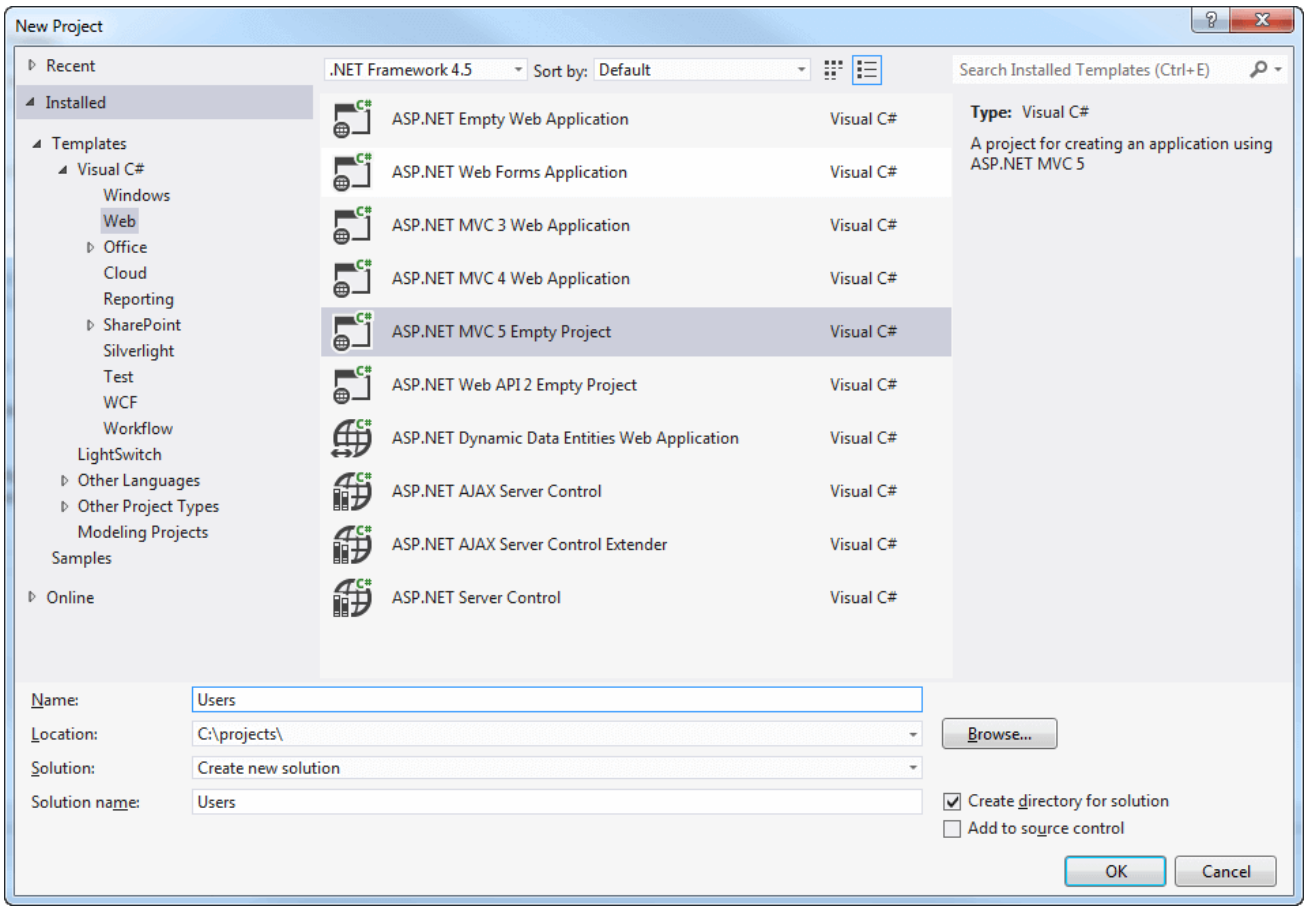
Настройка ASP.NET Identity

1 ***ASP.NET Identity*** — это новый API-интерфейс от Microsoft для управления пользователями в приложениях ASP.NET, призванный заменить устаревший подход на основе Membership API. В этой и последующих статьях я продемонстрирую возможности настройки Identity и создам простой инструмент администрирования пользователей, который управляет учетными записями, хранящимися в базе данных. ASP.NET Identity поддерживает другие типы учетных записей пользователей, такие как те, что хранятся с помощью Active Directory, но я не буду их описывать, так как они используются редко в веб-приложениях.

Пример проекта

Для целей тестирования платформы Identity мы будем использовать простой проект ASP.NET MVC с названием Users. Выберите пустой шаблон (Empty) на этапе создания нового проекта:

Пройди тесты C# тест (легкий) .NET тест (средний)



После создания проекта нам необходимо будет добавить ссылки и клиентские библиотеки для работы с проектом. Мы будем использовать *библиотеку* *Bootstrap* для стилизации приложения, поэтому введите следующую команду в окно Package Manager Console среды Visual Studio и нажмите клавишу Enter:

```
Install-Package -version 3.0.3 bootstrap
```

Теперь необходимо добавить в приложение контроллер Home, который в дальнейшем будет содержать код примеров. Определение контроллера приведено в примере ниже. Мы будем использовать этот контроллер для работы с учетными данными пользователей. Метод действия Index() возвращает представление для главной страницы приложения:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Web.Mvc;

