**КОМПЬЮТЕРНАЯ АКАДЕМИЯ «ШАГ»**

Секция: Разработка программного обеспечения

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

К ДИПЛОМНОМУ ПРОЕКТУ ПО ТЕМЕ:

Разработка web-приложения для владельцев собак

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Development of a web application for dog owners

СЛУШАТЕЛЬ 28ПР31 Терешкин Никита Сергеевич

Mykyta Tereshkin

РУКОВОДИТЕЛЬ ПРОЕКТА Задерей Юрий Николаевич

ДИРЕКТОР ХФ КА «ШАГ» Канд. техн. наук, доцент Макушенко Т.В.

Харьков 2019 г.

**КОМПЬЮТЕРНАЯ АКАДЕМИЯ «ШАГ»**

Секция: Разработка программного обеспечения

УТВЕРЖДАЮ

Зав. учебной частью

Остапова Т.В.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2019 г.

**ЗАДАНИЕ**

НА ДИПЛОМНЫЙ ПРОЕКТ СЛУШАТЕЛЮ

Терешкину Никите Сергеевичу

1. Тема проекта:

2. Срок сдачи слушателем законченного проекта:

3. Исходные данные к проекту:

4. Содержание пояснительной записки (перечень вопросов, подлежащих  разработке):

5. Дата выдачи задания:

Руководитель

Задание принял к выполнению

СОДЕРЖАНИЕ

АННОТАЦИЯ 4

ВВЕДЕНИЕ 5

ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 6

1. Выбор технологии реализации проекта 7

2. Разработка структуры системы 8

3. Разработка алгоритмов функционирования системы 14

4. Разработка базы данных для системы 22

5. Разработка интерфейса системы 24

6. Руководство пользователя 29

ВЫВОДЫ 32

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 33

ПРИЛОЖЕНИЯ 34

АННОТАЦИЯ

Целью данной работы являлась разработка web приложения для владельцев собак, где они могут обмениваться информацией и находить единомышленников.

Актуальность данного вопроса заключается практически в полном отсутствии правильно функционирующих вебсайтов подобной направленности.

Данное приложение ориентированно на широкую аудиторию: любителей четырехлапых друзей человека, владельцев и тех, кто собирается завести питомца.

ВВЕДЕНИЕ

Человек приручил собаку много тысяч лет назад. И по сегодняшний день находится с ними в тесной связи. Те, кому посчастливилось быть хозяином для одного такого животного, понимают насколько это иногда тяжело.

В настоящее время существует несколько сайтов, предназначенных для любителей собак, такие как: pet2you.com, petup.ru и другие. Но они не предоставляют удобный функционал для коммуникации владельцев между собой и поиска пары для питомца.

На сегодняшний день, интернет есть почти везде. Выбор в пользу веб-приложений, а не обычного приложения под определенную операционную систему, обусловлен возможностью открыть его и получить доступ ко всем функциям не зависимо от устройства. Будь то смартфон, планшетный ПК, настольный ПК с любой операционной системой – всегда можно зайти на вебсайт.

ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Главной целью данной работы являлось создание web-приложения «Petoo», предоставляющий список функций для пользователей, а именно:

* возможность зарегистрироваться на сайте, а после этого пользоваться своей учетной записью;
* возможность добавить питомца в список своих питомцев и указать данные, характеризующие его;
* наличие функционала, предназначенного для общения пользователей между собой;
* наличие функционала для поиска пары для своего питомца;

Для выполнения поставленных задач и целей будет создано web-приложение «Petoo» средствами языка программирования C# с использованием ASP.NET MVC 5, фреймворка Entity Framework, СУБД Microsoft SQL Server.

1. Выбор технологии реализации проекта

В настоящее время создание вебсайтов возможно, в основном, на языках программирования PHP, Ruby on Rails и C#. Использование последнего предполагает работу с ASP.NET.

ASP.NET – это платформа разработки [веб-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5), в состав которой входит: [веб-сервисы](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81), программная инфраструктура, модель программирования, от компании [Майкрософт](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82). ASP.NET входит в состав платформы [.NET Framework](https://ru.wikipedia.org/wiki/Microsoft_.NET) и является развитием более старой технологии [Microsoft ASP](https://ru.wikipedia.org/wiki/Active_Server_Pages).

ASP.NET внешне во многом сохраняет схожесть с более старой технологией ASP, что позволяет разработчикам относительно легко перейти на ASP.NET. В то же время внутреннее устройство ASP.NET существенно отличается от ASP, поскольку она основана на платформе [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET) и, следовательно, использует все новые возможности, предоставляемые этой платформой.

