1. **Аутентификация** – это процесс проверки подлинности пользователя. ASP.NET Core использует куки для хранения данных о сеансе пользователя, и Identity Framework позволяет управлять этой информацией.
2. **Авторизация** – это процесс определения прав пользователя на выполнение определенных действий или доступ к ресурсам. ASP.NET Core предоставляет политики и роли для управления доступом.

**Идентификация** – заявление пользователя о себе.

При создании проекта: тип аутентификации **Individual User Accounts.**

# Структура проекта

Добавление новых частей в проект:

* **Areas/Identity.** В этой папке находится весь функционал, связанный с аутентификацией и управлением пользователями. Она содержит контроллеры, представления и страницы для регистрации, входа, восстановления пароля и управления профилем. Обычно здесь можно найти папку **Pages**, в которой находятся Razor Pages, такие как Login.cshtml, Register.cshtml, Manage и другие. **Файл \_ViewStart.cshtml** появился в папке **Areas/Identity** для настройки общего поведения представлений (Views) внутри этой конкретной области.
* **Data.** Здесь находится файл контекста базы данных — класс, который расширяет IdentityDbContext. Этот контекст управляет всеми таблицами, связанными с пользователями, ролями, правами и другими аспектами Identity. Чаще всего класс называется ApplicationDbContext и настраивается для подключения к базе данных через строку подключения в appsettings.json. Контекст базы данных — это основной класс, который связывает вашу ASP.NET Core или Entity Framework Core с базой данных. **Пустой ApplicationDbContext** — это нормальное явление, если он только что был создан с использованием шаблона аутентификации *Individual User Accounts* в ASP.NET Core. В этом случае ApplicationDbContext наследуется от IdentityDbContext, и все необходимые таблицы для пользователей и ролей уже автоматически включены.
  + **Migrations.** Миграции — это механизм Entity Framework Core, который помогает управлять изменениями схемы базы данных. Когда вы добавляете или изменяете модели в коде (например, добавляете новые свойства в классы), миграции фиксируют эти изменения и создают скрипты для их применения в базе данных.

Все настройки подключения бд заданы в файле *appsettings.json* в узле ConnectionStrings.

|  |
| --- |
| Add-Migration InitialCreate  Update-Database |

@inject SignInManager<IdentityUser> SignInManager

@inject UserManager<IdentityUser> UserManager

* @inject SignInManager<IdentityUser> SignInManager: внедряет сервис SignInManager, который предоставляет методы для работы с процессами аутентификации, например, проверяет, вошел ли пользователь в систему.
* @inject UserManager<IdentityUser> UserManager: внедряет сервис UserManager, который предоставляет методы для управления пользователями, такие как создание, удаление, обновление пользователей и получение их данных.

@if (SignInManager.IsSignedIn(User)) метод IsSignedIn проверяет, вошел ли пользователь в систему (т.е., аутентифицирован ли текущий User). Если да, то отображается приветствие с именем пользователя и кнопка выхода. Если нет, то показываются ссылки для регистрации и входа.

# Генерация страниц

Генерация страниц: dotnet aspnet-codegenerator identity -dc ApplicationDbContext

# UserManager и SignInManager

В ASP.NET Core объекты, такие как UserManager, SignInManager, RoleManager, и ILogger, передаются в конструктор контроллера с помощью **внедрения зависимостей** (Dependency Injection, DI). ASP.NET Core имеет встроенный контейнер для управления зависимостями, который автоматически создаёт и передаёт экземпляры зависимостей в конструкторы контроллеров и других компонентов.

В Startup добавлены сервисы Identity и в контроллере через конструктор мы можем их получить. Сервис по управлению пользователями – UserManager и сервис  SignInManager, который позволяет аутентифицировать пользователя и устанавливать или удалять его куки.

UserManager<TUser> — это класс, который предоставляет методы для управления пользователями. Этот класс работает с типом TUser, который обычно является либо IdentityUser (стандартная модель пользователя), либо вашим кастомным классом (например, ApplicationUser), который наследуется от IdentityUser и может содержать дополнительные свойства.

* **ChangePasswordAsync(user, old, new): изменяет пароль пользователя**
* **CreateAsync(user): создает нового пользователя**
* **DeleteAsync(user): удаляет пользователя**
* **FindByIdAsync(id): ищет пользователя по id**
* **FindByEmailAsync(email): ищет пользователя по email**
* **FindByNameAsync(name): ищет пользователя по нику**
* **UpdateAsync(user): обновляет пользователя**
* **Users: возвращает всех пользователей**
* **AddToRoleAsync(user, role): добавляет для пользователя user роль role**
* **GetRolesAsync (user): возвращает список ролей, к которым принадлежит пользователь user**
* **IsInRoleAsync(user, name): возвращает true, если пользователь user принадлежит роли name**
* **RemoveFromRoleAsync(user, name): удаляет роль name у пользователя user**

SignInManager<TUser> — это класс, который управляет процессом входа (и выхода) в систему. SignInManager использует UserManager для работы с пользователями, но сам отвечает за проверку учетных данных и настройку куков для авторизации.

