**
  The IQueryableUserStore<TUser> interface defines the members you implement to provide a queryable user store.

You implement only the interfaces that are needed in your app. For example:

```C#
public class UserStore : IUserStore<IdentityUser>,
                         IUserClaimStore<IdentityUser>,
                         IUserLoginStore<IdentityUser>,
                         IUserRoleStore<IdentityUser>,
                         IUserPasswordStore<IdentityUser>,
                         IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

The `Microsoft.AspNet.Identity.EntityFramework` namespace contains implementations of the IdentityUserClaim, IdentityUserLogin, and IdentityUserRole classes. If you are using these features, you may want to create your own versions of these classes and define the properties for your app. However, sometimes it's more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the `UserStore.GetClaimsAsync` method can call the `userClaimTable.FindByUserId(user.Id)` method to execute a query on that table directly and return a list of claims.

Customize the role class

When implementing a role storage provider, you can create a custom role type. It need not implement a particular interface, but it must have an `Id` and typically it will have a `Name` property.

The following is an example role class:

```csharp
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

Customize the role store

You can create a `RoleStore` class that provides the methods for all data operations on roles. This class is equivalent to the RoleStore<TRole> class. In the `RoleStore` class, you implement the `IRoleStore<TRole>` and optionally the `IQueryableRoleStore<TRole>` interface.

- **IRoleStore<TRole>**
  The IRoleStore<TRole> interface defines the methods to implement in the role store class. It contains methods for creating, updating, deleting, and retrieving roles.
- **RoleStore<TRole>**
  To customize `RoleStore`, create a class that implements the `IRoleStore<TRole>` interface.

Reconfigure app to use a new storage provider

Once you have implemented a storage provider, you configure your app to use it. If your app used the default provider, replace it with your custom provider.

1. Remove the `Microsoft.AspNetCore.EntityFramework.Identity` NuGet package.
2. If the storage provider resides in a separate project or package, add a reference to it.
3. Replace all references to `Microsoft.AspNetCore.EntityFramework.Identity` with a using statement for the namespace of your storage provider.
4. Change the `AddIdentity` method to use the custom types. You can create your own extension methods for this purpose. See IdentityServiceCollectionExtensions for an example.
5. If you are using Roles, update the `RoleManager` to use your `RoleStore` class.
6. Update the connection string and credentials to your app's configuration.

⚠ **Warning**

This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Example:

C#

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add identity types
```

```
builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
    .AddDefaultTokenProviders();

// Identity Services
builder.Services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>
();
builder.Services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>
();
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddTransient<SqlConnection>(e => new
SqlConnection(connectionString));
builder.Services.AddTransient<DapperUsersTable>();

// additional configuration

builder.Services.AddRazorPages();

var app = builder.Build();
```

References

- Identity model customization
- Custom Storage Providers for ASP.NET 4.x Identity
- ASP.NET Core Identity: This repository includes links to community maintained store providers.
- View or download sample from GitHub.

Account confirmation and password recovery in ASP.NET Core

Article • 09/18/2024

By Rick Anderson, Ponant, and Joe Audette

This tutorial shows how to build an ASP.NET Core app with email confirmation and password reset. This tutorial is **not** a beginning topic. You should be familiar with:

- ASP.NET Core
- Authentication
- Entity Framework Core

For Blazor guidance, which adds to or supersedes the guidance in this article, see Account confirmation and password recovery in ASP.NET Core Blazor.

Prerequisites

- .NET Core 6.0 SDK or later
- Successfully send email from a C# console app.

Create and test a web app with authentication

Run the following commands to create a web app with authentication.

.NET CLI

```
dotnet new webapp -au Individual -o WebPWrecover
cd WebPWrecover
dotnet run
```

Register user with simulated email confirmation

Run the app, select the **Register** link, and register a user. Once registered, you are redirected to the to `/Identity/Account/RegisterConfirmation` page which contains a link to simulate email confirmation:

- Select the `Click here to confirm your account` link.
- Select the **Login** link and sign-in with the same credentials.

- Select the `Hello YourEmail@provider.com!` link, which redirects to the `/Identity/Account/Manage/PersonalData` page.
- Select the **Personal data** tab on the left, and then select **Delete**.

The `Click here to confirm your account` link is displayed because an IEmailSender has not been implemented and registered with the dependency injection container. See the RegisterConfirmation source.

ⓘ **Note**

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Configure an email provider

In this tutorial, SendGrid is used to send email. A SendGrid account and key is needed to send email. We recommend using SendGrid or another email service to send email rather than SMTP. SMTP is difficult to secure and set up correctly.

The SendGrid account may require adding a Sender.

Create a class to fetch the secure email key. For this sample, create `Services/AuthMessageSenderOptions.cs`:

C#

```csharp
namespace WebPWrecover.Services;

public class AuthMessageSenderOptions
{
    public string? SendGridKey { get; set; }
}
```

Configure SendGrid user secrets

Set the `SendGridKey` with the secret-manager tool. For example:

.NET CLI

```
dotnet user-secrets set SendGridKey <key>

Successfully saved SendGridKey to the secret store.
```

On Windows, Secret Manager stores keys/value pairs in a `secrets.json` file in the `%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` directory.

The contents of the `secrets.json` file aren't encrypted. The following markup shows the `secrets.json` file. The `SendGridKey` value has been removed.

JSON

```json
{
  "SendGridKey": "<key removed>"
}
```

For more information, see the Options pattern and configuration.

Install SendGrid

This tutorial shows how to add email notifications through SendGrid, but other email providers can be used.

Install the `SendGrid` NuGet package:

Visual Studio

From the Package Manager Console, enter the following command:

PowerShell

```powershell
Install-Package SendGrid
```

See Get Started with SendGrid for Free to register for a free SendGrid account.

Implement IEmailSender

To Implement `IEmailSender`, create `Services/EmailSender.cs` with code similar to the following:

C#

```
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;

namespace WebPWrecover.Services;

public class EmailSender : IEmailSender
{
    private readonly ILogger _logger;

    public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor,
                       ILogger<EmailSender> logger)
    {
        Options = optionsAccessor.Value;
        _logger = logger;
    }

    public AuthMessageSenderOptions Options { get; } //Set with Secret
Manager.

    public async Task SendEmailAsync(string toEmail, string subject, string
message)
    {
        if (string.IsNullOrEmpty(Options.SendGridKey))
        {
            throw new Exception("Null SendGridKey");
        }
        await Execute(Options.SendGridKey, subject, message, toEmail);
    }

    public async Task Execute(string apiKey, string subject, string message,
string toEmail)
    {
        var client = new SendGridClient(apiKey);
        var msg = new SendGridMessage()
        {
            From = new EmailAddress("Joe@contoso.com", "Password Recovery"),
            Subject = subject,
            PlainTextContent = message,
            HtmlContent = message
        };
        msg.AddTo(new EmailAddress(toEmail));

        // Disable click tracking.
        // See https://sendgrid.com/docs/User_Guide/Settings/tracking.html
        msg.SetClickTracking(false, false);
        var response = await client.SendEmailAsync(msg);
        _logger.LogInformation(response.IsSuccessStatusCode
                               ? $"Email to {toEmail} queued successfully!"
                               : $"Failure Email to {toEmail}");
    }
}
```

Configure app to support email

Add the following code to the `Program.cs` file:

- Add `EmailSender` as a transient service.
- Register the `AuthMessageSenderOptions` configuration instance.

C#

```csharp
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.Run();
```

Disable default account verification when Account.RegisterConfirmation has been scaffolded

This section only applies when `Account.RegisterConfirmation` is scaffolded. Skip this section if you have not scaffolded `Account.RegisterConfirmation`.

The user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in the scaffolded `/Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs` file:

C#

```
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.
#nullable disable

using System;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.WebUtilities;

namespace WebPWrecover.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class RegisterConfirmationModel : PageModel
    {
        private readonly UserManager<IdentityUser> _userManager;
        private readonly IEmailSender _sender;

        public RegisterConfirmationModel(UserManager<IdentityUser>
userManager, IEmailSender sender)
        {
            _userManager = userManager;
            _sender = sender;
        }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public string Email { get; set; }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public bool DisplayConfirmAccountLink { get; set; }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public string EmailConfirmationUrl { get; set; }

        public async Task<IActionResult> OnGetAsync(string email, string
returnUrl = null)
        {
            if (email == null)
            {
                return RedirectToPage("/Index");
            }
            returnUrl = returnUrl ?? Url.Content("~/");

            var user = await _userManager.FindByEmailAsync(email);
            if (user == null)
            {
                return NotFound($"Unable to load user with email
'{email}'.");
            }

            Email = email;
            // Once you add a real email sender, you should remove this code
that lets you confirm the account
            DisplayConfirmAccountLink = false;
            if (DisplayConfirmAccountLink)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                EmailConfirmationUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code, returnUrl = returnUrl },
                    protocol: Request.Scheme);
            }

            return Page();
        }
    }
}
```

This step is only necessary when `Account.RegisterConfirmation` is scaffolded. The non-scaffolded RegisterConfirmation automatically detects when an IEmailSender has been implemented and registered with the dependency injection container.

Register, confirm email, and reset password

Run the web app, and test the account confirmation and password recovery flow.

- Run the app and register a new user
- Check your email for the account confirmation link. See Debug email if you don't get the email.
- Click the link to confirm your email.
- Sign in with your email and password.
- Sign out.

Test password reset

- If you're signed in, select **Logout**.
- Select the **Log in** link and select the **Forgot your password?** link.
- Enter the email you used to register the account.
- An email with a link to reset your password is sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can sign in with your email and new password.

Resend email confirmation

Select the **Resend email confirmation** link on the **Login** page.

Change email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to 5 days:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

builder.Services.ConfigureApplicationCookie(o => {
    o.ExpireTimeSpan = TimeSpan.FromDays(5);
    o.SlidingExpiration = true;
});

var app = builder.Build();

// Code removed for brevity
```

Change all data protection token lifespans

The following code changes all data protection tokens timeout period to 3 hours:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

builder.Services.Configure<DataProtectionTokenProviderOptions>(o =>
       o.TokenLifespan = TimeSpan.FromHours(3));

var app = builder.Build();

// Code removed for brevity.
```

The built in Identity user tokens (see AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs)have a one day timeout .

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day . This section shows how to change the email token lifespan.

Add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions:

```C#
public class CustomEmailConfirmationTokenProvider<TUser>
                                 :  DataProtectorTokenProvider<TUser> where
TUser : class
{
    public CustomEmailConfirmationTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
                                       : base(dataProtectionProvider,
options, logger)
    {

    }
}
public class EmailConfirmationTokenProviderOptions :
DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}
```

Add the custom provider to the service container:

```
C#

using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;
using WebPWrecover.TokenProviders;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(config =>
{
    config.SignIn.RequireConfirmedEmail = true;
    config.Tokens.ProviderMap.Add("CustomEmailConfirmation",
        new TokenProviderDescriptor(
            typeof(CustomEmailConfirmationTokenProvider<IdentityUser>)));
    config.Tokens.EmailConfirmationTokenProvider =
"CustomEmailConfirmation";
}).AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddTransient<CustomEmailConfirmationTokenProvider<IdentityU
ser>>();

builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

var app = builder.Build();

