Custom storage providers for ASP.NET Core Identity

07/23/2019 • 9 minutes to read • +5

In this article

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 walkthrough.

View or download sample from GitHub.

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 Core CLI, add `-au Individual`:

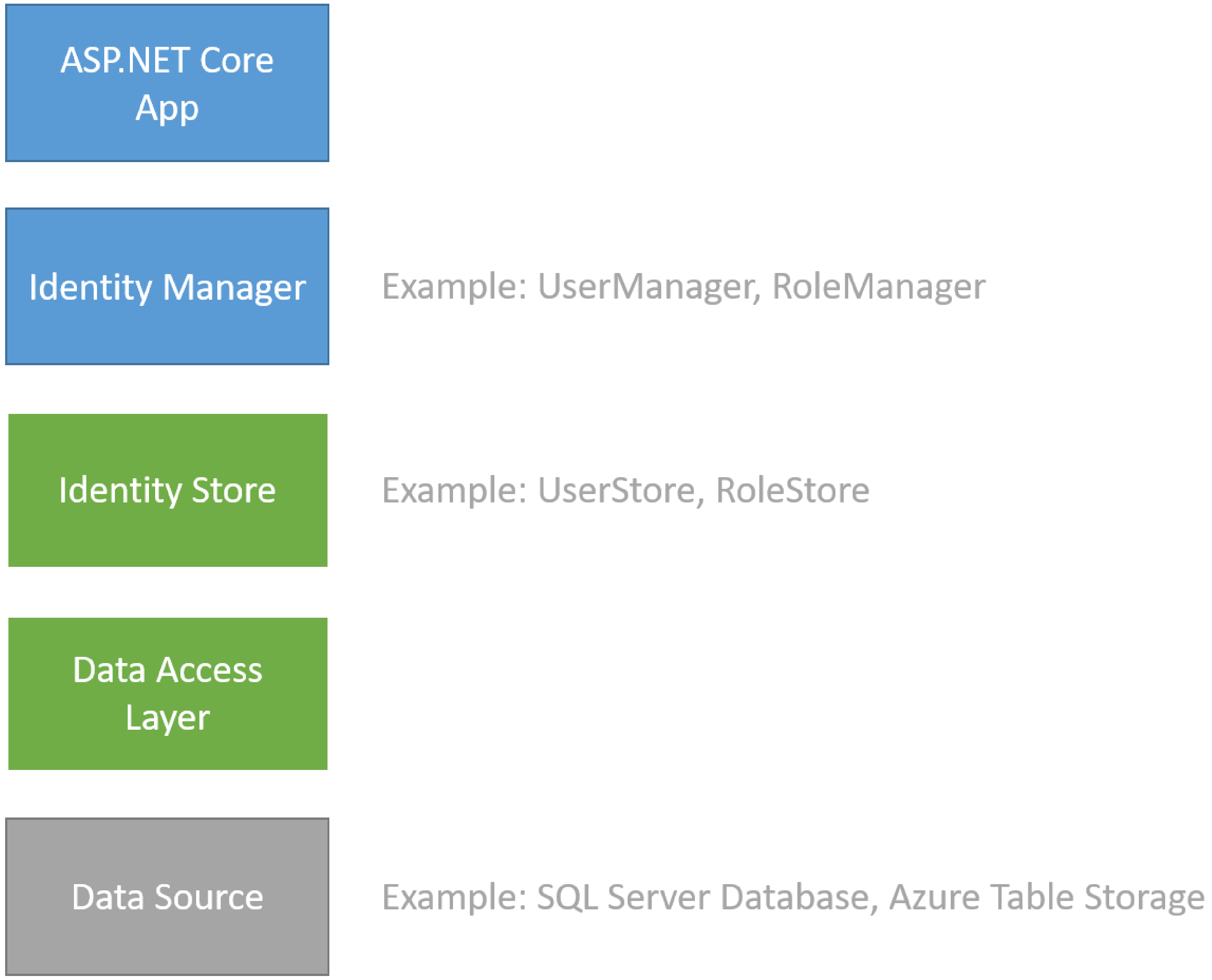
.NET Core CLI Copy

```
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# Copy

```
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:

C# Copy

```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# Copy

```
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:

C# Copy

```
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. In the `ConfigureServices` method, change the `AddIdentity` method to use your 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.

Example:

C# Copy

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Add identity types
    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddDefaultTokenProviders();

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

    // additional configuration
}
```

References

- Custom Storage Providers for ASP.NET 4.x Identity
- ASP.NET Core Identity: This repository includes links to community maintained store providers.

Is this page helpful?

Yes No