В отличии от других технологий, ASP.NET может похвастаться такими преимуществами:

* Компилируемый код выполняется быстрее, большинство ошибок отлавливается ещё на стадии разработки;
* Значительно улучшенная обработка ошибок во время выполнения запущенной готовой программы, с использованием блоков try..catch;
* Пользовательские элементы управления (controls) позволяют выделять часто используемые шаблоны, такие как меню сайта;
* Использование метафор, уже применяющихся в Windows-приложениях, например, таких как элементы управления и события;
* Расширяемый набор элементов управления и библиотек классов позволяет быстрее разрабатывать приложения;
* ASP.NET опирается на многоязыковые возможности [.NET](https://ru.wikipedia.org/wiki/.NET_Framework), что позволяет писать код страниц на [VB.NET](https://ru.wikipedia.org/wiki/Visual_Basic_.NET), [Delphi.NET](https://ru.wikipedia.org/wiki/Delphi_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), [Visual C#](https://ru.wikipedia.org/wiki/C_Sharp), [J#](https://ru.wikipedia.org/wiki/J) и т. д.;
* Возможность кэширования всей страницы или её части для увеличения производительности;
* Возможность кэширования данных, используемых на странице;
* Возможность разделения визуальной части и бизнес-логики по разным файлам («code behind»);
* Расширяемая модель обработки запросов;
* Расширенная событийная модель;
* Расширяемая модель серверных элементов управления;
* Наличие master-страниц для задания шаблонов оформления страниц;
* Поддержка [CRUD](https://ru.wikipedia.org/wiki/CRUD)-операций при работе с таблицами через GridView;
* Встроенная поддержка [AJAX](https://ru.wikipedia.org/wiki/AJAX);

Microsoft SQL Server — [система управления реляционными базами данных (РСУБД)](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94), разработанная корпорацией [Microsoft](https://ru.wikipedia.org/wiki/Microsoft). Основной используемый язык запросов — [Transact-SQL](https://ru.wikipedia.org/wiki/Transact-SQL), создан совместно Microsoft и [Sybase](https://ru.wikipedia.org/wiki/Sybase). Transact-SQL является реализацией стандарта [ANSI](https://ru.wikipedia.org/wiki/%D0%90%D0%BC%D0%B5%D1%80%D0%B8%D0%BA%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BD%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BE%D0%B2)/[ISO](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%B4%D1%83%D0%BD%D0%B0%D1%80%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8) по структурированному языку запросов ([SQL](https://ru.wikipedia.org/wiki/SQL)) с расширениями. Используется для работы с базами данных размером от персональных до крупных баз данных масштаба предприятия; конкурирует с другими СУБД в этом сегменте рынка.

Доступ к базе данных осуществляется при помощи технологии Entity Framework. Это объектно-ориентированная технология доступа к данным, является [object-relational mapping](https://ru.wikipedia.org/wiki/ORM) (ORM) решением для [.NET Framework](https://ru.wikipedia.org/wiki/.NET_Framework) от [Microsoft](https://ru.wikipedia.org/wiki/Microsoft). Предоставляет возможность взаимодействия с объектами как посредством [LINQ](https://ru.wikipedia.org/wiki/LINQ) в виде LINQ to Entities, так и с использованием Entity SQL. Для облегчения построения web-решений используется как [ADO.NET Data Services](https://ru.wikipedia.org/wiki/ADO.NET_Data_Services) ([Astoria](https://ru.wikipedia.org/wiki/ADO.NET_Data_Services)), так и связка из [Windows Communication Foundation](https://ru.wikipedia.org/wiki/Windows_Communication_Foundation) и [Windows Presentation Foundation](https://ru.wikipedia.org/wiki/Windows_Presentation_Foundation), позволяющая строить многоуровневые приложения, реализуя один из шаблонов проектирования [MVC](https://ru.wikipedia.org/wiki/Model-View-Controller), [MVP](https://ru.wikipedia.org/wiki/Model-View-Presenter) или [MVVM](https://ru.wikipedia.org/wiki/Model-View-ViewModel).

2. Разработка структуры системы

Для реализации данного приложения была предложена следующая структура.

Технология ASP.NET MVC 5 предполагает разбиение проекта на три главные части: Models, Views, Controllers (Рис. 2.1). Контроллер обрабатывает запросы пользователя и выдает соответствующее представление, иногда с данными из модели.