// Code removed for brevity.
```

Debug email

If you can't get email working:

- Set a breakpoint in `EmailSender.Execute` to verify `SendGridClient.SendEmailAsync` is called.

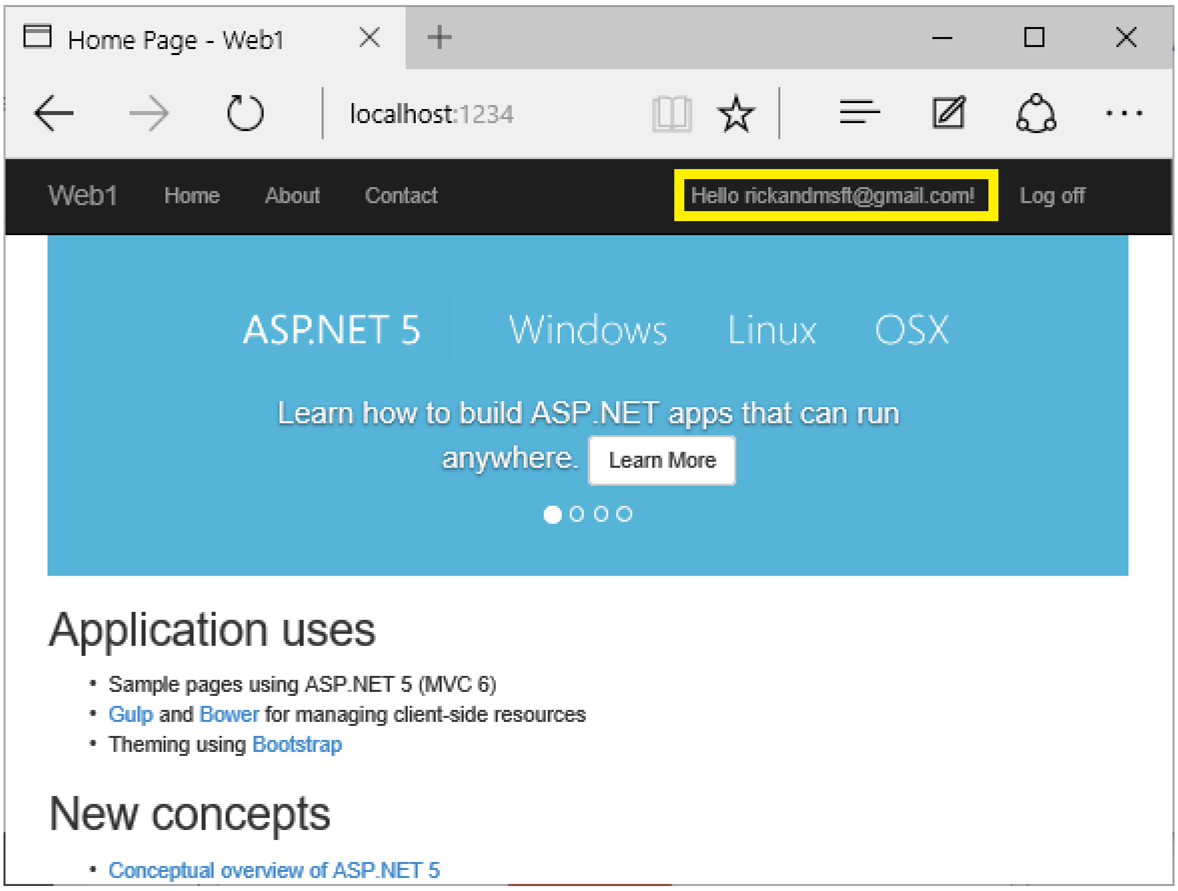
- Create a console app to send email using similar code to `EmailSender.Execute`.
- Review the Email Activity page.
- Check your spam folder.
- Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
- Try sending to different email accounts.

A security best practice is to **not** use production secrets in test and development. If you publish the app to Azure, set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is set up to read keys from environment variables.

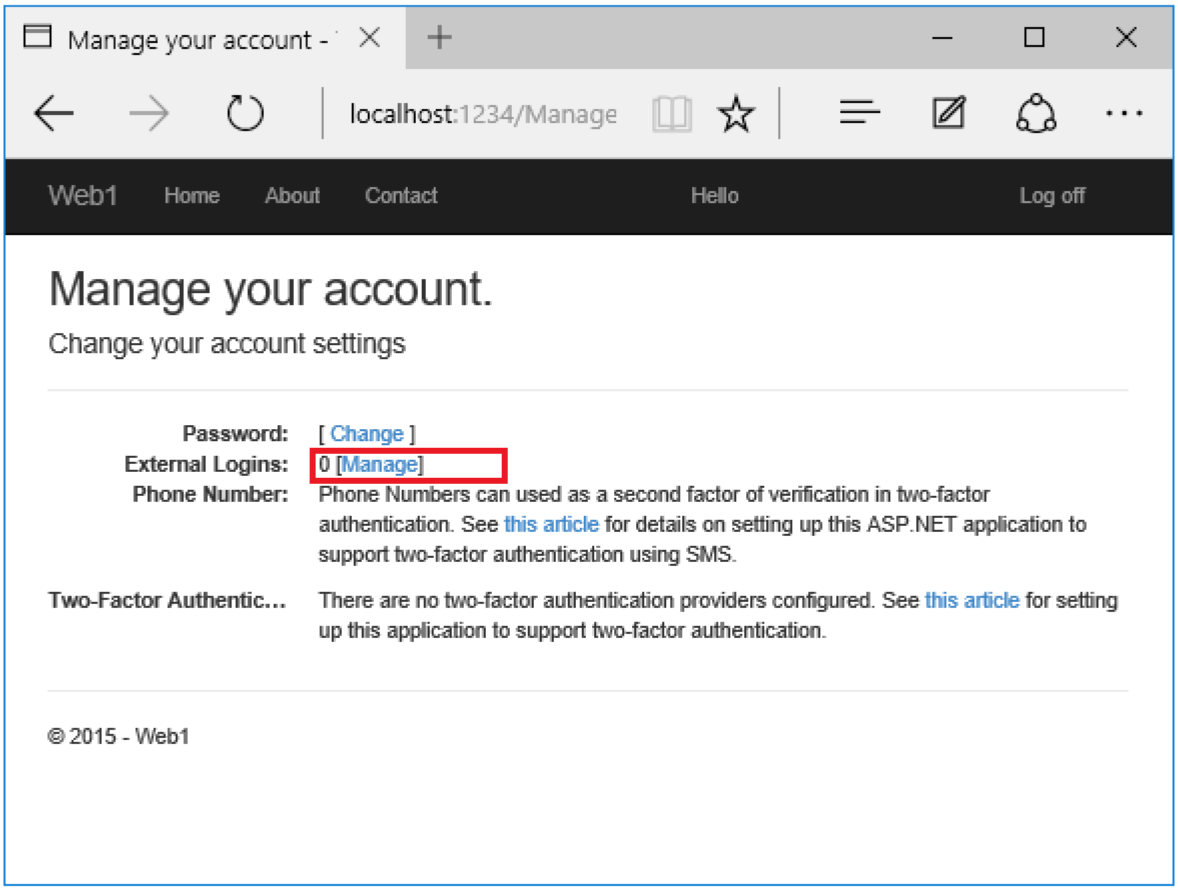
Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See Facebook, Google, and external provider authentication.

You can combine local and social accounts by clicking on your email link. In the following sequence, "RickAndMSFT@gmail.com" is first created as a local login; however, you can create the account as a social login first, then add a local login.


Home Page - Web1
localhost:1234
Web1
Home
About
Contact
Hello rickandmsft@gmail.com!
Log off
ASP.NET 5
Windows
Linux
OSX
Learn how to build ASP.NET apps that can run anywhere.
Learn More
Application uses
Sample pages using ASP.NET 5 (MVC 6)
Gulp and Bower for managing client-side resources
Theming using Bootstrap
New concepts
Conceptual overview of ASP.NET 5

Click on the **Manage** link. Note the 0 external (social logins) associated with this account.


Manage your account.

Change your account settings

Password: [Change]

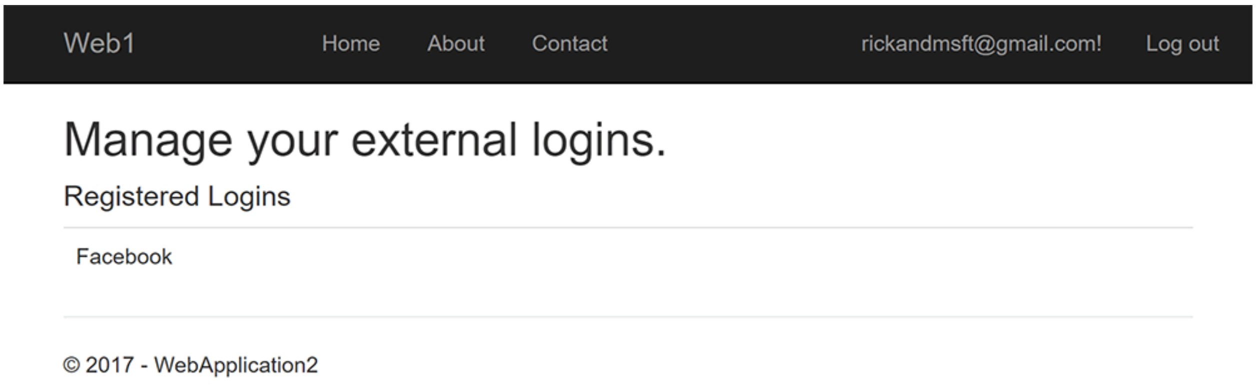
External Logins: 0 [Manage]

Phone Number: Phone Numbers can used as a second factor of verification in two-factor authentication. See this article for details on setting up this ASP.NET application to support two-factor authentication using SMS.

Two-Factor Authentic... There are no two-factor authentication providers configured. See this article for setting up this application to support two-factor authentication.

© 2015 - Web1

Click the link to another login service and accept the app requests. In the following image, Facebook is the external authentication provider:


Manage your external logins.

Registered Logins

Facebook

© 2017 - WebApplication2

The two accounts have been combined. You are able to sign in with either account. You might want your users to add local accounts in case their social login authentication service is down, or more likely they've lost access to their social account.

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as being confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Enable QR code generation for TOTP authenticator apps in ASP.NET Core

Article • 12/11/2024

ASP.NET Core ships with support for authenticator applications for individual authentication. Two factor authentication (2FA) authenticator apps, using a Time-based One-time Password Algorithm (TOTP), are the industry recommended approach for 2FA. 2FA using TOTP is preferred to SMS 2FA. An authenticator app provides a 6 to 8 digit code which users must enter after confirming their username and password. Typically an authenticator app is installed on a smartphone.

⚠ **Warning**

An ASP.NET Core TOTP code should be kept secret because it can be used to authenticate successfully multiple times before it expires.

The ASP.NET Core web app templates support authenticators but don't provide support for QR code generation. QR code generators ease the setup of 2FA. This document provides guidance for Razor Pages and MVC apps on how to add QR code generation to the 2FA configuration page. For guidance that applies to Blazor Web Apps, see Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App. For guidance that applies to Blazor WebAssembly apps, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

Two-factor authentication does not happen using an external authentication provider, such as Google or Facebook. External logins are protected by whatever mechanism the external login provider provides. Consider, for example, the Microsoft authentication provider requires a hardware key or another 2FA approach. If the default templates required 2FA for both the web app and the external authentication provider, then users would be required to satisfy two 2FA approaches. Requiring two 2FA approaches deviates from established security practices, which typically rely on a single, strong 2FA method for authentication.

Adding QR codes to the 2FA configuration page

These instructions use `qrcode.js` from the https://davidshimjs.github.io/qrcodejs/ repo.

- Download the qrcode.js JavaScript library to the `wwwroot\lib` folder in your project.
- Follow the instructions in Scaffold Identity to generate `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml`.
- In `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml`, locate the `Scripts` section at the end of the file:

CSHTML

```cshtml
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- Create a new JavaScript file called `qr.js` in `wwwroot/js` and add the following code to generate the QR Code:

JavaScript

```javascript
window.addEventListener("load", () => {
  const uri = document.getElementById("qrCodeData").getAttribute('data-url');
  new QRCode(document.getElementById("qrCode"),
    {
      text: uri,
      width: 150,
      height: 150
    });
});
```

- Update the `Scripts` section to add a reference to the `qrcode.js` library previously downloaded.
- Add the `qr.js` file with the call to generate the QR code:

CSHTML

```cshtml
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript" src="~/js/qr.js"></script>
}
```

- Delete the paragraph which links you to these instructions.

Run your app and ensure that you can scan the QR code and validate the code the authenticator proves.

Change the site name in the QR code

The site name in the QR code is taken from the project name you choose when initially creating your project. You can change it by looking for the `GenerateQrCodeUri(string email, string unformattedKey)` method in the `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml.cs`.

The default code from the template looks as follows:

```C#
private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenticatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

The second parameter in the call to `string.Format` is your site name, taken from your solution name. It can be changed to any value, but it must always be URL encoded.

Using a different QR Code library

You can replace the QR Code library with your preferred library. The HTML contains a `qrCode` element into which you can place a QR Code by whatever mechanism your library provides.

The correctly formatted URL for the QR Code is available in the:

- `AuthenticatorUri` property of the model.
- `data-url` property in the `qrCodeData` element.

TOTP client and server time skew

TOTP (Time-based One-Time Password) authentication depends on both the server and authenticator device having an accurate time. Tokens only last for 30 seconds. If TOTP

2FA logins are failing, check that the server time is accurate, and preferably synchronized to an accurate NTP service.

Facebook, Google, and external provider authentication in ASP.NET Core

Article • 09/05/2024

By Valeriy Novytskyy and Rick Anderson

This tutorial demonstrates how to build an ASP.NET Core app that enables users to sign in using OAuth 2.0 with credentials from external authentication providers.

Facebook, Twitter, Google, and Microsoft providers are covered in the following sections and use the starter project created in this article. Other providers are available in third-party packages such as OpenIddict, AspNet.Security.OAuth.Providers and AspNet.Security.OpenId.Providers.

Enabling users to sign in with their existing credentials:

- Is convenient for the users.
- Shifts many of the complexities of managing the sign-in process onto a third party.

Create a New ASP.NET Core Project

Visual Studio

- Select the **ASP.NET Core Web App** template. Select **OK**.
- In the **Authentication type** input, select **Individual Accounts**.

Apply migrations

- Run the app and select the **Register** link.
- Enter the email and password for the new account, and then select **Register**.
- Follow the instructions to apply migrations.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use SecretManager to store tokens assigned by login providers

Social login providers assign **Application Id** and **Application Secret** tokens during the registration process. The exact token names vary by provider. These tokens represent the credentials your app uses to access their API. The tokens constitute the "user secrets" that can be linked to your app configuration with the help of Secret Manager. User secrets are a more secure alternative to storing the tokens in a configuration file, such as `appsettings.json`.

ⓘ **Important**

Secret Manager is for development purposes only. You can store and protect Azure test and production secrets with the Azure Key Vault configuration provider.

Follow the steps in Safe storage of app secrets in development in ASP.NET Core topic to store tokens assigned by each login provider below.

Setup login providers required by your application

Use the following topics to configure your application to use the respective providers:

- Facebook instructions
- Twitter instructions
- Google instructions

- Microsoft instructions
- Other provider instructions

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods from AddAuthentication:

C#

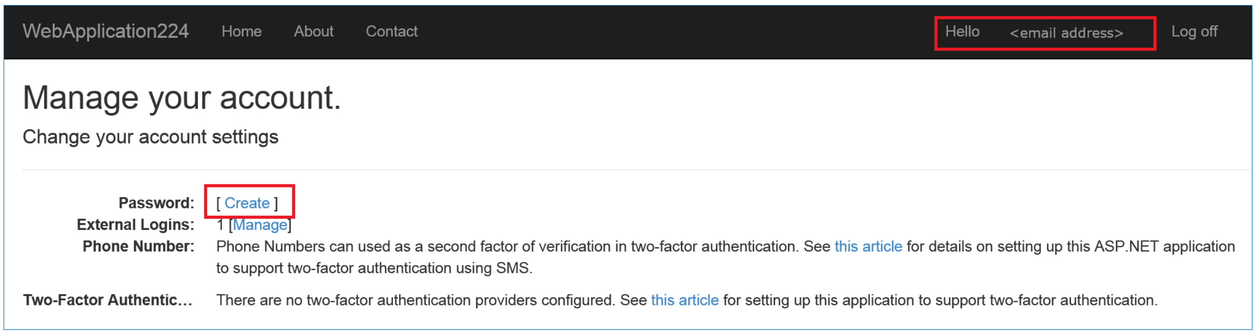
```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebApplication16.Data;

var builder = WebApplication.CreateBuilder(args);
var config = builder.Configuration;

var connectionString = config.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                 options.SignIn.RequireConfirmedAccount =
true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
builder.Services.AddAuthentication()
   .AddGoogle(options =>
   {
       IConfigurationSection googleAuthNSection =
       config.GetSection("Authentication:Google");
       options.ClientId = googleAuthNSection["ClientId"];
       options.ClientSecret = googleAuthNSection["ClientSecret"];
   })
   .AddFacebook(options =>
   {
       IConfigurationSection FBAuthNSection =
       config.GetSection("Authentication:FB");
       options.ClientId = FBAuthNSection["ClientId"];
       options.ClientSecret = FBAuthNSection["ClientSecret"];
   })
   .AddMicrosoftAccount(microsoftOptions =>
   {
       microsoftOptions.ClientId =
config["Authentication:Microsoft:ClientId"];
       microsoftOptions.ClientSecret =
config["Authentication:Microsoft:ClientSecret"];
   })
   .AddTwitter(twitterOptions =>
   {
       twitterOptions.ConsumerKey =
config["Authentication:Twitter:ConsumerAPIKey"];
       twitterOptions.ConsumerSecret =
config["Authentication:Twitter:ConsumerSecret"];
       twitterOptions.RetrieveUserDetails = true;
   });

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Optionally set password

When you register with an external login provider, you don't have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to sign in to the web site.

To create a password and sign in using your email that you set during the sign in process with external providers:

- Select the **Hello <email alias>** link at the top-right corner to navigate to the **Manage** view.

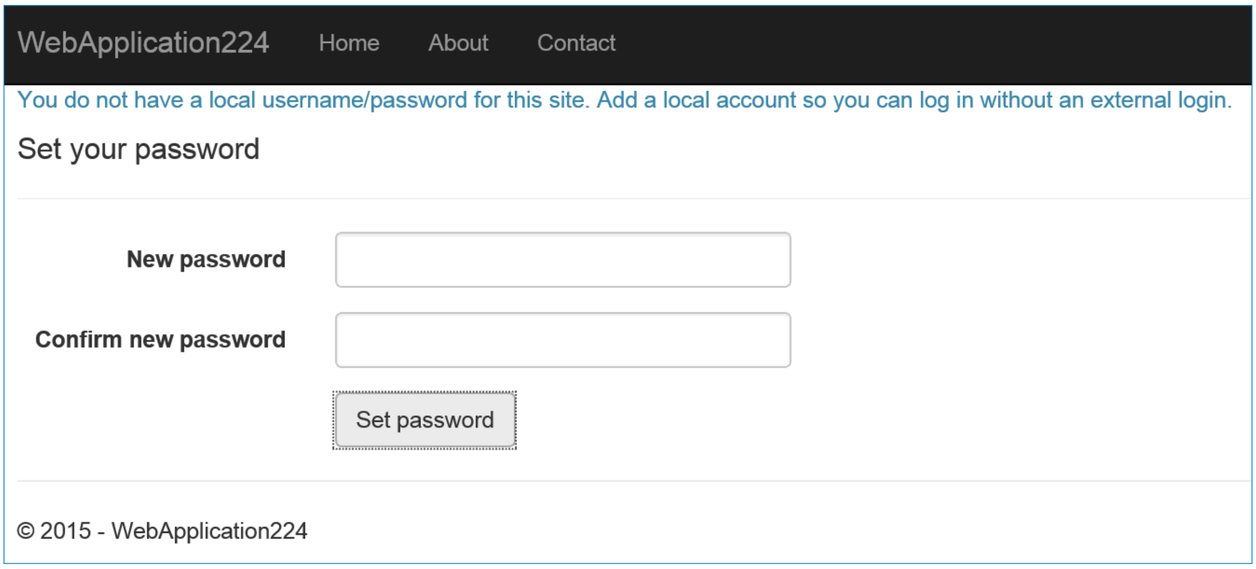


Manage your account.