namespace AuthUsers.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            Dictionary<string, object> data
                = new Dictionary<string, object>();
            data.Add("Ключ", "Значение");

            return View(data);
        }
    }
}
```

Далее создайте представление, щелкнув правой кнопкой мыши по названию метода `Index()` и выбрав в контекстном меню команду `Add View`. В появившемся модальном окне задайте имя представления `Index` и выберите пустой шаблон представления без модели (`Template to Empty (without model)`). Когда вы щелкните по кнопке `Add`, в приложение будет добавлено представление `Index.cshtml` (в папке `~/Views/Home`), а также будет добавлен файл компоновки `~/Views/Shared/_Layout.cshtml`. Мы будем использовать базовую компоновку для всех представлений. Ниже показано содержимое этого файла:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
</head>
<body class="container">
    <div class="container">
        @RenderBody()
    </div>
    <style>
        .container { padding-top: 10px; }
        .validation-summary-errors { color: #f00; }
    </style>
</body>
</html>
```

Ниже показана разметка представления Index.cshtml:

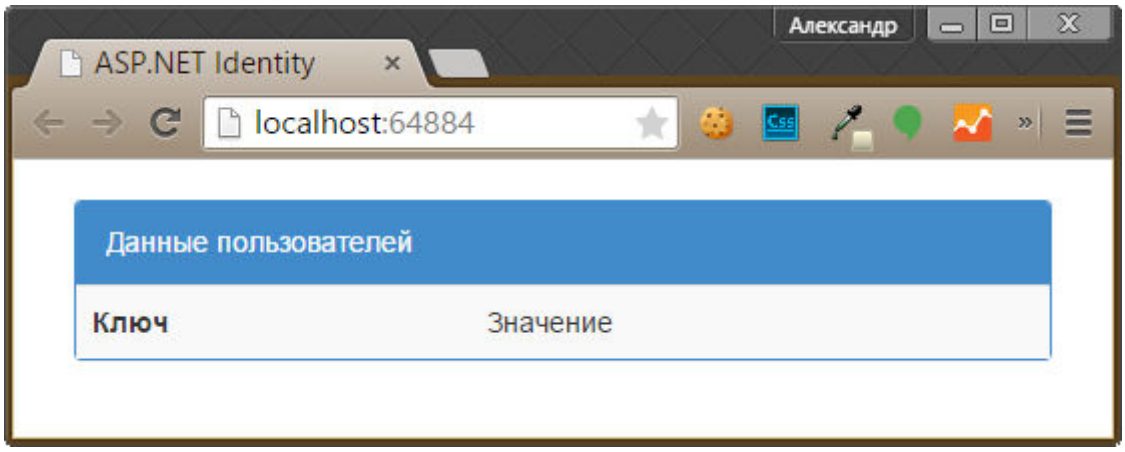
```
@using System.Collections.Generic
@model Dictionary<string, object>

@{
    ViewBag.Title = "ASP.NET Identity";
}

<div class="panel panel-primary">
    <div class="panel-heading">Данные пользователей</div>
    <table class="table table-striped">
        @foreach (string key in Model.Keys)
        {
            <tr>
                <th>@key</th>
                <td>@Model[key]</td>
            </tr>
        }
    </table>
</div>
```