**Вход по паролю**: PasswordSignInAsync выполняет попытку авторизации на основе имени пользователя и пароля. (имя, пароль, запоминать ли, блокировать ли после нескольких неудачных попыток входа?). Метод PasswordSignInAsync возвращает объект SignInResult, который хранит результат операции входа. SignInResult может иметь следующие состояния:

* **SignInResult.Success** — если вход выполнен успешно.
* **SignInResult.Failed** — если предоставленные учетные данные неверны.
* **SignInResult.LockedOut** — если учетная запись пользователя заблокирована после слишком большого количества неудачных попыток входа (если учетная запись настроена на блокировку).
* **SignInResult.RequiresTwoFactor** — если для учетной записи включена двухфакторная аутентификация, и для завершения входа нужно пройти дополнительную проверку.

**Запоминание входа**: SignInAsync позволяет войти с конкретным пользователем и настраивает продолжительность сессии (куков). Первый параметр – объект пользователя, второй – постоянный ли вход. Возвращает объект IdentityResult, который представляет результат операции по созданию пользователя IdentityResult — это специальный класс, используемый в ASP.NET Core Identity для представления результата выполнения операций. У него есть два основных свойства:

* **Succeeded** — логическое значение, указывающее, прошла ли операция успешно (true) или нет (false).
* **Errors** — коллекция объектов типа IdentityError, содержащая ошибки, возникшие при выполнении операции. Если Succeeded равно false, в Errors будут описания всех ошибок.

**Выход**: SignOutAsync удаляет куки и завершает сессию пользователя.

**Проверка двухфакторной аутентификации**: TwoFactorSignInAsync проверяет код двухфакторной аутентификации для пользователя.

# IdentityUser

**IdentityUser** — это базовый класс в ASP.NET Core Identity, который представляет пользователя приложения. Этот класс предоставляет стандартные свойства и функции для работы с пользователями, такие как хранение имен, паролей, адресов электронной почты и других базовых данных для идентификации и аутентификации пользователей.

Если нужно добавить дополнительные свойства, можно создать класс ApplicationUser, который наследуется от IdentityUser.

* **Id**: уникальный идентификатор пользователя
* **UserName**: ник пользователя
* **Email**: электронный адрес пользователя
* **Logins**: коллекция логинов, которые использовались пользователем для входа через сторонние сервисы (Google, Facebook и т.д.)
* **Claims**: коллекция клеймов или дополнительных объектов, которые используются для авторизации пользователя
* **PasswordHash**: хеш пароля. В базе данных напрямую не хранится пароль, а только его хеш.
* **Roles**: набор ролей, к которым принадлежит пользователь
* **PhoneNumber**: номер телефона
* **SecurityStamp**: некоторое специальное значение, которое меняется при смене аутентификационных данных, например, пароля
* **AccessFailedCount**: количество неудачных входов пользователя в систему
* **EmailConfirmed**: подтвержден ли адрес электронной почты
* **PhoneNumberConfirmed**: подтвержден ли номер телефона

# IdentityRole

По умолчанию роль в ASP.NET Core Identity представлена классом **IdentityRole**, который определяет три свойства:

* Id: идентификатор роли
* Name: название роли
* Users: коллекция объектов IdentityUserRole, через которые пользователи ассоциированы с данной ролью

Но, как правило, за работу с ролями в ASP.NET Core Identity отвечает класс **RoleManager<T>**, где параметр T представляет класс, описывающий роль. Этот класс представляет ряд методов ля управления ролями:

* **CreateAsync(role)**: создает новую роль
* **DeleteAsync(role)**: удаляет роль
* **FindByIdAsync(id)**: возвращает роль по id
* **FindByNameAsync(name)**: возвращает роль по названию
* **RoleExistsAsync(name)**: возвращает true, если роль с данным именем существует
* **UpdateAsync(role)**: обновляет роль
* **Roles**: возвращает все роли

# Program.cs

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

Эти строки настраивают подключение к базе данных SQL Server, используя строку подключения "DefaultConnection", определённую в appsettings.json. Если строка подключения не найдена, выбрасывается исключение. AddDbContext<ApplicationDbContext> регистрирует ApplicationDbContext как сервис, доступный через DI, что позволяет ASP.NET Core управлять жизненным циклом базы данных и подключением к ней.

builder.Services.AddDatabaseDeveloperPageExceptionFilter(); Этот фильтр добавляет обработку исключений, связанных с базой данных, на страницы разработчика.

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

.AddEntityFrameworkStores<ApplicationDbContext>();

1. AddDefaultIdentity<IdentityUser>: Добавляет ASP.NET Core Identity с IdentityUser как базовым классом для пользователей. Это означает, что ваша аутентификация и управление пользователями будут базироваться на стандартном классе IdentityUser.
2. options => options.SignIn.RequireConfirmedAccount = true: Устанавливает требование подтверждения учетной записи перед входом (например, подтверждение через email).
3. .AddEntityFrameworkStores<ApplicationDbContext>(): Настраивает Identity для использования ApplicationDbContext (контекста базы данных) в качестве хранилища пользователей и ролей. Таким образом, данные пользователей будут храниться в базе данных, подключенной через ApplicationDbContext.