Change your account settings

Password:	[Create]
External Logins:	1 [Manage]
Phone Number:	Phone Numbers can used as a second factor of verification in two-factor authentication. See this article for details on setting up this ASP.NET application to support two-factor authentication using SMS.
Two-Factor Authentic...	There are no two-factor authentication providers configured. See this article for setting up this application to support two-factor authentication.

- Select **Create**



You do not have a local username/password for this site. Add a local account so you can log in without an external login.

Set your password

New password	
Confirm new password	

© 2015 - WebApplication224

- Set a valid password and you can use this to sign in with your email.

Additional information

- Sign in with Apple Example Integration
- See this GitHub issue for information on how to customize the login buttons.

- Persist additional data about the user and their access and refresh tokens. For more information, see Persist additional claims and tokens from external providers in ASP.NET Core.

Google external login setup in ASP.NET Core

Article • 02/06/2024

By Valeriy Novytskyy and Rick Anderson

This tutorial shows you how to enable users to sign in with their Google account using the ASP.NET Core project created on the previous page.

Create the Google OAuth 2.0 Client ID and secret

- Follow the guidance in Integrating Google Sign-In into your web app (Google documentation).
- Go to Google API & Services.
- A **Project** must exist first, you may have to create one. Once a project is selected, enter the **Dashboard**.
- In the **Oauth consent screen** of the **Dashboard**:
    - Select **User Type - External** and **CREATE**.
    - In the **App information** dialog, Provide an **app name** for the app, **user support email**, and **developer contact information**.
    - Step through the **Scopes** step.
    - Step through the **Test users** step.
    - Review the **OAuth consent screen** and go back to the app **Dashboard**.
- In the **Credentials** tab of the application Dashboard, select **CREATE CREDENTIALS** > **OAuth client ID**.
- Select **Application type** > **Web application**, choose a **name**.
- In the **Authorized redirect URIs** section, select **ADD URI** to set the redirect URI. Example redirect URI: `https://localhost:{PORT}/signin-google`, where the `{PORT}` placeholder is the app's port.
- Select the **CREATE** button.
- Save the **Client ID** and **Client Secret** for use in the app's configuration.
- When deploying the site, either:

- Update the app's redirect URI in the **Google Console** to the app's deployed redirect URI.
- Create a new Google API registration in the **Google Console** for the production app with its production redirect URI.

Store the Google client ID and secret

Store sensitive settings such as the Google client ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.

2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret`:

.NET CLI

```
dotnet user-secrets set "Authentication:Google:ClientId" "<client-id>"
dotnet user-secrets set "Authentication:Google:ClientSecret" "<client-
secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

You can manage your API credentials and usage in the API Console.

Configure Google authentication

Add the Microsoft.AspNetCore.Authentication.Google NuGet package to the app.

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddGoogle(googleOptions =>
    {
        googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
        googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];
    });
```

The call to AddIdentity configures the default scheme settings. The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Google

- Run the app and select **Log in**. An option to sign in with Google appears.
- Select the **Google** button, which redirects to Google for authentication.
- After entering your Google credentials, you are redirected back to the web site.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```C#
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Google authentication, see the GoogleOptions API reference . This can be used to request different information about the user.

Change the default callback URI

The URI segment `/signin-google` is set as the default callback of the Google authentication provider. You can change the default callback URI while configuring the Google authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the GoogleOptions class.

Troubleshooting

- If the sign-in doesn't work and you aren't getting any errors, switch to development mode to make the issue easier to debug.
- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate results in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures Identity is configured.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Select **Apply Migrations** to create the database, and refresh the page to continue past the error.
- HTTP 500 error after successfully authenticating the request by the OAuth 2.0 provider such as Google: See this GitHub issue .
- How to implement external authentication with Google for React and other SPA apps: See this GitHub issue .

Next steps

- This article showed how you can authenticate with Google. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish the app to Azure, reset the `ClientSecret` in the Google API Console.
- Set the `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Facebook external login setup in ASP.NET Core

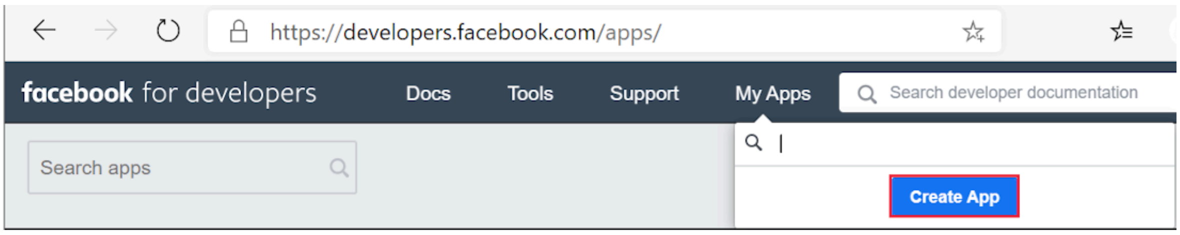
Article • 02/06/2024

By Valeriy Novytskyy and Rick Anderson

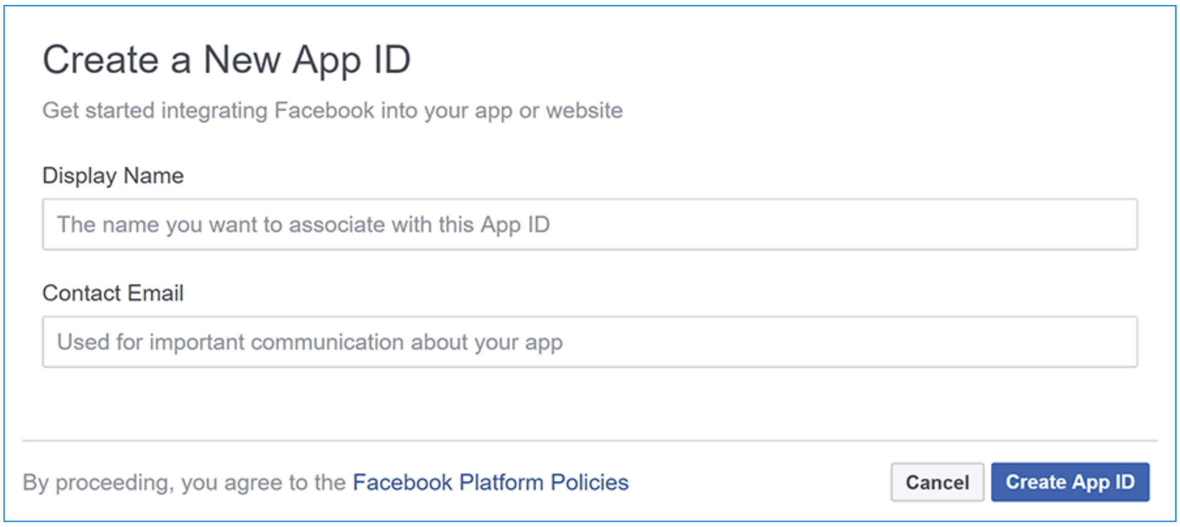
This tutorial with code examples shows how to enable your users to sign in with their Facebook account using a sample ASP.NET Core project created on the previous page. We start by creating a Facebook App ID by following the official steps.

Create the app in Facebook

- Add the Microsoft.AspNetCore.Authentication.Facebook NuGet package to the project.
- Navigate to the Facebook Developers app page and sign in. If you don't already have a Facebook account, use the **Sign up for Facebook** link on the login page to create one. Once you have a Facebook account, follow the instructions to register as a Facebook Developer.
- From the **My Apps** menu select **Create App**. The **Create an app** form appears.


https://developers.facebook.com/apps/
facebook for developers
Docs
Tools
Support
My Apps
Search developer documentation
Search apps
Create App

- Select an app type that best fits your project. For this project, select **Consumer**, and then **Next**. A new App ID is created.
- Fill out the form and tap the **Create App** button.



Create a New App ID

Get started integrating Facebook into your app or website

Display Name

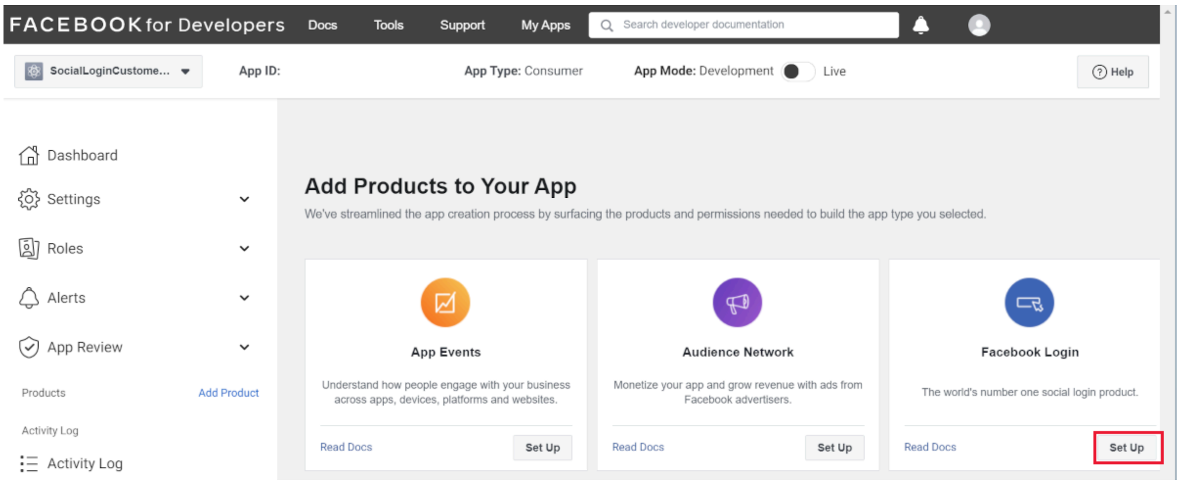
The name you want to associate with this App ID

Contact Email

Used for important communication about your app

By proceeding, you agree to the Facebook Platform Policies

- On the **Add Products to Your App** page, select **Set Up** on the **Facebook Login** card.



App ID:

App Type: Consumer

App Mode: Development Live

Add Products to Your App

We've streamlined the app creation process by surfacing the products and permissions needed to build the app type you selected.

App Events

Understand how people engage with your business across apps, devices, platforms and websites.

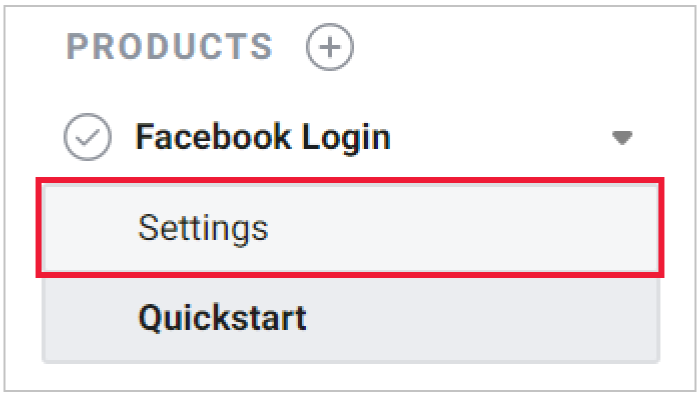
Audience Network

Monetize your app and grow revenue with ads from Facebook advertisers.

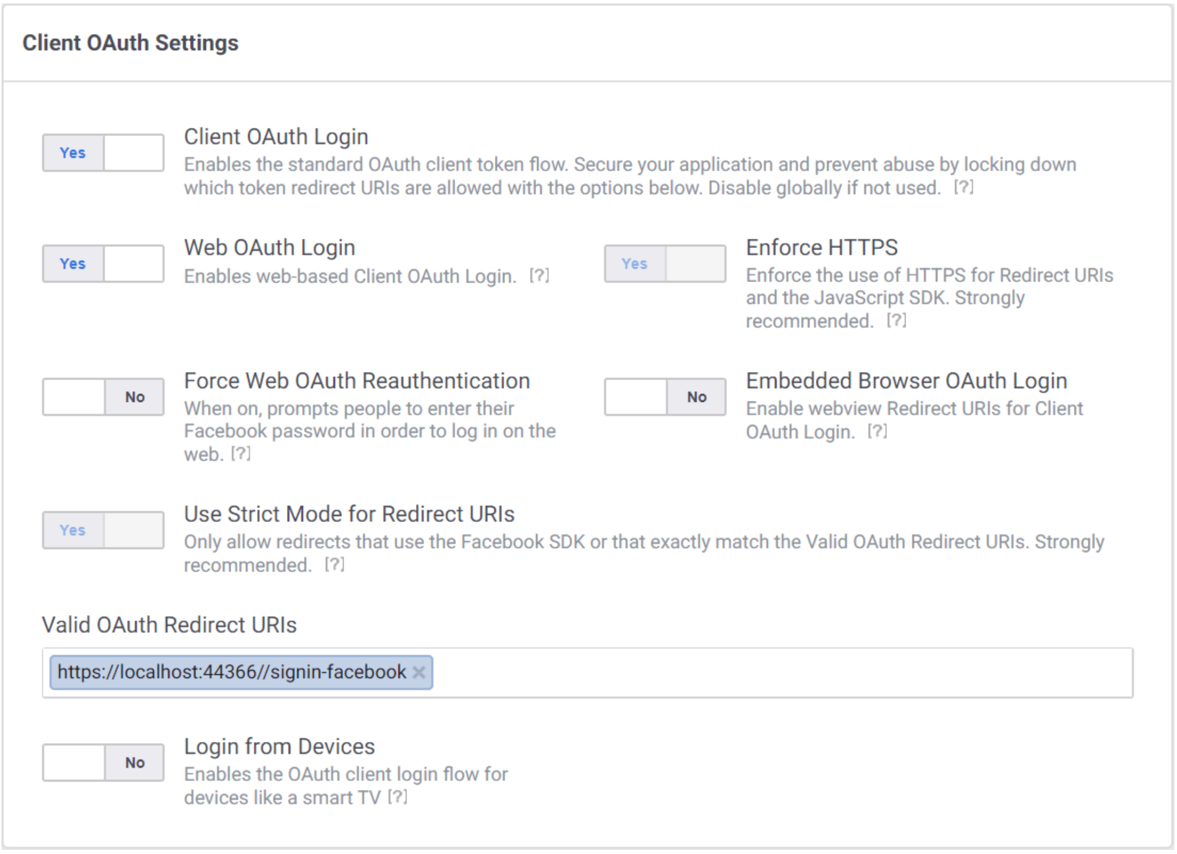
Facebook Login

The world's number one social login product.

- The **Quickstart** wizard launches with **Choose a Platform** as the first page. Bypass the wizard for now by clicking the **FaceBook Login Settings** link in the menu on the lower left:



- The **Client OAuth Settings** page is presented:


Client OAuth Settings

Yes — Client OAuth Login
Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]

Yes — Web OAuth Login
Enables web-based Client OAuth Login. [?]

Yes — Enforce HTTPS
Enforce the use of HTTPS for Redirect URIs and the JavaScript SDK. Strongly recommended. [?]

No — Force Web OAuth Reauthentication
When on, prompts people to enter their Facebook password in order to log in on the web. [?]

No — Embedded Browser OAuth Login
Enable webview Redirect URIs for Client OAuth Login. [?]