Пройди тесты C# тест (легкий) .NET тест (средний)

Чтобы проверить на данном этапе работоспособность приложения, щелкните по кнопке Start Debugging (F5) в среде Visual Studio и в открывшейся вкладке вашего браузера перейдите по адресу /Home/Index (если маршрутизация по умолчанию не редактировалась (файл ~/App_Start_RouteConfig.cs), то можно запустить эту же страницу по адресу /). Результат показан на рисунке ниже:



OWIN

Для большинства разработчиков ASP.NET платформа Identity является первым знакомством с архитектурным шаблоном ***Open Web Interface for .NET (OWIN)***. OWIN - это уровень абстракции, который изолирует веб-приложения из среды, в которой они размещены. Идея заключается в том, что такая абстракция позволит добиться больших возможностей в стеке технологий ASP.NET, большей гибкости в среде разработки веб-приложений и облегченной разработки серверной инфраструктуры приложений.

OWIN - это открытый стандарт (с которым более подробно вы можете ознакомиться по ссылке owin.org). Microsoft создала проект **Katana Project**, представляющий наглядную реализацию стандарта OWIN и включающий набор компонентов, которые обеспечивают функциональность веб-приложений. В приложении OWIN/Katana Microsoft наглядно продемонстрировали изоляцию стека технологий ASP.NET от остальной платформы .NET Framework.

Используя OWIN, разработчики могут подключать только те компоненты, которые нужны прямо здесь и сейчас, а не работать с целой платформой, как это сейчас происходит с ASP.NET. Благодаря такому подходу разрабатываемое приложение не будет перегружено избыточным функционал Пройдя тестов C# тест (легкий) е .NET тест (средний) ıть быстрее (в теории).

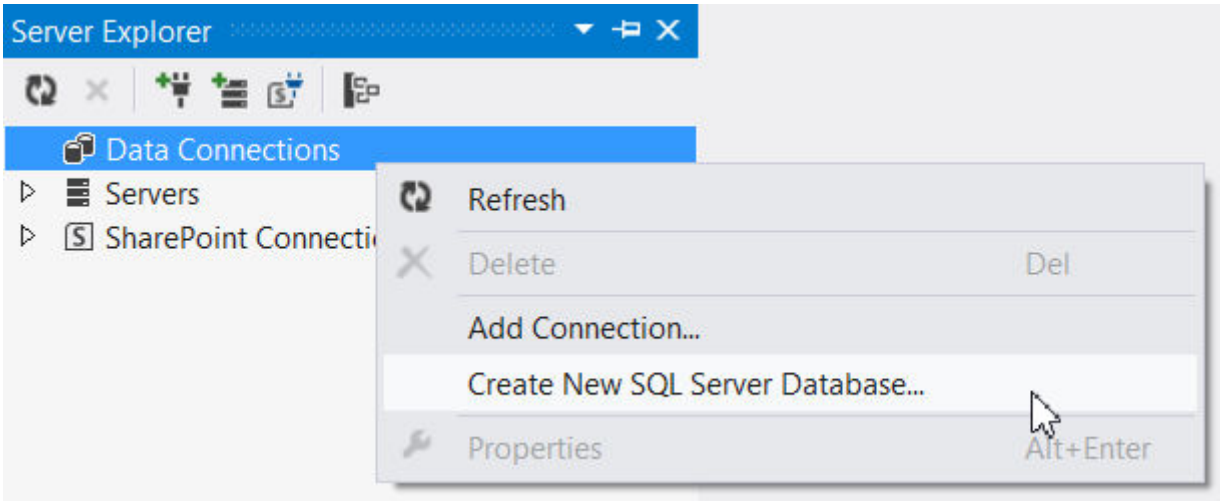
Наглядным примером компонентов, разработанных по принципу OWIN являются библиотеки Web API и SignalR, которые не требуют наличия пространства имен System.Web или работающего сервера IIS для обработки HTTP-запросов. Платформа ASP.NET MVC Framework, в отличие от этих библиотек, зависит от стандартной платформы ASP.NET.