[Required]

[DataType(DataType.Password)]

[Display(Name = "Пароль")]

public string Password { get; set; }

 **[Required]**: Как и для Email, этот атрибут указывает, что поле Password обязательно для заполнения. Если пользователь не введет пароль, ASP.NET отобразит ошибку валидации.

 **[DataType(DataType.Password)]**: Атрибут [DataType(DataType.Password)] указывает, что это поле предназначено для ввода пароля. В браузере это повлияет на отображение поля: символы будут скрыты, и появятся точки или звездочки вместо букв. Этот атрибут также влияет на безопасность, так как браузер не сохранит это поле в обычном виде.

 **[Display(Name = "Пароль")]**: Этот атрибут задает отображаемое имя "Пароль" для поля Password. В интерфейсе пользователя, при выводе этого поля, будет использоваться текст "Пароль".

# Контекст данных

По умолчанию данный класс наследует весь функционал от IdentityDbContext. Так, мы можем получить содержимое таблиц из бд с помощью следующих свойств:

* **Users**: набор объектов IdentityUser, соответствует таблице пользователей
* **Roles**: набор объектов IdentityRole, соответствует таблице ролей
* **RoleClaims**: набор объектов IdentityRoleClaim, соответствует таблице связи ролей и объектов claims
* **UserLogins**: набор объектов IdentityUserLogin, соответствует таблице связи пользователей с их логинами их внешних сервисов
* **UserClaims**: набор объектов IdentityUserClaim, соответствует таблице связи пользователей и объектов claims
* **UserRoles**: набор объектов IdentityUserRole, соответствует таблице, которая сопоставляет пользователей и их роли
* **UserTokens**: набор объектов IdentityUserToken, соответствует таблице токенов пользователей

# Почему для пользователя определена отдельная ViewModel?

Пользователи в приложении представлены классом User, который наследует множество свойств. Однако нам необязательно всех их устанавливать. Достаточно установить ключевые свойства вроде логина и пароля. Для этого лучше воспользоваться вспомогательной моделью, которая установит все необходимые свойства.

* 1. **[Required]** — атрибут, указывающий, что поле является обязательным. Если пользователь не заполнит его, то ASP.NET Core выдаст ошибку валидации.
  2. **[Display(Name = "Название")]** — этот атрибут задает имя для отображения рядом с полем в интерфейсе (вместо имени свойства).
  3. **[DataType(DataType.Password)]** — указывает тип данных для пароля. При этом ASP.NET Core обрабатывает это поле как парольное (например, скрывая вводимые символы в текстовом поле).
  4. **[Compare("Password", ErrorMessage = "Пароли не совпадают")]** — атрибут, указывающий, что это поле должно совпадать с полем Password. Если значения не совпадают, появится ошибка с указанным сообщением "Пароли не совпадают".

В ASP.NET Core MVC **if (ModelState.IsValid)** проверяет, прошла ли модель валидацию успешно. ModelState — это объект, который содержит информацию о состоянии модели, включая результаты проверки каждого свойства модели на соответствие установленным атрибутам валидации (таким, как [Required], [Compare] и другие).

Когда данные из формы отправляются на сервер, ASP.NET Core MVC автоматически проверяет модель на основе атрибутов валидации, которые были указаны для её свойств. Например, атрибуты [Required], [EmailAddress], [Range] и другие проверяют, что поля заполнены, содержат корректные значения и соответствуют нужным форматам.

**ModelState.AddModelError** — метод, который добавляет ошибки валидации в объект ModelState. ModelState используется ASP.NET Core MVC для хранения информации о валидации данных, и ошибки, добавленные с помощью ModelState.AddModelError, автоматически отображаются в представлении, если есть соответствующий блок отображения ошибок (например, @Html.ValidationSummary() в представлении).

**string.Empty** — первый параметр метода AddModelError определяет, к какому полю относятся ошибки. Если в качестве ключа указано string.Empty, то ошибки относятся ко всей модели в целом, а не к какому-либо отдельному полю. Это значит, что ошибки будут отображены как общие ошибки формы, а не привязаны к конкретному полю.

Вопрос: где в бд хранится возраст пользователей.

# Авторизация

Метод Url.IsLocalUrl проверяет, принадлежит ли URL-адрес текущему сайту или домену.

Атрибут [ValidateAntiForgeryToken] используется в ASP.NET Core MVC для защиты от атак типа CSRF (Cross-Site Request Forgery), или межсайтовой подделки запросов. Этот атрибут проверяет, что запрос поступил из того же источника, на который он направлен, а не от внешнего сайта, тем самым защищая действия, выполняемые от имени пользователя, от несанкционированного выполнения.

# Роли

Для работы с ролями ASP.NET Core Identity предоставляет класс **IdentityRole**. В этом классе определяется несколько свойств:

public virtual TKey Id { get; set; }

public virtual string Name { get; set; }

public virtual string NormalizedName { get; set; }

public virtual string ConcurrencyStamp { get; set; }

Доступ к констроллеру по ролям:

[Authorize(Roles="admin")]