Yes — Use Strict Mode for Redirect URIs
Only allow redirects that use the Facebook SDK or that exactly match the Valid OAuth Redirect URIs. Strongly recommended. [?]

Valid OAuth Redirect URIs
https://localhost:44366//signin-facebook

No — Login from Devices
Enables the OAuth client login flow for devices like a smart TV [?]

- Enter your development URI with */signin-facebook* appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-facebook`). The Facebook authentication configured later in this tutorial will automatically handle requests at */signin-facebook* route to implement the OAuth flow.

ⓘ Note

The URI */signin-facebook* is set as the default callback of the Facebook authentication provider. You can change the default callback URI while configuring the Facebook authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the FacebookOptions class.

- Select **Save Changes**.
- Select **Settings** > **Basic** link in the left navigation.
- Make a note of your `App ID` and your `App Secret`. You will add both into your ASP.NET Core application in the next section:
- When deploying the site you need to revisit the **Facebook Login** setup page, and register a new public URI.

Store the Facebook app ID and secret

Store sensitive settings such as the Facebook app ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.

2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret`:

   .NET CLI

   ```
   dotnet user-secrets set "Authentication:Facebook:AppId" "<app-id>"
   dotnet user-secrets set "Authentication:Facebook:AppSecret" "<app-secret>"
   ```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

Configure Facebook Authentication

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddFacebook(facebookOptions =>
    {
        facebookOptions.AppId =
configuration["Authentication:Facebook:AppId"];
        facebookOptions.AppSecret =
configuration["Authentication:Facebook:AppSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Facebook

- Run the app and select **Log in**.
- Under **Use another service to log in.**, select Facebook.
- You are redirected to **Facebook** for authentication.
- Enter your Facebook credentials.
- You are redirected back to your site where you can set your email.

You are now logged in using your Facebook credentials:

React to cancel authorize external sign-in

AccessDeniedPath can provide a redirect path to the user agent when the user doesn't approve the requested authorization demand.

The following code sets the `AccessDeniedPath` to `"/AccessDeniedPathInfo"`:

C#

```
services.AddAuthentication().AddFacebook(options =>
{
    options.AppId = Configuration["Authentication:Facebook:AppId"];
    options.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
    options.AccessDeniedPath = "/AccessDeniedPathInfo";
});
```

We recommend the `AccessDeniedPath` page contains the following information:

- Remote authentication was canceled.
- This app requires authentication.
- To try sign-in again, select the Login link.

Test AccessDeniedPath

- Navigate to facebook.com
- If you are signed in, you must sign out.

- Run the app and select Facebook sign-in.
- Select **Not now**. You are redirected to the specified `AccessDeniedPath` page.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```C#
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Facebook authentication, see the FacebookOptions API reference. Configuration options can be used to:

- Request different information about the user.
- Add query string arguments to customize the login experience.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with other providers listed on the previous page.

- Once you publish your web site to Azure web app, you should reset the `AppSecret` in the Facebook developer portal.

- Set the `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Microsoft Account external login setup with ASP.NET Core

Article • 06/03/2022

By Valeriy Novytskyy and Rick Anderson

This sample shows you how to enable users to sign in with their work, school, or personal Microsoft account using the ASP.NET Core 6.0 project created on the previous page.

Create the app in Microsoft Developer Portal

- Add the Microsoft.AspNetCore.Authentication.MicrosoftAccount NuGet package to the project.
- Navigate to the Azure portal - App registrations page and create or sign into a Microsoft account:

If you don't have a Microsoft account, select **Create one**. After signing in, you are redirected to the **App registrations** page:

- Select **New registration**
- Enter a **Name**.
- Select an option for **Supported account types**.
  - The `MicrosoftAccount` package supports App Registrations created using "Accounts in any organizational directory" or "Accounts in any organizational directory and Microsoft accounts" options by default.
  - To use other options, set `AuthorizationEndpoint` and `TokenEndpoint` members of `MicrosoftAccountOptions` used to initialize the Microsoft Account authentication to the URLs displayed on **Endpoints** page of the App Registration after it is created (available by clicking Endpoints on the **Overview** page).
- Under **Redirect URI**, enter your development URL with `/signin-microsoft` appended. For example, `https://localhost:5001/signin-microsoft`. The Microsoft authentication scheme configured later in this sample will automatically handle requests at `/signin-microsoft` route to implement the OAuth flow.
- Select **Register**

Create client secret

- In the left pane, select **Certificates & secrets**.
- Under **Client secrets**, select **New client secret**
  - Add a description for the client secret.
  - Select the **Add** button.
- Under **Client secrets**, copy the value of the client secret.

The URI segment `/signin-microsoft` is set as the default callback of the Microsoft authentication provider. You can change the default callback URI while configuring the Microsoft authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the MicrosoftAccountOptions class.

Store the Microsoft client ID and secret

Store sensitive settings such as the Microsoft **Application (client) ID** found on the **Overview** page of the App Registration and **Client Secret** you created on the **Certificates & secrets page** with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.

2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret`:

   .NET CLI

   ```
   dotnet user-secrets set "Authentication:Microsoft:ClientId" "<client-id>"
   dotnet user-secrets set "Authentication:Microsoft:ClientSecret" "<client-secret>"
   ```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

Configure Microsoft Account Authentication

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId = 
configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret = 
configuration["Authentication:Microsoft:ClientSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

For more information about configuration options supported by Microsoft Account authentication, see the MicrosoftAccountOptions API reference. This can be used to request different information about the user.

Sign in with Microsoft Account

- Run the app and select **Log in**. An option to sign in with Microsoft appears.
- Select to sign in with Microsoft. You are redirected to Microsoft for authentication. After signing in with your Microsoft Account, you will be prompted to let the app access your info:
- Select **Yes**. You are redirected back to the web site where you can set your email.

You are now logged in using your Microsoft credentials.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- If the Microsoft Account provider redirects you to a sign in error page, note the error title and description query string parameters directly following the `#` (hashtag) in the Uri.

  Although the error message seems to indicate a problem with Microsoft authentication, the most common cause is your application Uri not matching any of the **Redirect URIs** specified for the **Web** platform.

- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this sample ensures that this is done.

- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Microsoft. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish your web site to Azure web app, create a new client secrets in the Microsoft Developer Portal.
- Set the `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Twitter external sign-in setup with ASP.NET Core

Article • 09/05/2024

By Valeriy Novytskyy and Rick Anderson

This sample shows how to enable users to sign in with their Twitter account using a sample ASP.NET Core project created on the previous page.

> **Note**
>
> The Microsoft.AspNetCore.Authentication.Twitter package described below uses the OAuth 1.0 APIs provided by Twitter. Twitter has since added OAuth 2.0 APIs with a different set of functionality. The **OpenIddict** and **AspNet.Security.OAuth.Twitter** packages are community implementations that use the new OAuth 2.0 APIs.

Create the app in Twitter

- Add the Microsoft.AspNetCore.Authentication.Twitter NuGet package to the project.
- Navigate to twitter developer portal Dashboard and sign in. If you don't already have a Twitter account, use the **Sign up now** link to create one.
- If you don't have a project, create one.
- Select **+ Add app**. Fill out the **App name** then record the generated API Key, API Key Secret and Bearer Token. These will be needed later.
- In the **App Settings** page, select **Edit** in the **Authentication settings** section, then:
  - Enable 3-legged OAuth
  - Request email address from users
  - Fill out the required fields and select **Save**

  > **Note**

Microsoft.AspNetCore.Identity requires users to have an email address by default. For Callback URLs during development, use `https://localhost:{PORT}/signin-twitter`, where the `{PORT}` placeholder is the app's port.

ⓘ **Note**

The URI segment `/signin-twitter` is set as the default callback of the Twitter authentication provider. You can change the default callback URI while configuring the Twitter authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the TwitterOptions class.

Store the Twitter consumer API key and secret

Store sensitive settings such as the Twitter consumer API key and secret with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.

2. Store the sensitive settings in the local secret store with the secrets keys `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret`:

   .NET CLI

   ```
   dotnet user-secrets set "Authentication:Twitter:ConsumerAPIKey" "
   <consumer-api-key>"
   dotnet user-secrets set "Authentication:Twitter:ConsumerSecret" "
   <consumer-secret>"
   ```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

These tokens can be found on the **Keys and Access Tokens** tab after creating a new Twitter application:

Configure Twitter Authentication

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddTwitter(twitterOptions =>
    {
        twitterOptions.ConsumerKey =
configuration["Authentication:Twitter:ConsumerAPIKey"];
        twitterOptions.ConsumerSecret =
configuration["Authentication:Twitter:ConsumerSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Twitter authentication, see the TwitterOptions API reference. This can be used to request different information about the user.

Sign in with Twitter

Run the app and select **Log in**. An option to sign in with Twitter appears:

Selecting **Twitter** redirects to Twitter for authentication:

After entering your Twitter credentials, you are redirected back to the web site where you can set your email.

You are now logged in using your Twitter credentials:

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this sample ensures Identity is configured.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Twitter. You can follow a similar approach to authenticate with other providers listed on the previous page.

- Once you publish your web site to Azure web app, you should reset the `ConsumerSecret` in the Twitter developer portal.
- Set the `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

External OAuth authentication providers

Article • 09/05/2024

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

The following list includes common external OAuth authentication providers that work with ASP.NET Core apps. Third-party NuGet packages, such as the ones maintained by OpenIddict or aspnet-contrib, can be used to complement the authentication providers implemented by the ASP.NET Core team.

- LinkedIn
- Instagram
- Reddit (Instructions)
- Github (Instructions)
- Yahoo (Instructions)
- Tumblr (Instructions)
- Pinterest (Instructions)
- Pocket (Instructions)
- Flickr (Instructions)
- Dribbble (Instructions)
- Vimeo (Instructions)
- SoundCloud (Instructions)
- VK (Instructions)

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Persist additional claims and tokens from external providers in ASP.NET Core

Article • 09/07/2023

An ASP.NET Core app can establish additional claims and tokens from external authentication providers, such as Facebook, Google, Microsoft, and Twitter. Each provider reveals different information about users on its platform, but the pattern for receiving and transforming user data into additional claims is the same.

Prerequisites

Decide which external authentication providers to support in the app. For each provider, register the app and obtain a client ID and client secret. For more information, see Facebook and Google authentication in ASP.NET Core. The sample app uses the Google authentication provider.

Set the client ID and client secret

The OAuth authentication provider establishes a trust relationship with an app using a client ID and client secret. Client ID and client secret values are created for the app by the external authentication provider when the app is registered with the provider. Each external provider that the app uses must be configured independently with the provider's client ID and client secret. For more information, see the external authentication provider topics that apply:

- Facebook authentication
- Google authentication
- Microsoft authentication
- Twitter authentication
- Other authentication providers
- OpenIdConnect

Optional claims sent in the ID or access token from the authentication provider are usually configured in the provider's online portal. For example, Microsoft Entra ID permits assigning optional claims to the app's ID token in the app registration's **Token configuration** blade. For more information, see How to: Provide optional claims to your app (Azure documentation). For other providers, consult their external documentation sets.

The sample app configures the Google authentication provider with a client ID and client secret provided by Google:

```C#
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebGoogOauth.Data;

var builder = WebApplication.CreateBuilder(args);
var configuration = builder.Configuration;

builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                  options.SignIn.RequireConfirmedAccount =
true)
```

```
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

// Remaining code removed for brevity.
```

Establish the authentication scope

Specify the list of permissions to retrieve from the provider by specifying the Scope. Authentication scopes for common external providers appear in the following table.

Expand table

Provider	Scope
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`profile`, `email`, `openid`
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

In the sample app, Google's `profile`, `email`, and `openid` scopes are automatically added by the framework when AddGoogle is called on the AuthenticationBuilder. If the app requires additional scopes, add them to the options. In the following example, the Google `https://www.googleapis.com/auth/user.birthday.read` scope is added to retrieve a user's birthday:

C#

```
options.Scope.Add("https://www.googleapis.com/auth/user.birthday.read");
```

Map user data keys and create claims

In the provider's options, specify a MapJsonKey or MapJsonSubKey for each key or subkey in the external provider's JSON user data for the app identity to read on sign in. For more information on claim types, see ClaimTypes.

The sample app creates locale (`urn:google:locale`) and picture (`urn:google:picture`) claims from the `locale` and `picture` keys in Google user data:

C#

```
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

In `Microsoft.AspNetCore.Identity.UI.Pages.Account.Internal.ExternalLoginModel.OnPostConfirmationAsync`, an IdentityUser (`ApplicationUser`) is signed into the app with SignInAsync. During the sign in process, the UserManager<TUser> can store an `ApplicationUser` claims for user data available from the Principal.

In the sample app, `OnPostConfirmationAsync` (`Account/ExternalLogin.cshtml.cs`) establishes the locale (`urn:google:locale`) and picture (`urn:google:picture`) claims for the signed in `ApplicationUser`, including a claim for GivenName:

C#

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl =
null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information during 
confirmation.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email, 
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email, 
CancellationToken.None);

        var result = await _userManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);
            if (result.Succeeded)
            {
                _logger.LogInformation("User created an account using {Name} 
provider.", info.LoginProvider);

                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type == 
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type == 
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type == 
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                // using Microsoft.AspNetCore.Authentication;
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = false;

                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                // If account confirmation is required, we need to show the
link if we don't have a real email sender
                if (_userManager.Options.SignIn.RequireConfirmedAccount)
                {
                    return RedirectToPage("./RegisterConfirmation", new {
Email = Input.Email });
                }

                await _signInManager.SignInAsync(user, props,
info.LoginProvider);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    ProviderDisplayName = info.ProviderDisplayName;
    ReturnUrl = returnUrl;
    return Page();
}
```

By default, a user's claims are stored in the authentication cookie. If the authentication cookie is too large, it can cause the app to fail because:

- The browser detects that the cookie header is too long.
- The overall size of the request is too large.

If a large amount of user data is required for processing user requests:

- Limit the number and size of user claims for request processing to only what the app requires.
- Use a custom ITicketStore for the Cookie Authentication Middleware's SessionStore to store identity across requests. Preserve large quantities of identity information on the server while only sending a small session identifier key to the client.

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

The sample app sets the value of `SaveTokens` to `true` in GoogleOptions:

C#

```
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

When `OnPostConfirmationAsync` executes, store the access token (ExternalLoginInfo.AuthenticationTokens) from the external provider in the `ApplicationUser`'s `AuthenticationProperties`.

The sample app saves the access token in `OnPostConfirmationAsync` (new user registration) and `OnGetCallbackAsync` (previously registered user) in `Account/ExternalLogin.cshtml.cs`:

C#

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl =
null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information during
confirmation.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);

        var result = await _userManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);
            if (result.Succeeded)
            {
                _logger.LogInformation("User created an account using {Name}
provider.", info.LoginProvider);

                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type ==
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }
                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                // using Microsoft.AspNetCore.Authentication;
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = false;

                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                // If account confirmation is required, we need to show the
link if we don't have a real email sender
                if (_userManager.Options.SignIn.RequireConfirmedAccount)
                {
                    return RedirectToPage("./RegisterConfirmation", new {
Email = Input.Email });
                }

                await _signInManager.SignInAsync(user, props,
info.LoginProvider);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }
    ProviderDisplayName = info.ProviderDisplayName;
    ReturnUrl = returnUrl;
    return Page();
}
```

ⓘ **Note**

For information on passing tokens to the Razor components of a server-side Blazor app, see Server-side ASP.NET Core Blazor additional security scenarios.