OWIN и Katana пока не имеют серьезного влияния на MVC Framework, но появление такой платформы, как ASP.NET Identity, являющейся полноценным компонентом OWIN, говорит о том, что в скором времени такое поведение может измениться.

Создание базы данных ASP.NET Identity

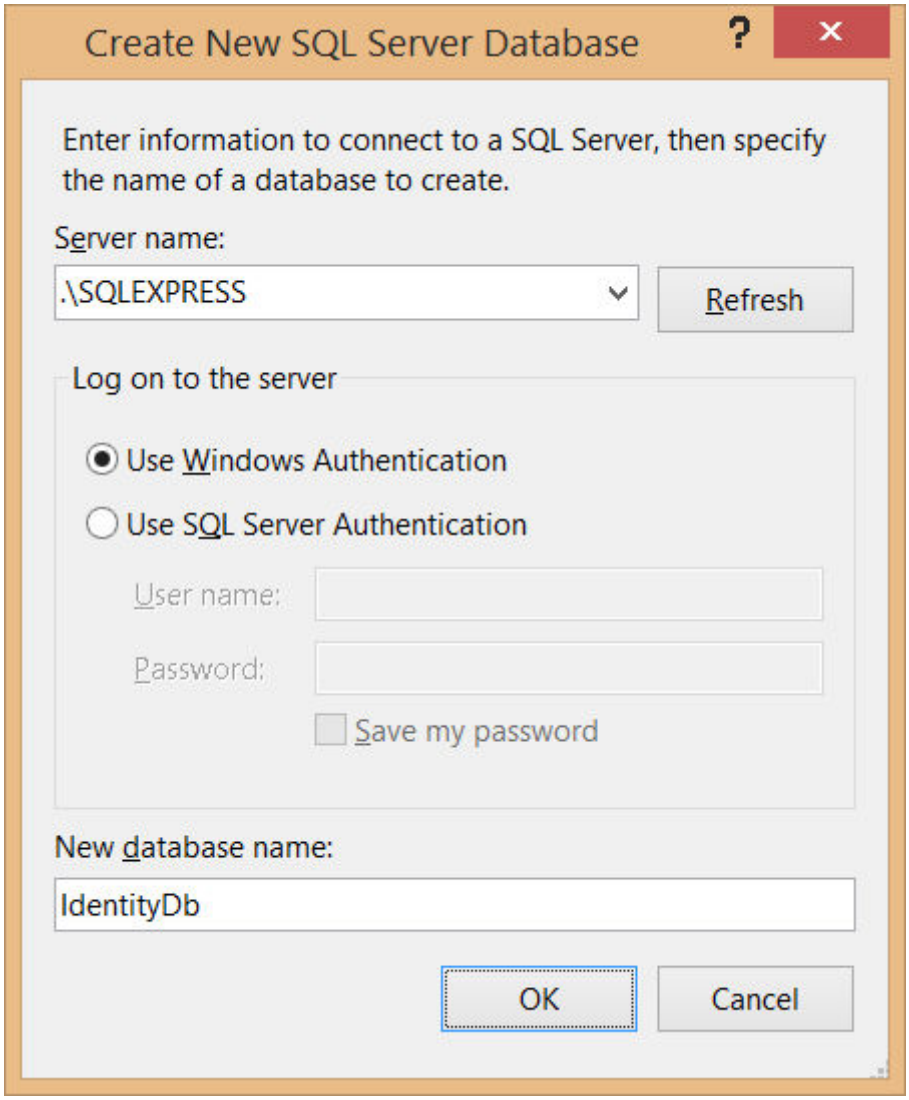
Платформа ASP.NET Identity не привязана к схеме базы данных SQL Server, в отличие от Membership API, но реляционное представление данных пользователей по-прежнему является поведением по умолчанию. Хотя в последнее время набрал обороты подход для работы с данными NoSQL, реляционные базы данных остаются основным местом хранения информации и используются в большинстве команд разработки.