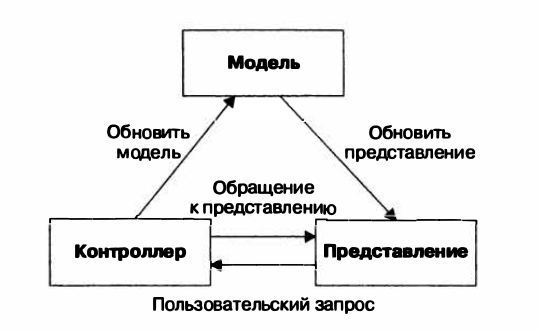


Рис. 2.1 Схема работы MVC

Для хранения данных были добавлены следующие модели:

* User
* Login
* RegistrationModel
* Role
* Pet
* Breed

Рассмотрим детальнее модель User (Приложение А).

В нем помимо полей для хранения самих данных пользователя, например, имя, фамилия и электронный адрес, добавлено поле Id для обеспечения уникальности записи в базе данных. Кроме этого, добавлены по два поля для связи с моделями Login и Role. Такая структура гарантирует, что Entity Framework распознает то, что эти таблицы связаны между собой. Коллекция Pets говорит нам о том, что у одного пользователя может быть неограниченное количество питомцев и все они будут закреплены в базе данных за этим пользователем.

Модель Login, в свою очередь, содержит всего три поля (Листинг 2.2).

Первое, как и в случае с User моделью, содержит уникальный номер, который присваивается автоматически. Для этого добавлены так называемые аннотации. Содержатся они в пространстве имен System.ComponentModel.DataAnnotations. С их помощью мы показываем, что первое поле является ключом и что он генерируется базой данных автоматически по принципу Idenity.

Остальные поля почти одинаковы, если смотреть со стороны принципа работы. Это текстовые значения, которые являются обязательными при создании. Об этом нам говорит аннотация Required над каждым из полей. Помимо этого, написаны аннотации, которые показывают, как при автоматическом создании представлений будут подписаны эти поля. К примеру, само поле у нас называется «UserLogin», но в представлении мы увидим надпись «Login», что упростит понимание для пользователя.

public class Login

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

[Display(Name = "Login")]

public string UserLogin { get; set; }

[Required]

[Display(Name = "Password")]

public string Password { get; set; }

}

Листинг 2.2. Модель Login

В моделе Role (Листинг 2.3) нету ничего, что бы мы уже не рассмотрели на этом этапе. Только в этот раз мы не указывали явно аннотацией что поле Id является ключом. Entity Framework имеет свойство автоматически распознавать его, если в названии поля содержится часть «Id». Такое решение делает код менее загроможденным.

public class Role

{

public int Id { get; set; }

[Display(Name = "Role")]

public string Name { get; set; }

public virtual ICollection<User> Users { get; set; }

public Role()

{

Users = new List<User>();

}

}

Листинг 2.3 Модель Role

Следующая модель – RegistrationModel (Приложение Б). Эта модель будет использоваться при регистрации нового пользователя и будет передаваться в контроллер для создания других моделей типа User и Login. В ней содержатся такие поля, как имя, фамилия, электронный адрес, логин, два поля для пароля (второе используется для подтверждения, дабы пользователь не ввел такой пароль, который потом не смог бы воспроизвести). Так как данные типа RegistrationModel не будут храниться в базе данных, мы не создаем поле с ключом и не делаем различные связи с другими таблицами. Для паролей с помощью аннотаций задаем тип «пароль». Это поможет при автоматическом создании представления проекту понять, что для этих полей необходимо создать input с type=”password”. Также, для второго поля пароля добавляем аннотацию, которая сравнивает значения этого поля со значением первого поля пароля. Теперь это происходит автоматически, и пользователь даже не сможет отправить данные с формы на представлении в контроллер, пока, конечно, два этих поля не будут совпадать.

Модели Pet (Листинг 2.4) и Breed (Листинг 2.5) взаимосвязаны. Так, каждый питомец имеет одну породу, а одна порода имеет много питомцев. Такая связь называется «один ко многим». Но стоит заметить что питомец еще имеет связь с пользователем, который является его хозяином.

public class Pet

{

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public bool Gender { get; set; }

[Required]

public int Age { get; set; }

[ForeignKey("Breed")]

public int BreedId { get; set; }

public Breed Breed { get; set; }

[ForeignKey("User")]

public int UserId { get; set; }

public User User { get; set; }

}

Листинг 2.4. Модель Pet

public class Breed

{

public int BreedId { get; set; }

[Required]

[Display(Name = "Breed")]

public string Name { get; set; }

public virtual ICollection<Pet> Pets { get; set; }

public Breed()

{

Pets = new List<Pet>();