How to add additional custom tokens

To demonstrate how to add a custom token, which is stored as part of `SaveTokens`, the sample app adds an AuthenticationToken with the current DateTime for an AuthenticationToken.Name of `TicketCreated`:

C#

```csharp
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;
    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Create and add claims

The framework provides common actions and extension methods for creating and adding claims to the collection. For more information, see the ClaimActionCollectionMapExtensions and ClaimActionCollectionUniqueExtensions.

Users can define custom actions by deriving from ClaimAction and implementing the abstract Run method.

For more information, see Microsoft.AspNetCore.Authentication.OAuth.Claims.

Add and update user claims

Claims are copied from external providers to the user database on first registration, not on sign in. If additional claims are enabled in an app after a user registers to use the app, call SignInManager.RefreshSignInAsync on a user to force the generation of a new authentication cookie.

In the Development environment working with test user accounts, delete and recreate the user account. For production systems, new claims added to the app can be backfilled into user accounts. After scaffolding the ExternalLogin page into the app at `Areas/Pages/Identity/Account/Manage`, add the following code to the `ExternalLoginModel` in the `ExternalLogin.cshtml.cs` file.

Add a dictionary of added claims. Use the dictionary keys to hold the claim types, and use the values to hold a default value. Add the following line to the top of the class. The following example assumes that one claim is added for the user's Google picture with a generic headshot image as the default value:

C#

```csharp
private readonly IReadOnlyDictionary<string, string> _claimsToSync =
     new Dictionary<string, string>()
     {
             { "urn:google:picture", "https://localhost:5001/headshot.png"
},
     };
```

Replace the default code of the `OnGetCallbackAsync` method with the following code. The code loops through the claims dictionary. Claims are added (backfilled) or updated for each user. When claims are added or updated, the user sign-in is refreshed using the SignInManager<TUser>, preserving the existing authentication properties (`AuthenticationProperties`).

C#

```csharp
private readonly IReadOnlyDictionary<string, string> _claimsToSync =
     new Dictionary<string, string>()
     {
             { "urn:google:picture", "https://localhost:5001/headshot.png"
},
     };

public async Task<IActionResult> OnGetCallbackAsync(string returnUrl = null,
string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (remoteError != null)
    {
        ErrorMessage = $"Error from external provider: {remoteError}";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    // Sign in the user with this external login provider if the user
already has a login.
    var result = await
_signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false, bypassTwoFactor: true);
    if (result.Succeeded)
    {
        _logger.LogInformation("{Name} logged in with {LoginProvider}
provider.", info.Principal.Identity.Name, info.LoginProvider);
```

```
        if (_claimsToSync.Count > 0)
        {
            var user = await
_userManager.FindByLoginAsync(info.LoginProvider,
                info.ProviderKey);
            var userClaims = await _userManager.GetClaimsAsync(user);
            bool refreshSignIn = false;

            foreach (var addedClaim in _claimsToSync)
            {
                var userClaim = userClaims
                    .FirstOrDefault(c => c.Type == addedClaim.Key);

                if (info.Principal.HasClaim(c => c.Type == addedClaim.Key))
                {
                    var externalClaim =
info.Principal.FindFirst(addedClaim.Key);

                    if (userClaim == null)
                    {
                        await _userManager.AddClaimAsync(user,
                            new Claim(addedClaim.Key, externalClaim.Value));
                        refreshSignIn = true;
                    }
                    else if (userClaim.Value != externalClaim.Value)
                    {
                        await _userManager
                            .ReplaceClaimAsync(user, userClaim,
externalClaim);
                        refreshSignIn = true;
                    }
                }
                else if (userClaim == null)
                {
                    // Fill with a default value
                    await _userManager.AddClaimAsync(user, new
Claim(addedClaim.Key,
                        addedClaim.Value));
                    refreshSignIn = true;
                }
            }

            if (refreshSignIn)
            {
                await _signInManager.RefreshSignInAsync(user);
            }
        }

        return LocalRedirect(returnUrl);
    }
    if (result.IsLockedOut)
    {
        return RedirectToPage("./Lockout");
    }
    else
    {
        // If the user does not have an account, then ask the user to create
an account.
        ReturnUrl = returnUrl;
        ProviderDisplayName = info.ProviderDisplayName;
        if (info.Principal.HasClaim(c => c.Type == ClaimTypes.Email))
        {
            Input = new InputModel
            {
                Email = info.Principal.FindFirstValue(ClaimTypes.Email)
            };
        }
        return Page();
    }
}
```

A similar approach is taken when claims change while a user is signed in but a backfill step isn't required. To update a user's claims, call the following on the user:

- UserManager.ReplaceClaimAsync on the user for claims stored in the identity database.
- SignInManager.RefreshSignInAsync on the user to force the generation of a new authentication cookie.

Remove claim actions and claims

ClaimActionCollection.Remove(String) removes all claim actions for the given ClaimType from the collection.
ClaimActionCollectionMapExtensions.DeleteClaim(ClaimActionCollection, String) deletes a claim of the given ClaimType from the identity. DeleteClaim is primarily used with OpenID Connect (OIDC) to remove protocol-generated claims.

Sample app output

Run the sample app and select the **MyClaims** link:

```text
User Claims

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    9b342344f-7aab-43c2-1ac1-ba75912ca999
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    someone@gmail.com
AspNet.Identity.SecurityStamp
    7D4312MOWRYYBFI1KXRPHGOSTBVWSFDE
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname
```

```
    Judy
urn:google:locale
    en
urn:google:picture
    https://lh4.googleusercontent.com/-XXXXXX/XXXXXX/XXXXXX/XXXXXX/photo.jpg

Authentication Properties

.Token.access_token
    yc23.AlvoZqz56...1lxltXV7D-ZWP9
.Token.token_type
    Bearer
.Token.expires_at
    2019-04-11T22:14:51.0000000+00:00
.Token.TicketCreated
    4/11/2019 9:14:52 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.persistent
.issued
    Thu, 11 Apr 2019 20:51:06 GMT
.expires
    Thu, 25 Apr 2019 20:51:06 GMT
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

View or download sample code (how to download)

Additional resources

- dotnet/AspNetCore engineering SocialSample app: The linked sample app is on the dotnet/AspNetCore GitHub repo's `main` engineering branch. The `main` branch contains code under active development for the next release of ASP.NET Core. To see a version of the sample app for a released version of ASP.NET Core, use the **Branch** drop down list to select a release branch (for example `release/{X.Y}`).

Articles based on ASP.NET Core projects created with individual user accounts

Article • 05/31/2024

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

The authentication templates are available in .NET CLI with `-au Individual`:

```
dotnet new mvc -au Individual
dotnet new webapp -au Individual
```

See this GitHub issue for web API authentication.

No Authentication

Authentication is specified in the .NET CLI with the `-au` option. In Visual Studio, the **Change Authentication** dialog is available for new web applications. The default for new web apps in Visual Studio is **No Authentication**.

Projects created with no authentication:

- Don't contain web pages and UI to sign in and sign out.
- Don't contain authentication code.

Windows Authentication

Windows Authentication is specified for new web apps in the .NET CLI with the `-au Windows` option. In Visual Studio, the **Change Authentication** dialog provides the **Windows Authentication** options.

If Windows Authentication is selected, the app is configured to use the Windows Authentication IIS module. Windows Authentication is intended for Intranet web sites.

dotnet new webapp authentication options

The following table shows the authentication options available for new web apps:

Expand table

Option	Type of authentication	Link for more information
None	No authentication.	
Individual	Individual authentication.	Introduction to Identity on ASP.NET Core
IndividualB2C	Cloud-hosted individual authentication with Azure AD B2C.	Azure AD B2C
SingleOrg	Organizational authentication for a single tenant. Entra External ID tenants also use SingleOrg.	Entra ID
MultiOrg	Organizational authentication for multiple tenants.	Entra ID
Windows	Windows authentication.	Windows Authentication

Visual Studio new webapp authentication options

The following table shows the authentication options available when creating a new web app with Visual Studio:

Expand table

Option	Type of authentication	Link for more information
None	No authentication	
Individual User Accounts / Store user accounts in-app	Individual authentication	Introduction to Identity on ASP.NET Core
Individual User Accounts / Connect to an existing user store in the cloud	Cloud-hosted individual authentication with Azure AD B2C	Azure AD B2C
Work or School Cloud / Single Org	Organizational authentication for a single tenant	Azure AD
Work or School Cloud / Multiple Org	Organizational authentication for multiple tenants	Azure AD
Windows	Windows authentication	Windows Authentication

Additional resources

The following articles show how to use the code generated in ASP.NET Core templates that use individual user accounts:

- Account confirmation and password recovery in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization

Microsoft Entra ID with ASP.NET Core

Article • 02/06/2024

These tutorials and samples demonstrate authentication in ASP.NET Core using Microsoft identity platform and Microsoft Entra ID. For additional tutorials and samples using ASP.NET Core with Azure AD, see Microsoft identity platform.

Application Scenarios

- Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app
- Web app that signs in users
- Web app that calls web APIs
- Protected web API
- Web API that calls other web APIs
- Web app that signs in users with Azure AD B2C

Samples

- Enable your ASP.NET Core app to sign-in users and call web APIs using Microsoft identity platform for developers

Microsoft identity platform documentation

Use the Microsoft identity platform and our open-source authentication libraries to sign in users with Microsoft Entra accounts, Microsoft personal accounts, and social accounts like Facebook and Google. Protect your web APIs and access protected APIs like Microsoft Graph to work with your users' and organization's data.

OVERVIEW
What is the Microsoft identity platform?

CONCEPT
Authentication and authorization basics

CONCEPT
App types and authentication flows

SAMPLE
Code samples

WHAT'S NEW
What's new in docs

CONCEPT
OAuth 2.0 and OpenID Connect (OIDC)

CONCEPT
Migrate apps to MSAL

QUICKSTART
Register an application

Authentication and authorization for any app

Focus on documentation for the app you're building or extending with identity and access management (IAM) support by selecting its type.

Single-page app (SPA)

Web app

Web API

A web app whose code is downloaded and run in the browser itself.

A traditional, or "classic," web app whose code runs on a server, returning page data for the browser to render.

A RESTful web service accessed by apps or other web APIs, typically to work with data served by the API.

Desktop app

Apps with a user interface (UI) whose code runs on a user's desktop, laptop, or notebook computer.

Mobile app

Apps with a UI whose code runs on a user's phone, tablet, or other mobile device.

Background service, daemon, or script

An app or script without a UI that performs non-interactive tasks like server-to-server communication or scheduled...

Get started

Quick access to guidance on adding core IAM features to your apps and best practices for keeping your apps secure and available.

Sign in users

- Single-page web app (SPA)
- Web app
- Desktop app
- Mobile app

Protect a web API

- Configure an app to expose a web API
- Build a protected web API
- Call an API from a non-interactive app or script

Test and deploy apps

- Build a test environment
- Run integration tests

Build for security and resilience

- Build Zero Trust-ready apps
- Prevent an over-privileged app
- Build auth resilience into your apps

Microsoft authentication libraries

The open-source Microsoft Authentication Library (MSAL) is built and supported by Microsoft. We recommend MSAL for any app that uses the Microsoft identity platform for authentication and authorization.

.NET

Android

Angular

iOS & macOS

Java

JavaScript

Node.js

Python

React

Authenticate partners and customers

Sign in users from partner organizations in a business-to-business (B2B) scenario or create custom sign-up and sign-in experiences for your customers in a business-to-customer (B2C)...

External Identities documentation

Connect to Microsoft Graph

Programmatic access to organizational, user, and app data stored in Microsoft Entra ID. Call Microsoft Graph from your app to create and manage Microsoft Entra users and groups, get an...

Microsoft Graph API documentation

Manage and market your apps

Make existing SaaS apps like Dropbox, Salesforce, and ServiceNow available to your organization's users, configure SSO, and manage security. Or, become an independent software vendor (ISV) by...

App management documentation

Manage app users and their access

Automatically create user identities and their roles in your organization's installed SaaS apps. HR-driven provisioning, System for Cross-domain Identity Management (SCIM), and more.

App user and role provisioning documentation

Authentication and authorization in Azure App Service and Azure Functions

Article • 09/27/2024

Note

Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention `<app-name>-<random-hash>.<region>.azurewebsites.net`. Existing app names will remain unchanged.

Example: `myapp-ds27dh7271aah175.westus-01.azurewebsites.net`

For further details, refer to Unique Default Hostname for App Service Resource.

Azure App Service provides built-in authentication and authorization capabilities (sometimes referred to as "Easy Auth"), so you can sign in users and access data by writing minimal or no code in your web app, RESTful API, and mobile back end, and also Azure Functions. This article describes how App Service helps simplify authentication and authorization for your app.

Why use the built-in authentication?

You're not required to use this feature for authentication and authorization. You can use the bundled security features in your web framework of choice, or you can write your own utilities. However, you will need to ensure that your solution stays up to date with the latest security, protocol, and browser updates.

Implementing a secure solution for authentication (signing-in users) and authorization (providing access to secure data) can take significant effort. You must make sure to follow industry best practices and standards, and keep your implementation up to date. The built-in authentication feature for App Service and Azure Functions can save you time and effort by providing out-of-the-box authentication with federated identity providers, allowing you to focus on the rest of your application.

- Azure App Service allows you to integrate a variety of auth capabilities into your web app or API without implementing them yourself.
- It’s built directly into the platform and doesn’t require any particular language, SDK, security expertise, or even any code to utilize.

- You can integrate with multiple login providers. For example, Microsoft Entra, Facebook, Google, X.

Your app might need to support more complex scenarios such as Visual Studio integration or incremental consent. There are several different authentication solutions available to support these scenarios. To learn more, read Identity scenarios.

Identity providers

App Service uses federated identity, in which a third-party identity provider manages the user identities and authentication flow for you. The following identity providers are available by default:

Provider	Sign-in endpoint	How-To guidance
Microsoft Entra	`/.auth/login/aad`	App Service Microsoft Entra platform login
Facebook	`/.auth/login/facebook`	App Service Facebook login
Google	`/.auth/login/google`	App Service Google login
X	`/.auth/login/x`	App Service X login
GitHub	`/.auth/login/github`	App Service GitHub login
Sign in with Apple	`/.auth/login/apple`	App Service Sign in With Apple login (Preview)
Any OpenID Connect provider	`/.auth/login/<providerName>`	App Service OpenID Connect login

When you configure this feature with one of these providers, its sign-in endpoint is available for user authentication and for validation of authentication tokens from the provider. You can provide your users with any number of these sign-in options.

Considerations for using built-in authentication

Enabling this feature will cause all requests to your application to be automatically redirected to HTTPS, regardless of the App Service configuration setting to enforce HTTPS. You can disable this with the `requireHttps` setting in the V2 configuration. However, we do recommend sticking with HTTPS, and you should ensure no security tokens ever get transmitted over non-secure HTTP connections.

App Service can be used for authentication with or without restricting access to your site content and APIs. Access restrictions can be set in the **Authentication** > **Authentication settings** section of your web app. To restrict app access only to authenticated users, set **Action to take when request is not authenticated** to log in with one of the configured identity providers. To authenticate but not restrict access, set **Action to take when request is not authenticated** to "Allow anonymous requests (no action)."