ASP.NET Identity использует подход Code-First платформы Entity Framework для автоматического создания своих схем данных, но прежде необходимо вручную создать базу данных. (Вам не нужно знать как работает Entity Framework Code-First, чтобы использовать ASP.NET Identity.) Для создания базы данных откройте окно Server Explorer в среде Visual Studio ($Ctrl+W, L$). Щелкните правой кнопкой мыши по узлу Data Connections и выберите в контекстном меню команду Create New SQL Server Database, как показано на рисунке ниже:



В открывшемся диалоговом окне Create New SQL Server Database подключитесь к SQL Server Express (по умолчанию используется строка «.\SQLEXPRESS») и задайте название для базы данных IdentityDb, как показано на рисунке:

Пройди тесты C# тест (легкий) .NET тест (средний)



Когда вы щелкните по кнопке OK, Visaul Studio направит запрос к SQL Server на создание базы данных.

Добавление библиотеки ASP.NET Identity

Быстро добавить библиотеки для работы с Identity можно с помощью пакетов NuGet. Для этого введите следующие команды в панели Package Manager Console:

```
Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.0.0
Install-Package Microsoft.AspNet.Identity.OWIN -Version 2.0.0
Install-Package Microsoft.Owin.Host.SystemWeb -Version 2.1.0
```

Помимо этого, в Visual Studio есть возможность автоматически включить в проект все необходимые сборки Identity API на этапе создания проекта. Для этого, при создании приложения ASP.NET, на экране выбора шаблона проекта указывается галочка Authentication. Я не использую стандартные шаблоны проектов ASP.NET, т.к. нахожу их слишком общими и слишком многословными, а также мне нравится иметь прямой контроль над содержанием и конфигурацией моих проектов. Я рекомендую вам делать то же самое, хотя бы потому, что

Пройди тесты C# тест (легкий) / .NET тест (средний)

о, как работает ваш проект (вручную добавляя необходимые сборки, библиотеки и т. д.) Хотя иногда бывает интересно посмотреть на шаблоны, чтобы увидеть, как решаются простые задачи проектирования приложений.

Обновление файла Web.config

Необходимо произвести два изменения в файле Web.config для подготовки к работе с Identity. Во-первых, необходимо добавить строку подключения к базе данных, которую мы создали ранее. Во-вторых необходимо определить параметр, в котором передается имя класса, запускающего OWIN-приложение. В следующем примере показано содержимое файла Web.config для нашего приложения:

```
<configuration>
  ...

  <connectionStrings>
    <add name="IdentityDb" providerName="System.Data.SqlClient"
         connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=IdentityDb;Ir
  </connectionStrings>

  <appSettings>
    ...
    <add key="owin:AppStartup" value="Users.IdentityConfig" />
  </appSettings>

  ...
</configuration>
```

OWIN определяет собственную модель запуска приложения, которая не связана с глобальным классом приложения ASP.NET (класс унаследованный от HttpApplication и определенный в файле Global.asax). В примере выше мы передали параметр owin:AppStartup, который указывает класс, используемый OWIN при запуске приложения, для получения его конфигурации.

Модель классов для Entity Framework

Если вы ранее использовали Membership API в своих проектах, то вы будете удивлены насколько много требуется первоначальной подготовки перед использованием ASP.NET Identity. Гибкость настройки данных, которой не хватало Membership, теперь присутствует в Identity, однако, это приводит к тому, что вам необходимо вручную определять набор классов модели данных, которые использует Entity Framework для взаимодействия с базой данных. В следующих разделах я покажу вам как определить набор классов, необходимых для Entity Framework и влияющих на работоспособность Identity.

Класс пользователя

Пройди тесты: C# тест (легкий) у .NET тест (средний)

Первый класс который мы создадим для приложения. Этот класс должен быть унаследован от класса **IdentityUser**, который определен в *пространстве имен* *Microsoft.AspNet.Identity.EntityFramework*.

IdentityUser дает базовые представления о пользователе, которые могут быть расширены путем добавления свойств в производный класс. В таблице ниже описаны базовые свойства IdentityUser:

Свойства класса IdentityUser

Свойство	Описание
Claims	Возвращает данных с требованиями для пользователя
Email	Адрес электронной почты пользователя
Id	Уникальный идентификатор для пользователя
Logins	Коллекция логинов для пользователя
PasswordHash	Возвращает строку с хэшированным паролем пользователя
Roles	Список ролей, в которых находится пользователь
PhoneNumber	Номер телефона пользователя
SecurityStamp	Возвращает значение, которое обновляется когда были изменены любые данные пользователя (например, пароль)
UserName	Имя пользователя

Классы, определенные в пространстве имен Microsoft.AspNet.Identity.EntityFramework являются конкретными реализациями интерфейсов из пространства имен *Microsoft.AspNet.Identity*. Например, класс IdentityUser реализует *интерфейс IUser*. Я использую конкретные классы, реализованные по умолчанию, т. к. работаю с Entity Framework в качестве основы работы с Identity. Вы можете встретить другие реализации интерфейсов из пространства имен Microsoft.AspNet.Identity, ь бы взаимодействия с источником данных (не об же можете создать собственные реализации этих интерфейсов.

Пройди тесты C# тест (легкий) .NET тест (средний)

Важно отметить, что класс IdentityUser предоставляет только базовые данные о пользователе: имя, логин, почта и т. д. Если вы захотите добавить какую-нибудь дополнительную информацию о пользователе, вам необходимо будет добавить дополнительные свойства в класс, унаследованный от IdentityUser. Я покажу как это сделать позже.

Чтобы создать пользовательский класс для приложения, добавьте файл AppUserModels.cs в папку Models. В этом файле создайте класс AppUser, как показано в примере ниже:

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Models
{
    public class AppUser : IdentityUser
    {
        // Здесь будут указываться дополнительные свойства
    }
}
```

Это все, что нам нужно на данный момент, но мы вернемся к определению этого класса в одной из последующих статей, где я покажу вам, как нужно добавлять произвольные свойства для описания пользователей.

Создание класса контекста базы данных

Следующий шаг заключается в создании класса контекста базы данных для Entity Framework, который будет работать с AppUser. Это является стандартным действием при работе с подходом Code-First - вы определяете классы C# для создания и управления схемой базы данных и обеспечения доступа к данным. Класс контекста должен быть унаследован от **IdentityDbContext<T>**, где T - это класс, описывающий пользователя (в нашем случае AppUser). Создайте папку Infrastructure в проекте и добавьте файл класса с названием AppIdentityDbContext.cs, код которого показан в примере ниже:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
using Users.Models;
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Infrastructure
{
    public class AppIdentityDbContext : IdentityDbContext<AppUser>
    {
        public AppIdentityDbContext() : base("name=IdentityDb") { }

        static AppIdentityDbContext()
        {
            Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
        }

        public static AppIdentityDbContext Create()
        {
            return new AppIdentityDbContext();
        }
    }

    public class IdentityDbInit : DropCreateDatabaseIfModelChanges<AppIdentityDbC
    {
        protected override void Seed(AppIdentityDbContext context)
        {
            PerformInitialSetup(context);
            base.Seed(context);
        }
        public void PerformInitialSetup(AppIdentityDbContext context)
        {
            // настройки конфигурации контекста будут указываться здесь
        }
    }
}
```