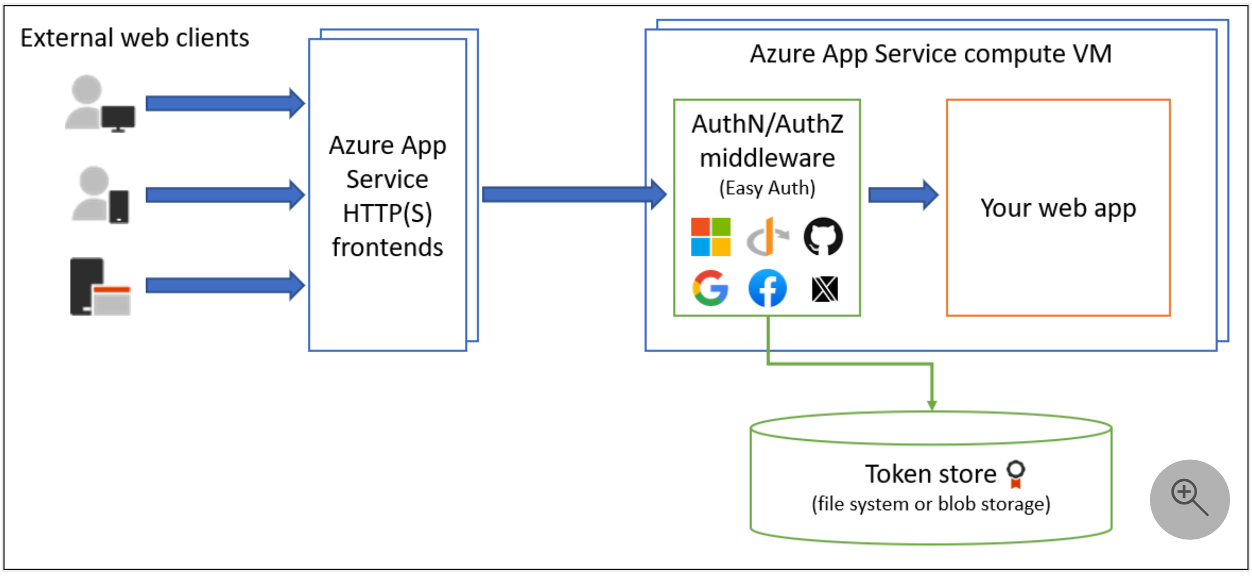
ⓘ **Important**

You should give each app registration its own permission and consent. Avoid permission sharing between environments by using separate app registrations for separate deployment slots. When testing new code, this practice can help prevent issues from affecting the production app.

How it works

Feature architecture

The authentication and authorization middleware component is a feature of the platform that runs on the same VM as your application. When it's enabled, every incoming HTTP request passes through it before being handled by your application.


External web clients
Azure App Service HTTP(S) frontends
Azure App Service compute VM
AuthN/AuthZ middleware
(Easy Auth)
Your web app
Token store
(file system or blob storage)

The platform middleware handles several things for your app:

- Authenticates users and clients with the specified identity provider(s)
- Validates, stores, and refreshes OAuth tokens issued by the configured identity provider(s)
- Manages the authenticated session
- Injects identity information into HTTP request headers

The module runs separately from your application code and can be configured using Azure Resource Manager settings or using a configuration file. No SDKs, specific programming languages, or changes to your application code are required.

Feature architecture on Windows (non-container deployment)

The authentication and authorization module runs as a native IIS module in the same sandbox as your application. When it's enabled, every incoming HTTP request passes through it before being handled by your application.

Feature architecture on Linux and containers

The authentication and authorization module runs in a separate container, isolated from your application code. Using what's known as the Ambassador pattern, it interacts with the incoming traffic to perform similar functionality as on Windows. Because it does not run in-process, no direct integration with specific language frameworks is possible; however, the relevant information that your app needs is passed through using request headers as explained below.

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- Without provider SDK: The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*. This case applies to browser apps and mobile apps that use an embedded browser for authentication.
- With provider SDK: The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This case applies to REST APIs, Azure Functions, and JavaScript browser clients, as well as browser apps that need more flexibility in the sign-in process. It also applies to native mobile apps that sign users in using the provider's SDK.

Calls from a trusted browser app in App Service to another REST API in App Service or Azure Functions can be authenticated using the server-directed flow. For more information, see Customize sign-ins and sign-outs.

The table below shows the steps of the authentication flow.

Expand table

Step	Without provider SDK	With provider SDK
1. Sign user in	Redirects client to `/.auth/login/<provider>`.	Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation.
2. Post-authentication	Provider redirects client to `/.auth/login/<provider>/callback`.	Client code posts token from provider to `/.auth/login/<provider>` for validation.
3. Establish authenticated session	App Service adds authenticated cookie to response.	App Service returns its own authentication token to client code.
4. Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in `X-ZUMO-AUTH` header.

For client browsers, App Service can automatically direct all unauthenticated users to `/.auth/login/<provider>`. You can also present users with one or more `/.auth/login/<provider>` links to sign in to your app using their provider of choice.

Authorization behavior

> **Important**
>
> By default, this feature only provides authentication, not authorization. Your application may still need to make authorization decisions, in addition to any checks you configure here.

In the Azure portal, you can configure App Service with a number of behaviors when incoming request is not authenticated. The following headings describe the options.

Restrict access

- **Allow unauthenticated requests** This option defers authorization of unauthenticated traffic to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers.

  This option provides more flexibility in handling anonymous requests. For example, it lets you present multiple sign-in providers to your users. However, you must write code.

- **Require authentication** This option will reject any unauthenticated traffic to your application. Specific action to take is specified in the **Unauthenticated requests** section.

  With this option, you don't need to write any authentication code in your app. Finer authorization, such as role-specific authorization, can be handled by inspecting the user's claims (see Access user claims).

  > **Caution**
  >
  > Restricting access in this way applies to all calls to your app, which may not be desirable for apps wanting a publicly available home page, as in many single-page applications.

  > **Note**

When using the Microsoft identity provider for users in your organization, the default behavior is that any user in your Microsoft Entra tenant can request a token for your application. You can configure the application in Microsoft Entra if you want to restrict access to your app to a defined set of users. App Service also offers some basic built-in authorization checks which can help with some validations. To learn more about authorization in Microsoft Entra, see Microsoft Entra authorization basics.

Unauthenticated requests

- **HTTP 302 Found redirect: recommended for websites** Redirects action to one of the configured identity providers. In these cases, a browser client is redirected to `/.auth/login/<provider>` for the provider you choose.
- **HTTP 401 Unauthorized: recommended for APIs** If the anonymous request comes from a native mobile app, the returned response is an `HTTP 401 Unauthorized`. You can also configure the rejection to be an `HTTP 401 Unauthorized` for all requests.
- **HTTP 403 Forbidden** Configures the rejection to be an `HTTP 403 Forbidden` for all requests.
- **HTTP 404 Not found** Configures the rejection to be an `HTTP 404 Not found` for all requests.

Token store

App Service provides a built-in token store, which is a repository of tokens that are associated with the users of your web apps, APIs, or native mobile apps. When you enable authentication with any provider, this token store is immediately available to your app. If your application code needs to access data from these providers on the user's behalf, such as:

- post to the authenticated user's Facebook timeline
- read the user's corporate data using the Microsoft Graph API

You typically must write code to collect, store, and refresh these tokens in your application. With the token store, you just retrieve the tokens when you need them and tell App Service to refresh them when they become invalid.

The ID tokens, access tokens, and refresh tokens are cached for the authenticated session, and they're accessible only by the associated user.

If you don't need to work with tokens in your app, you can disable the token store in your app's **Authentication / Authorization** page.

Logging and tracing

If you enable application logging, you will see authentication and authorization traces directly in your log files. If you see an authentication error that you didn't expect, you can conveniently find all the details by looking in your existing application logs. If you enable failed request tracing, you can see exactly what role the authentication and authorization module may have played in a failed request. In the trace logs, look for references to a module named `EasyAuthModule_32/64`.

Cross-site request forgery mitigation

App Service authentication mitigates CSRF by inspecting client requests for the following conditions:

- It's a POST request that authenticated using a session cookie.
- The request came from a known browser (as determined by the HTTP `User-Agent` header).
- The HTTP `Origin` or HTTP `Referer` header is missing or is not in the configured list of approved external domains for redirection.
- The HTTP `Origin` header is missing or is not in the configured list of CORS origins.

When a request fulfills all these conditions, App Service authentication automatically rejects it. You can workaround this mitigation logic by adding your external domain to the redirect list to **Authentication** > **Edit authentication settings** > **Allowed external redirect URLs**.

Considerations when using Azure Front Door

When using Azure App Service with authentication behind Azure Front Door or other reverse proxies, a few additional things have to be taken into consideration.

- Disable Front Door caching for the authentication workflow.

- Use the Front Door endpoint for redirects.

  App Service is usually not accessible directly when exposed via Azure Front Door. This can be prevented, for example, by exposing App Service via Private Link in Azure Front Door Premium. To prevent the authentication workflow to redirect traffic back to App Service directly, it is important to configure the application to redirect back to `https://<front-door-endpoint>/.auth/login/<provider>/callback`.

- Ensure that App Service is using the right redirect URI

In some configurations, the App Service is using the App Service FQDN as the redirect URI instead of the Front Door FQDN. This will lead to an issue when the client is being redirected to App Service instead of Front Door. To change that, the `forwardProxy` setting needs to be set to `Standard` to make App Service respect the `X-Forwarded-Host` header set by Azure Front Door.

Other reverse proxies like Azure Application Gateway or 3rd-party products might use different headers and need a different forwardProxy setting.

This configuration cannot be done via the Azure portal today and needs to be done via `az rest`:

Export settings

```
az rest --uri /subscriptions/REPLACE-ME-SUBSCRIPTIONID/resourceGroups/REPLACE-ME-RESOURCEGROUP/providers/Microsoft.Web/sites/REPLACE-ME-APPNAME/config/authsettingsV2?api-version=2020-09-01 --method get > auth.json
```

Update settings

Search for

```JSON
"httpSettings": {
  "forwardProxy": {
    "convention": "Standard"
  }
}
```

and ensure that `convention` is set to `Standard` to respect the `X-Forwarded-Host` header used by Azure Front Door.

Import settings

```
az rest --uri /subscriptions/REPLACE-ME-SUBSCRIPTIONID/resourceGroups/REPLACE-ME-RESOURCEGROUP/providers/Microsoft.Web/sites/REPLACE-ME-APPNAME/config/authsettingsV2?api-version=2020-09-01 --method put --body @auth.json
```

More resources

- How-To: Configure your App Service or Azure Functions app to use Microsoft Entra login

- Customize sign-ins and sign-outs
- Work with OAuth tokens and sessions
- Access user and application claims
- File-based configuration

Samples:

- Tutorial: Add authentication to your web app running on Azure App Service
- Tutorial: Authenticate and authorize users end-to-end in Azure App Service (Windows or Linux)
- .NET Core integration of Azure AppService EasyAuth (3rd party)
- Getting Azure App Service authentication working with .NET Core (3rd party)

Feedback

Web app that signs in users: App registration

Article • 03/25/2024

This article explains the app registration steps for a web app that signs in users.

To register your application, you can use:

- The web app quickstarts. In addition to being a great first experience with creating an application, quickstarts in the Azure portal contain a button named **Make this change for me**. You can use this button to set the properties you need, even for an existing app. Adapt the values of these properties to your own case. In particular, the web API URL for your app is probably going to be different from the proposed default, which will also affect the sign-out URI.
- The Azure portal to register your application manually.
- PowerShell and command-line tools.

Register an app by using the quickstarts

You can use the following link to bootstrap the creation of your web application:

Register an application

Register an app by using the Azure portal

Tip

Steps in this article might vary slightly based on the portal you start from.

Note

The portal to use is different depending on whether your application runs in the Microsoft Azure public cloud or in a national or sovereign cloud. For more information, see National clouds.

1. Sign in to the Microsoft Entra admin center.
2. If you have access to multiple tenants, use the **Settings** icon ⚙ in the top menu to switch to the tenant in which you want to register the application from the

Directories + subscriptions menu.

3. Browse to **Identity** > **Applications** > **App registrations**, select **New registration**.

ASP.NET Core

1. When the **Register an application** page appears, enter your application's registration information:
   a. Enter a **Name** for your application, for example `AspNetCore-WebApp`. Users of your app might see this name, and you can change it later.
   b. Choose the supported account types for your application. (See Supported account types.)
   c. For **Redirect URI**, add the type of application and the URI destination that will accept returned token responses after successful authentication. For example, enter `https://localhost:44321`.
   d. Select **Register**.
2. Under **Manage**, select **Authentication** and then add the following information:
   a. In the **Web** section, add `https://localhost:44321/signin-oidc` as a **Redirect URI**.
   b. In **Front-channel logout URL**, enter `https://localhost:44321/signout-oidc`.
   c. Select **Save**.

Register an app by using PowerShell

You can also register an application with Microsoft Graph PowerShell, using the New-MgApplication.

Here's an idea of the code. For a fully functioning code, see this sample

PowerShell

```powershell
# Connect to the Microsoft Graph API, non-interactive is not supported for
the moment (Oct 2021)
Write-Host "Connecting to Microsoft Graph"
if ($tenantId -eq "") {
   Connect-MgGraph -Scopes "User.Read.All Organization.Read.All
Application.ReadWrite.All" -Environment $azureEnvironmentName
}
else {
   Connect-MgGraph -TenantId $tenantId -Scopes "User.Read.All
Organization.Read.All Application.ReadWrite.All" -Environment
$azureEnvironmentName
}
```

```
$context = Get-MgContext
$tenantId = $context.TenantId

# Get the user running the script
$currentUserPrincipalName = $context.Account
$user = Get-MgUser -Filter "UserPrincipalName eq '$($context.Account)'"

# get the tenant we signed in to
$Tenant = Get-MgOrganization
$tenantName = $Tenant.DisplayName

$verifiedDomain = $Tenant.VerifiedDomains | where {$_.Isdefault -eq $true}
$verifiedDomainName = $verifiedDomain.Name
$tenantId = $Tenant.Id

Write-Host ("Connected to Tenant {0} ({1}) as account '{2}'. Domain is
'{3}'" -f  $Tenant.DisplayName, $Tenant.Id, $currentUserPrincipalName,
$verifiedDomainName)

# Create the webApp AAD application
Write-Host "Creating the AAD application (WebApp)"
# create the application
$webAppAadApplication = New-MgApplication -DisplayName "WebApp" `
                                                   -Web `
                                                   @{ `
                                                         RedirectUris =
"https://localhost:44321/", "https://localhost:44321/signin-oidc"; `
                                                         HomePageUrl =
"https://localhost:44321/"; `

                                                         LogoutUrl =
"https://localhost:44321/signout-oidc"; `
                                                      } `
                                                      -SignInAudience
AzureADandPersonalMicrosoftAccount `
                                                   #end of command

$currentAppId = $webAppAadApplication.AppId
$currentAppObjectId = $webAppAadApplication.Id

$tenantName = (Get-MgApplication -ApplicationId
$currentAppObjectId).PublisherDomain
#Update-MgApplication -ApplicationId $currentAppObjectId -IdentifierUris
@("https://$tenantName/WebApp")

# create the service principal of the newly created application
$webAppServicePrincipal = New-MgServicePrincipal -AppId $currentAppId -Tags
{WindowsAzureActiveDirectoryIntegratedApp}

# add the user running the script as an app owner if needed
$owner = Get-MgApplicationOwner -ApplicationId $currentAppObjectId
if ($owner -eq $null)
{
   New-MgApplicationOwnerByRef -ApplicationId $currentAppObjectId  -
BodyParameter = @{"@odata.id" =
"https://graph.microsoft.com/v1.0/directoryObjects/$user.ObjectId"}
```

```
    Write-Host "'$($user.UserPrincipalName)' added as an application owner to
app '$($webAppServicePrincipal.DisplayName)'"
}
Write-Host "Done creating the webApp application (WebApp)"
```

Next step

ASP.NET Core

Move on to the next article in this scenario, App's code configuration.

Feedback

A web app that calls web APIs: App registration

Article • 10/23/2023

A web app that calls web APIs has the same registration as a web app that signs users in. So, follow the instructions in A web app that signs in users: App registration.