Конструктор класса AppIdentityDbContext в примере вызывает базовый конструктор, передавая название строки подключения к базе данных, которую мы добавили ранее в файл Web.config.

Класс AppIdentityDbContext определяет также статический конструктор в котором вызывается *метод Database.SetInitializer()*, [illegible] а, связывающего схему базы данных с классами [illegible]. В примере выше таким классом является IdentityDbInit, в котором мы переопределили метод базового класса Seed(), добавив в него вызов

Пройди тесты C# тест (легкий) .NET тест (средний)

вспомогательного метода PerformInitialSetup(). В этот метод в дальнейшем мы будем добавлять инструкции для работы с базой данных.

Наконец, класс AppIdentityDbContext определяет статический метод Create(), с помощью которого будут создаваться экземпляры этого класса.

Не волнуйтесь, если структура приведенных блоков кода для работы с Entity Framework кажется вам незнакомой — вы можете рассматривать ее как «черный ящик» - после того как мы определим все строительные блоки для работы с Entity Framework, вы можете просто копировать их в свои проекты.

Создание класса управления пользователями

Одним из наиболее важных классов платформы Identity является класс управления пользователями, который должен быть унаследован от **UserManager<T>**, где T это класс, описывающий пользователя. Класс UserManager<T> не является частью Entity Framework, он описывает более общие особенности создания и функционирования данных пользователя. В таблице ниже приведены основные методы и свойства класса UserManager<T>. Есть несколько других членов данного класса, не указанных в таблице. Я опишу их позже, в контексте примеров, когда они будут необходимы.

Пройди тесты C# тест (легкий) .NET тест (средний)

Основные методы и свойства класса `UserManager<T>`

Название	Описание
ChangePasswordAsync(id, old, new)	Изменяет пароль для указанного пользователя
CreateAsync(user)	Создает нового пользователя без пароля
CreateAsync(user, pass)	Перегруженная версия предыдущего метода для создания пользователя с паролем
DeleteAsync(user)	Удаляет указанного пользователя
FindAsync(user, pass)	Находит объект, представляющий пользователя и проверяет подлинность пароля
FindByIdAsync(id)	Поиск пользователя по его идентификатору `Id`
FindByNameAsync(name)	Поиск пользователя по имени
UpdateAsync(user)	Сохраняет изменения данных пользователя в базе данных
Users	Свойство, возвращающее список всех пользователей

Обратите внимание, что имена всех приведенных методов на конце содержат суффикс `Async`. Платформа `ASP.NET Identity` почти полностью реализована на асинхронных методах `C#`, благодаря которым не блокируется основной поток на время выполнения метода, освобождая место другим операциям в пуле потоков `CLR`. Вы увидите, как это работает, как только я продемонстрирую как создавать и управлять данными пользователей. Все перечисленные выше методы имеют синхронные аналоги (без суффикса `Async` на конце). Я придерживаюсь работы с асинхронными методами в своих примерах, Пройди тесты. C# тест (легкий) .NET тест (средний) ы, если необходимо выполнить несколько операций в определенной последовательности.