However, because the web app now also calls web APIs, it becomes a confidential client application. That's why some extra registration is required. The app must share client credentials, or *secrets*, with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the Azure portal or by using a command-line tool like PowerShell.

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in Quickstart: Register an application with the Microsoft identity platform for the type of credential you want to add:

- Add a client secret
- Add a certificate

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The active-directory-dotnetcore-daemon-v2 code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see AppCreationScripts/Configure.ps1 .
- For details on how to add a **certificate** with PowerShell, see AppCreationScripts-withCert/Configure.ps1 .

API permissions

Web apps call APIs on behalf of the signed-in user. To do that, they must request *delegated permissions*. For details, see Add permissions to access your web API.

Next steps

Move on to the next article in this scenario, Code configuration.

Protected web API: App registration

Article • 05/28/2024

This article explains how to register an application for a protected web API.

For the common steps to register an app, see Quickstart: Register an application with the Microsoft identity platform.

Accepted token version

The Microsoft identity platform can issue v1.0 tokens and v2.0 tokens. For more information about these tokens, refer to Access tokens.

The token version your API may accept depends on your **Supported account types** selection when you create your web API application registration in the Azure portal.

- If the value of **Supported account types** is **Accounts in any organizational directory and personal Microsoft accounts (such as Skype, Xbox, Outlook.com)**, the accepted token version must be v2.0.
- Otherwise, the accepted token version can be v1.0.

After you create the application, you can determine or change the accepted token version by following these steps:

1. In the Microsoft Entra admin center, select your app and then select **Manifest**.
2. Find the property **accessTokenAcceptedVersion** in the manifest.
3. The value specifies to Microsoft Entra which token version the web API accepts.
    - If the value is 2, the web API accepts v2.0 tokens.
    - If the value is **null**, the web API accepts v1.0 tokens.
4. If you changed the token version, select **Save**.

The web API specifies which token version it accepts. When a client requests a token for your web API from the Microsoft identity platform, the client gets a token that indicates which token version the web API accepts.

No redirect URI

Web APIs don't need to register a redirect URI because no user is interactively signed in.

Exposed API

Other settings specific to web APIs are the exposed API and the exposed scopes or app roles.

Scopes and the Application ID URI

Scopes usually have the form `resourceURI/scopeName`. For Microsoft Graph, the scopes have shortcuts. For example, `User.Read` is a shortcut for `https://graph.microsoft.com/user.read`.

During app registration, define these parameters:

- The resource URI
- One or more scopes
- One or more app roles

By default, the application registration portal recommends that you use the resource URI `api://{clientId}`. This URI is unique but not human readable. If you change the URI, make sure the new value is unique. The application registration portal will ensure that you use a configured publisher domain.

To client applications, scopes show up as *delegated permissions* and app roles show up as *application permissions* for your web API.

Scopes also appear on the consent window that's presented to users of your app. Therefore, provide the corresponding strings that describe the scope:

- As seen by a user.
- As seen by a tenant admin, who can grant admin consent.

App roles cannot be consented to by a user (as they're used by an application that calls the web API on behalf of itself). A tenant administrator will need to consent to client applications of your web API exposing app roles. See Admin consent for details.

Expose delegated permissions (scopes)

To expose delegated permissions, or *scopes*, follow the steps in Configure an application to expose a web API.

If you're following along with the web API scenario described in this set of articles, use these settings:

- **Application ID URI**: Accept the proposed application ID URI (*api://<clientId>*) (if prompted)
- **Scope name**: *access_as_user*
- **Who can consent**: *Admins and users*
- **Admin consent display name**: *Access TodoListService as a user*
- **Admin consent description**: *Accesses the TodoListService web API as a user*
- **User consent display name**: *Access TodoListService as a user*
- **User consent description**: *Accesses the TodoListService web API as a user*
- **State**: *Enabled*

Tip

For the **Application ID URI**, you have the option to set it to the physical authority of the API, for example `https://graph.microsoft.com`. This can be useful if the URL of the API that needs to be called is known.

If your web API is called by a service or daemon app

Expose *application permissions* instead of delegated permissions if your API should be accessed by daemons, services, or other non-interactive (by a human) applications. Because daemon- and service-type applications run unattended and authenticate with their own identity, there is no user to "delegate" their permission.

Expose application permissions (app roles)

To expose application permissions, follow the steps in Add app roles to your app.

In the **Create app role** pane under **Allowed member types**, select **Applications**. Or, add the role by using the **Application manifest editor** as described in the article.

Restrict access tokens to specific clients apps

App roles are the mechanism an application developer uses to expose their app's permissions. Your web API's code should check for app roles in the access tokens it receives from callers.

To add another layer of security, a Microsoft Entra tenant administrator can configure their tenant so the Microsoft identity platform issues security tokens *only* to the client apps they've approved for API access.

To increase security by restricting token issuance only to client apps that have been assigned app roles:

1. In the Microsoft Entra admin center, select your app under **Identity** > **Applications** > **App registrations**.
2. On the application's **Overview** page, in **Essentials**, find and select its **Managed application in local directory** link to navigate to its **Enterprise Application Overview** page.
3. Under **Manage**, select **Properties**.
4. Set **Assignment required?** to **No**.
5. Select **Save**.

Microsoft Entra ID will now check for app role assignments of client applications that request access tokens for your web API. If a client app hasn't been assigned any app roles, Microsoft Entra ID returns an error message to the client similar to `_invalid_client: AADSTS501051: Application \<application name\> isn't assigned to a role for the \<web API\>_`.

⚠ Warning

DO NOT use AADSTS error codes or their message strings as literals in your application's code. The "AADSTS" error codes and the error message strings returned by Microsoft Entra ID are *not immutable*, and may be changed by Microsoft at any time and without your knowledge. If you make branching decisions in your code based on the values of either the AADSTS codes or their message strings, you put your application's functionality and stability at risk.

Next step

The next article in this series is App code configuration.

Feedback

Was this page helpful? Yes No

Provide product feedback

A web API that calls web APIs: App registration

Article • 10/23/2023

A web API that calls downstream web APIs has the same registration as a protected web API. Follow the instructions in Protected web API: App registration.

Because the web app now calls web APIs, it becomes a confidential client application. That's why extra registration information is required: the app needs to share secrets (client credentials) with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the Azure portal or by using a command-line tool like PowerShell.

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in Quickstart: Register an application with the Microsoft identity platform for the type of credential you want to add:

- Add a client secret
- Add a certificate

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The active-directory-dotnetcore-daemon-v2 code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see AppCreationScripts/Configure.ps1 .
- For details on how to add a **certificate** with PowerShell, see AppCreationScripts-withCert/Configure.ps1 .

API permissions

Web apps call APIs on behalf of users for whom the bearer token was received. The web apps need to request delegated permissions. For more information, see Add permissions to access your web API.

Next steps

Move on to the next article in this scenario, App Code configuration.

Cloud authentication with Azure Active Directory B2C in ASP.NET Core

Article • 10/25/2024

By Damien Bod

Azure Active Directory B2C (Azure AD B2C) is a cloud identity management solution for web and mobile apps. The service provides authentication for apps hosted in the cloud and on-premises. Authentication types include individual accounts, social network accounts, and federated enterprise accounts. Additionally, Azure AD B2C can provide multi-factor authentication with minimal configuration.

Tip

Microsoft Entra ID, Microsoft Entra External ID and Azure AD B2C are separate product offerings. An Entra ID tenant generally represents an organization, while an Azure AD B2C tenant or a Microsoft Entra External ID tenant can represent a collection of identities to be used with relying party applications. To learn more, see Azure AD B2C: Frequently asked questions (FAQ).

Tip

Microsoft Entra External ID for customers is Microsoft's new customer identity and access management (CIAM) solution.

In this tutorial, you'll learn how to configure an ASP.NET Core app for authentication with Azure AD B2C.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- .NET SDK. Install the latest .NET SDK for your platform.

Preparation

1. Create an Azure Active Directory B2C tenant.

2. Create a new ASP.NET Core Razor pages app:

.NET CLI

```
dotnet new razor -o azure-ad-b2c
```

The previous command creates a Razor pages app in a directory named *azure-ad-b2c*.

Tip

You may prefer to use Visual Studio to create your app.

3. Create a web app registration in the tenant. For **Redirect URI**, use `https://localhost:5001/signin-oidc`. Replace `5001` with the port used by your app when using Visual Studio generated ports.

Modify the app

1. Add the `Microsoft.Identity.Web` and `Microsoft.Identity.Web.UI` packages to the project. If you're using Visual Studio, you can use NuGet Package Manager.

.NET CLI

```
dotnet add package Microsoft.Identity.Web
dotnet add package Microsoft.Identity.Web.UI
```

In the preceding:

- `Microsoft.Identity.Web` includes the basic set of dependencies for authenticating with the Microsoft identity platform.
- `Microsoft.Identity.Web.UI` includes UI functionality encapsulated in an area named `MicrosoftIdentity`.

2. Add an `AzureADB2C` object to `appsettings.json`.

Note

When using Azure B2C user flows, you need to set the **Instance** and the PolicyId of the type of flow.

JSON

```
{
  "AzureADB2C": {
    "Instance": "https://--your-domain--.b2clogin.com",
    "Domain": "[Enter the domain of your B2C tenant, e.g.
contoso.onmicrosoft.com]",
    "TenantId": "[Enter 'common', or 'organizations' or the Tenant Id
(Obtained from the Azure portal. Select 'Endpoints' from the 'App
registrations' blade and use the GUID in any of the URLs), e.g.
da41245a5-11b3-996c-00a8-4d99re19f292]",
    "ClientId": "[Enter the Client Id (Application ID obtained from the
Azure portal), e.g. ba74781c2-53c2-442a-97c2-3d60re42f403]",
    // Use either a secret or a certificate. ClientCertificates are
recommended.
    "ClientSecret": "[Copy the client secret added to the app from the
Azure portal]",
    "ClientCertificates": [
    ],
    // the following is required to handle Continuous Access Evaluation
challenges
    "ClientCapabilities": [ "cp1" ],
    "CallbackPath": "/signin-oidc",
    // Add your policy here
    "SignUpSignInPolicyId": "B2C_1_signup_signin",
    "SignedOutCallbackPath": "/signout-callback-oidc"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

- For **Domain**, use the domain of your Azure AD B2C tenant.
- For **ClientId**, use the **Application (client) ID** from the app registration you created in your tenant.
- For **Instance**, use the domain of your Azure AD B2C tenant.
- For **SignUpSignInPolicyId**, use the user flow policy defined in the Azure B2C tenant
- Use either the **ClientSecret** or the **ClientCertificates** configuration. ClientCertificates are recommended.
- Leave all other values as they are.

3. In *Pages/Shared*, create a file named `_LoginPartial.cshtml`. Include the following code:

razor

```
@using System.Security.Principal

<ul class="navbar-nav">
@if (User.Identity?.IsAuthenticated == true)
{
        <span class="navbar-text text-dark">Hello @User.Identity?.Name!
</span>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity"
asp-controller="Account" asp-action="SignOut">Sign out</a>
        </li>
}
else
{
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity"
asp-controller="Account" asp-action="SignIn">Sign in</a>
        </li>
}
</ul>
```

The preceding code:

- Checks if the user is authenticated.
- Renders a **Sign out** or **Sign in** link as appropriate.
    - The link points to an action method on the `Account` controller in the `MicrosoftIdentity` area.

4. In *Pages/Shared/_Layout.cshtml*, add the highlighted line within the `<header>` element:

razor

```
<header>
    <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
        <div class="container">
            <a class="navbar-brand" asp-area="" asp-
page="/Index">azure_ad_b2c</a>
            <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                    aria-expanded="false" aria-label="Toggle
navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                <ul class="navbar-nav flex-grow-1">
                    <li class="nav-item">
```

```
                        <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                    </li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>
</header>
```

Adding `<partial name="_LoginPartial" />` renders the `_LoginPartial.cshtml` partial view in every page request that uses this layout.

5. In *Program.cs*, make the following changes:

   a. Add the following `using` directives:

   ```C#
   using Microsoft.Identity.Web;
   using Microsoft.Identity.Web.UI;
   using Microsoft.AspNetCore.Authentication.OpenIdConnect;
   ```

   The preceding code resolves references used in the next steps.

   b. Update the `builder.Services` lines with the following code:

   ```C#
   builder.Services.AddAuthentication(OpenIdConnectDefaults.Authenticat
   ionScheme)

   .AddMicrosoftIdentityWebApp(builder.Configuration.GetSection("AzureA
   DB2C"));

   builder.Services.AddAuthorization(options =>
   {
       // By default, all incoming requests will be authorized
   according to
       // the default policy
       options.FallbackPolicy = options.DefaultPolicy;
   });
   builder.Services.AddRazorPages(options => {
       options.Conventions.AllowAnonymousToPage("/Index");
   })
   ```

```
    .AddMvcOptions(options => { })
    .AddMicrosoftIdentityUI();
```

In the preceding code:

- Calls to the `AddAuthentication` and `AddMicrosoftIdentityWebApp` methods configure the app to use Open ID Connect, specifically configured for the Microsoft identity platform.
- `AddAuthorization` initializes ASP.NET Core authorization.
- The `AddRazorPages` call configures the app so anonymous browsers can view the Index page. All other requests require authentication.
- `AddMvcOptions` and `AddMicrosoftIdentityUI` add the required UI components for redirecting to/from Azure AD B2C.

c. Update the highlighted line to the `Configure` method:

C#

```
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
```

The preceding code enables authentication in ASP.NET Core.

Run the app

ⓘ **Note**

Use the profile which matches the Azure App registration **Redirect URIs**

1. Run the app.

.NET CLI

```
dotnet run --launch-profile https
```

2. Browse to the app's secure endpoint, for example, `https://localhost:5001/`.

- The Index page renders with no authentication challenge.
- The header includes a **Sign in** link because you're not authenticated.

3. Select the **Privacy** link.
    - The browser is redirected to your tenant's configured authentication method.
    - After signing in, the header displays a welcome message and a **Sign out** link.

Next steps

In this tutorial, you learned how to configure an ASP.NET Core app for authentication with Azure AD B2C.

Now that the ASP.NET Core app is configured to use Azure AD B2C for authentication, the Authorize attribute can be used to secure your app. Continue developing your app by learning to:

- Customize the Azure AD B2C user interface.
- Configure password complexity requirements.
- Enable multi-factor authentication.
- Configure additional identity providers, such as Microsoft, Facebook, Google, Amazon, Twitter, and others.
- Use the Microsoft Graph API to retrieve additional user information, such as group membership, from the Azure AD B2C tenant.
- How to secure a Web API built with ASP.NET Core using the Azure AD B2C.
- Tutorial: Grant access to an ASP.NET web API using Azure Active Directory B2C.

Enable authentication in your own web API by using Azure AD B2C

Article • 01/11/2024

To authorize access to a web API, you can serve only requests that include a valid access token that Azure Active Directory B2C (Azure AD B2C) issues. This article shows you how to enable Azure AD B2C authorization to your web API. After you complete the steps in this article, only users who obtain a valid access token will be authorized to call your web API endpoints.

Prerequisites

Before you begin, read one of the following articles, which discuss how to configure authentication for apps that call web APIs. Then, follow the steps in this article to replace the sample web API with your own web API.

- Configure authentication in a sample ASP.NET Core application
- Configure authentication in a sample single-page application (SPA)

Overview

Token-based authentication ensures that requests to a web API includes a valid access token.

The app completes the following steps:

1. It authenticates users with Azure AD B2C.

2. It acquires an access token with the required permissions (scopes) for the web API endpoint.

3. It passes the access token as a bearer token in the authentication header of the HTTP request by using this format:

```HTTP
Authorization: Bearer <access token>
```

The web API completes the following steps:

1. It reads the bearer token from the authorization header in the HTTP request.

2. It validates the token.

3. It validates the permissions (scopes) in the token.

4. It reads the claims that are encoded in the token (optional).

5. It responds to the HTTP request.

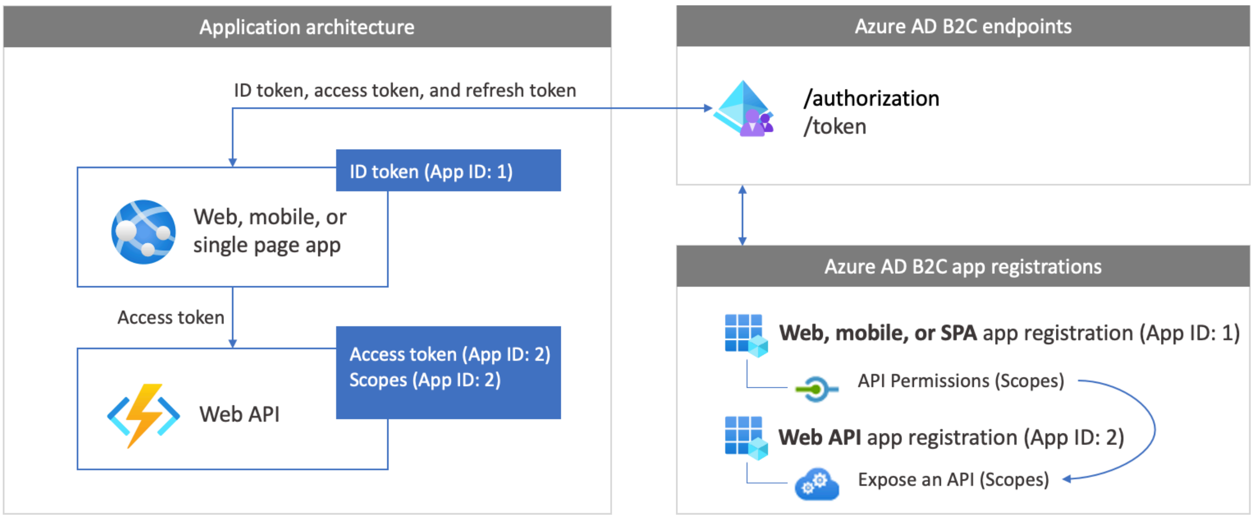
App registration overview

To enable your app to sign in with Azure AD B2C and call a web API, you need to register two applications in the Azure AD B2C directory.

- The *web, mobile, or SPA application* registration enables your app to sign in with Azure AD B2C. The app registration process generates an *Application ID*, also known as the *client ID*, which uniquely identifies your application (for example, *App ID: 1*).

- The *web API* registration enables your app to call a protected web API. The registration exposes the web API permissions (scopes). The app registration process generates an *Application ID*, which uniquely identifies your web API (for example, *App ID: 2*). Grant your app (App ID: 1) permissions to the web API scopes (App ID: 2).

The application registrations and the application architecture are described in the following diagram:



Prepare your development environment

In the next sections, you create a new web API project. Select your programming language, ASP.NET Core or Node.js. Make sure you have a computer that's running either of the following software:

ASP.NET Core

- Visual Studio Code
- C# for Visual Studio Code (latest version)
- .NET 5.0 SDK

Step 1: Create a protected web API

Create a new web API project. First, select the programming language you want to use, **ASP.NET Core** or **Node.js**.

ASP.NET Core

Use the dotnet new command. The `dotnet new` command creates a new folder named *TodoList* with the web API project assets. Open the directory, and then open Visual Studio Code.

.NET CLI

```
dotnet new webapi -o TodoList
cd TodoList
code .
```

When you're prompted to "add required assets to the project," select **Yes**.

Step 2: Install the dependencies

Add the authentication library to your web API project. The authentication library parses the HTTP authentication header, validates the token, and extracts claims. For more information, review the documentation for the library.

ASP.NET Core

To add the authentication library, install the package by running the following command:

.NET CLI

```
dotnet add package Microsoft.Identity.Web
```

Step 3: Initiate the authentication library

Add the necessary code to initiate the authentication library.

ASP.NET Core

Open *Startup.cs* and then, at the beginning of the class, add the following `using` declarations:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Identity.Web;
```

Find the `ConfigureServices(IServiceCollection services)` function. Then, before the `services.AddControllers();` line of code, add the following code snippet:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds Microsoft Identity platform (Azure AD B2C) support to
protect this Api
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApi(options =>
    {
        Configuration.Bind("AzureAdB2C", options);

        options.TokenValidationParameters.NameClaimType = "name";
    },
    options => { Configuration.Bind("AzureAdB2C", options); });
    // End of the Microsoft Identity platform block

    services.AddControllers();
}
```

Find the `Configure` function. Then, immediately after the `app.UseRouting();` line of code, add the following code snippet:

C#

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

After the change, your code should look like the following snippet:

```C#
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    // Add the following line
    app.UseAuthentication();
    // End of the block you add

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Step 4: Add the endpoints

Add two endpoints to your web API:

- Anonymous `/public` endpoint. This endpoint returns the current date and time. Use it to debug your web API with anonymous calls.
- Protected `/hello` endpoint. This endpoint returns the value of the `name` claim within the access token.

To add the anonymous endpoint:

ASP.NET Core

Under the */Controllers* folder, add a *PublicController.cs* file, and then add it to the following code snippet:

C#

```csharp
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace TodoList.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class PublicController : ControllerBase
    {
        private readonly ILogger<PublicController> _logger;

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

        [HttpGet]
        public ActionResult Get()
        {
            return Ok( new {date = DateTime.UtcNow.ToString()});
        }
    }
}
```

To add the protected endpoint:

ASP.NET Core

Under the */Controllers* folder, add a *HelloController.cs* file, and then add it to the following code:

C#

```csharp
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.Identity.Web.Resource;

namespace TodoList.Controllers
{
    [Authorize]
    [RequiredScope("tasks.read")]
    [ApiController]
    [Route("[controller]")]
    public class HelloController : ControllerBase
    {
        private readonly ILogger<HelloController> _logger;
        private readonly IHttpContextAccessor _contextAccessor;

        public HelloController(ILogger<HelloController> logger,
IHttpContextAccessor contextAccessor)
        {
            _logger = logger;
            _contextAccessor = contextAccessor;
        }

        [HttpGet]
        public ActionResult Get()
        {
            return Ok( new { name = User.Identity.Name});
        }
    }
}
```

The `HelloController` controller is decorated with the AuthorizeAttribute, which limits access to authenticated users only.

The controller is also decorated with the `[RequiredScope("tasks.read")]`. The RequiredScopeAttribute verifies that the web API is called with the right scopes, `tasks.read`.

Step 5: Configure the web server

In a development environment, set the web API to listen on incoming HTTP or HTTPS requests port number. In this example, use HTTP port 6000 and HTTPS port 6001. The base URI of the web API will be `http://localhost:6000` for HTTP and `https://localhost:6001` for HTTPS.

ASP.NET Core

Add the following JSON snippet to the *appsettings.json* file.

JSON

```json
"Kestrel": {
    "EndPoints": {
      "Http": {
        "Url": "http://localhost:6000"
      },
      "Https": {
         "Url": "https://localhost:6001"
        }
    }
  }
```

Step 6: Configure the web API

Add configurations to a configuration file. The file contains information about your Azure AD B2C identity provider. The web API app uses this information to validate the access token that the web app passes as a bearer token.

ASP.NET Core

Under the project root folder, open the *appsettings.json* file, and then add the following settings:

JSON

```
{
  "AzureAdB2C": {
    "Instance": "https://contoso.b2clogin.com",
    "Domain": "contoso.onmicrosoft.com",
    "ClientId": "<web-api-app-application-id>",
    "SignedOutCallbackPath": "/signout/<your-sign-up-in-policy>",
    "SignUpSignInPolicyId": "<your-sign-up-in-policy>"
  },
  // More settings here
}
```

In the *appsettings.json* file, update the following properties:

Expand table

Section	Key	Value
AzureAdB2C	Instance	The first part of your Azure AD B2C tenant name (for example, `https://contoso.b2clogin.com`).
AzureAdB2C	Domain	Your Azure AD B2C tenant full tenant name (for example, `contoso.onmicrosoft.com`).
AzureAdB2C	ClientId	The web API application ID. In the preceding diagram, it's the application with *App ID: 2*. To learn how to get your web API application registration ID, see Prerequisites.
AzureAdB2C	SignUpSignInPolicyId	The user flows, or custom policy. To learn how to get

Section	Key	Value
		your user flow or policy, see Prerequisites.

Step 7: Run and test the web API

Finally, run the web API with your Azure AD B2C environment settings.

ASP.NET Core

In the command shell, start the web app by running the following command:

```bush
dotnet run
```

You should see the following output, which means that your app is up and running and ready to receive requests.

```
Now listening on: http://localhost:6000
```

To stop the program, in the command shell, select Ctrl+C. You can rerun the app by using the `node app.js` command.

Tip

Alternatively, to run the `dotnet run` command, you can use the Visual Studio Code debugger. Visual Studio Code's built-in debugger helps accelerate your edit, compile, and debug loop.

Open a browser and go to `http://localhost:6000/public`. In the browser window, you should see the following text displayed, along with the current date and time.

Step 8: Call the web API from your app

Try to call the protected web API endpoint without an access token. Open a browser and go to `http://localhost:6000/hello`. The API will return an unauthorized HTTP error message, confirming that web API is protected with a bearer token.

Continue to configure your app to call the web API. For guidance, see the Prerequisites section.

Watch this video to learn about some best practices when you integrate Azure AD B2C with an API.
https://www.youtube-nocookie.com/embed/wuUu71RcsIo

Next steps

Get the complete example on GitHub:

ASP.NET Core

- Get the web API by using the Microsoft identity library.

Feedback

Use cookie authentication without ASP.NET Core Identity

Article • 04/25/2024

By Rick Anderson

ASP.NET Core Identity is a complete, full-featured authentication provider for creating and maintaining logins. However, a cookie-based authentication provider without ASP.NET Core Identity can be used. For more information, see Introduction to Identity on ASP.NET Core.

View or download sample code (how to download)

For demonstration purposes in the sample app, the user account for the hypothetical user, Maria Rodriguez, is hardcoded into the app. Use the **Email** address `maria.rodriguez@contoso.com` and any password to sign in the user. The user is authenticated in the `AuthenticateUser` method in the `Pages/Account/Login.cshtml.cs` file. In a real-world example, the user would be authenticated against a datastore.

Add cookie authentication

- Add the Authentication Middleware services with the AddAuthentication and AddCookie methods.
- Call UseAuthentication and UseAuthorization to set the `HttpContext.User` property and run the Authorization Middleware for requests. `UseAuthentication` and `UseAuthorization` must be called before `Map` methods such as MapRazorPages and MapDefaultControllerRoute

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.Authenticati
onScheme)
    .AddCookie();

builder.Services.AddHttpContextAccessor();
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

AuthenticationScheme passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and the app needs to authorize with a specific scheme. Setting the `AuthenticationScheme` to CookieAuthenticationDefaults.AuthenticationScheme provides a value of `"Cookies"` for the scheme. Any string value can be used that distinguishes the scheme.

The app's authentication scheme is different from the app's cookie authentication scheme. When a cookie authentication scheme isn't provided to AddCookie, it uses CookieAuthenticationDefaults.AuthenticationScheme. The CookieAuthenticationDefaults.AuthenticationScheme GitHub Source shows it's set to `"Cookies"`.

The authentication cookie's IsEssential property is set to `true` by default. Authentication cookies are allowed when a site visitor hasn't consented to data collection. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

The CookieAuthenticationOptions class is used to configure the authentication provider options.

Configure CookieAuthenticationOptions in the AddCookie method:

```C#
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.Authenticati
onScheme)
    .AddCookie(options =>
    {
        options.ExpireTimeSpan = TimeSpan.FromMinutes(20);
        options.SlidingExpiration = true;
        options.AccessDeniedPath = "/Forbidden/";
    });

builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Cookie Policy Middleware

The Cookie Policy Middleware (GitHub Source) UseCookiePolicy enables cookie policy capabilities. Middleware is processed in the order it's added:

C#

```
app.UseCookiePolicy(cookiePolicyOptions);
```

Use CookiePolicyOptions provided to the Cookie Policy Middleware to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

The default MinimumSameSitePolicy value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

```C#
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect the setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.

Expand table

MinimumSameSitePolicy	Cookie.SameSite	Resultant Cookie.SameSite setting
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

Create an authentication cookie

To create a cookie holding user information, construct a ClaimsPrincipal. The user information is serialized and stored in the cookie.

Create a ClaimsIdentity with any required Claims and call SignInAsync to sign in the user. `Login.cshtml.cs` in the sample app contains the following code:

```C#
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    ReturnUrl = returnUrl;

    if (ModelState.IsValid)
    {
        // Use Input.Email and Input.Password to authenticate the user
        // with your custom authentication logic.
        //
        // For demonstration purposes, the sample validates the user
        // on the email address maria.rodriguez@contoso.com with
        // any password that passes model validation.

        var user = await AuthenticateUser(Input.Email, Input.Password);

        if (user == null)
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, user.Email),
            new Claim("FullName", user.FullName),
            new Claim(ClaimTypes.Role, "Administrator"),
        };

        var claimsIdentity = new ClaimsIdentity(
            claims, CookieAuthenticationDefaults.AuthenticationScheme);

        var authProperties = new AuthenticationProperties
        {
            //AllowRefresh = <bool>,
            // Refreshing the authentication session should be allowed.

            //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
            // The time at which the authentication ticket expires. A
            // value set here overrides the ExpireTimeSpan option of
            // CookieAuthenticationOptions set with AddCookie.

            //IsPersistent = true,
            // Whether the authentication session is persisted across
            // multiple requests. When used with cookies, controls
            // whether the cookie's lifetime is absolute (matching the
            // lifetime of the authentication ticket) or session-based.

            //IssuedUtc = <DateTimeOffset>,
            // The time at which the authentication ticket was issued.

            //RedirectUri = <string>
            // The full path or absolute URI to be used as an http
            // redirect response value.
        };

        await HttpContext.SignInAsync(
            CookieAuthenticationDefaults.AuthenticationScheme,
            new ClaimsPrincipal(claimsIdentity),
            authProperties);
        _logger.LogInformation("User {Email} logged in at {Time}.",
            user.Email, DateTime.UtcNow);

        return LocalRedirect(Url.GetLocalUrl(returnUrl));
    }

    // Something failed. Redisplay the form.
    return Page();
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`SignInAsync` creates an encrypted cookie and adds it to the current response. If `AuthenticationScheme` isn't specified, the default scheme is used.

RedirectUri is only used on a few specific paths by default, for example, the login path and logout paths. For more information see the CookieAuthenticationHandler source.

ASP.NET Core's Data Protection system is used for encryption. For an app hosted on multiple machines, load balancing across apps, or using a web farm, configure data protection to use the same key ring and app identifier.

Sign out

To sign out the current user and delete their cookie, call SignOutAsync:

```C#
public async Task OnGetAsync(string returnUrl = null)
{
    if (!string.IsNullOrEmpty(ErrorMessage))
    {
        ModelState.AddModelError(string.Empty, ErrorMessage);
    }

    // Clear the existing external cookie
    await HttpContext.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);

    ReturnUrl = returnUrl;
}
```

If `CookieAuthenticationDefaults.AuthenticationScheme` or "Cookies" isn't used as the scheme, supply the scheme used when configuring the authentication provider. Otherwise, the default scheme is used. For example, if "ContosoCookie" is used as the scheme, supply the scheme used when configuring the authentication provider.