Итак, добавьте файл класса AppUserManager.cs в папку Infrastructure вашего проекта со следующим содержимым:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Users.Models;

namespace Users.Infrastructure
{
    public class AppUserManager : UserManager<AppUser>
    {
        public AppUserManager(IUserStore<AppUser> store)
            : base(store)
        { }

        public static AppUserManager Create(IdentityFactoryOptions<AppUserManager
            IOwinContext context)
        {
            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(new UserStore<AppUser>(db
            return manager;
        }
    }
}
```

Статический метод Create() вызывается, когда Identity нуждается в экземпляре класса AppUserManager для работы с данными пользователей.

Чтобы создать экземпляр класса AppUserManager, нам нужен объект UserStore<AppUser>. **Класс UserStore<T>** реализует интерфейс *IUserStore<T>* из Entity Framework, который описывает CRUD-методы (create/read/update/delete) для работы с хранилищем данных (в нашем случае с базой данных). Для создания UserStore<AppUser>, мне нужен экземпляр класса AppIdentityDbContext, который я получаю через OWIN следующим образом:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
// ...
AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
// ...
```

Реализация *IOwinContext* передается в качестве параметра метода Create и определяет обобщенный метод Get(), который возвращает экземпляры объектов, которые были зарегистрированы в классе запуска OWIN, который я опишу в следующем разделе.

Создание класса запуска OWIN

Последнее, что нужно для базовой конфигурации Identity — создать класс запуска OWIN. Ранее, в файле Web.config мы определили параметр приложения, указывающий на название класса запуска OWIN:

```
...
<add key="owin:AppStartup" value="Users.IdentityConfig" />
...
```

Напомню, что стандарт OWIN развивался отдельно от ASP.NET. Он гласит, что в приложении должен существовать класс, который загружает и настраивает данные для *промежуточного (middleware) слоя* платформы, реализующей OWIN и выполняет любые другие работы по настройке, которые необходимы. По умолчанию этот класс должен называться Start и располагаться в глобальном пространстве имен. Этот класс должен содержать метод, называемый Configuration, который вызывается инфраструктурой OWIN и принимает параметр типа **Owin.IAppBuilder**, который поддерживает создание промежуточного слоя для приложения.

Я буду игнорировать базовые соглашения по определению класса запуска OWIN, т.к. в нашем приложении MVC единственным промежуточным слоем является Identity. Для этого мы и указали настройку owin:AppStartup в файле Web.config, указывающую на определение класса запуска приложения в пространстве имен верхнего уровня приложения. Добавьте файл класса IdentityConfig.cs в папку App_start со следующим содержимым:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;

namespace Users
{
    public class IdentityConfig
    {
        public void Configuration(IAppBuilder app)
        {
            app.CreatePerOwinContext<AppIdentityDbContext>(AppIdentityDbContext.C
            app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);

            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie
                LoginPath = new PathString("/Account/Login"),
            });
        }
    }
}
```

Интерфейс `IAppBuilder` содержит несколько методов расширения, определенных в классах в пространстве имен `Owin`. (Напомню, методы расширения `C#` вызываются на экземплярах класса, а их реализация содержится в других классах.) *Метод* *`CreatePerOwinContext`* создает новые экземпляры классов `AppUserManager` и `AppIdentityDbContext` для каждого запроса. Это гарантирует, что каждый запрос будет отдельно работать с данными `ASP.NET Identity` и что не придется беспокоиться о синхронизации или настройке кэширования данных.

Метод *`UseCookieAuthentication`* говорит платформе `Identity` о том, что нужно использовать куки для авторизации пользователей, а параметры передаются через объект **`CookieAuthenticationOptions`**. Самой главной настройкой здесь является задание *свойства* *`LoginPath`*, указывающего на `URL` куда будет перенаправляться неаутентифицированный пользователь, при запросе контента, требующего авторизации. Я добавил `URL` `Account/Login`, для которого контроллер и представление мы создадим в одной из следующих статей.

Пройди тесты **`C# тест (легкий)`** **`.NET тест (средний)`**