}

}

Листинг 2.5. Модель Breed

Для того, чтобы Entity Framework распознал наши модели, необходим так называемый контекст. В нашем проекте он называется «AccountContext» (Листинг 2.6). В нем мы добавляем поля типа DbSet< > в которых будут содержаться объекты типа наших моделей. Помимо этого, мы еще передаем в конструктор базового класса строку, содержащую название нашей базы данных.

public class AccountContext : DbContext

{

public AccountContext() : base("PetooDb")

{ }

public DbSet<User> Users { get; set; }

public DbSet<Login> Logins { get; set; }

public DbSet<Role> Roles { get; set; }

public DbSet<Breed> Breeds { get; set; }

public DbSet<Pet> Pets { get; set; }

}

Листинг 2.6. Класс AccountContext

Теперь у нас будет создаваться пустая база данных при запуске приложения. Но исходных данных в ней не будет никаких. По этому, нам необходимо хотя бы создавать роли, которые потом будут присваиваться пользователям, и учетную запись администратора. Для этого реализуем класс «DbInitializer» (Приложение В). Нам необходимо наследовать его от класса «CreateDatabaseIfNotExists< >» типизированного под наш контекст. Но на этапе тестирования нам будет более удобно, если база данных будет создаваться и заполняться исходными значениями. По этому наследуем от «DropCreateDatabaseAlways< >». Внутри самого класса нам необходимо переопределить всего лишь один метод «Seed». Внутрь него передается экземпляр класса AccountContext. Дальше, в самом методе, нам придется создать экземпляры моделей, которые мы хотим видеть в нашей базе данных изначально. После того как они созданы, добавляем их в DbSet c соответствующим названием, содержащийся в экземпляре контекста. Как только все необходимые объекты добавлены, мы просто вызываем метод Seed базового класса, куда передаем наш контекст.

Таким образом получаем готовый DbInitializer, который запишет в базу данных изначальные значения. Только, очень важно помнить, что необходимо прописать его в файле Global.asax в методе Application\_Start (Листинг 2.7).

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

Database.SetInitializer(new PetooDbInitializer());

}

Листинг 2.7. Метод Application\_Start

3. Разработка алгоритмов функционирования системы

Переходим к контроллерам, которые и реализуют всю логику работы приложения.

Для управления авторизацией в нашем проекте был создан AccountController, в нем реализованы разные методы, такие как Login, Logout и Register. Рассмотрим принцип работы метода Login (Листинг 3.1).

В качестве аргументов в него приходят экземпляр класса Login и строка. Приходят данные методом POST.

В программировании POST — один из многих методов запроса, поддерживаемых [HTTP](https://ru.wikipedia.org/wiki/HTTP) протоколом, используемым во Всемирной паутине. Метод запроса POST предназначен для запроса, при котором [веб-сервер](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80) принимает данные, заключённые в тело сообщения, для хранения. Он часто используется для загрузки файла или представления заполненной [веб-формы](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%84%D0%BE%D1%80%D0%BC%D0%B0).

В отличие от него, метод HTTP GET предназначен для получения информации от сервера. В рамках GET-запроса некоторые данные могут быть переданы в строке запроса URI, указывающие, например, условия поиска, диапазоны дат, или другую информацию, определяющую запрос. В рамках POST запроса произвольное количество данных любого типа может быть отправлено на сервер в теле сообщения запроса. Поля заголовка в POST-запросе обычно указывают на [тип содержимого](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_MIME-%D1%82%D0%B8%D0%BF%D0%BE%D0%B2).

Таким образом, внутри метода мы проверяем, есть ли пользователем с логином и паролем, полученными в качестве аргументов и, если есть, делаем переход на другой контроллер, под названием Home.

Для проверки, валиден ли пользователь, был реализован CustomMembershipProvider (Приложение Г). Использование метода ValidateUser даст понимание, есть ли пользователь в базе данных с такими логином, паролем и ключом.

[HttpPost]

public ActionResult Login(Login model, string returnUrl)

{

if (ModelState.IsValid)

{

if (Membership.ValidateUser(model.UserLogin, model.Password))

{

FormsAuthentication.SetAuthCookie(model.UserLogin, false);

if (Url.IsLocalUrl(returnUrl))

{

return Redirect(returnUrl);

}

else

{

var currentUser = GetUser(model.UserLogin);

var user = db.Users.Where(x => x.Login.UserLogin == model.UserLogin).FirstOrDefault();

return RedirectToAction("Index", "Home", new { userId = user.Id });

}

}

else

{

ModelState.AddModelError("", "Wrong login or password");

}

}

return View(model);

}

Листинг 3.1. Метод Login

Метод Register (Листинг 3.2) работает со схожим принципом, только нам требуется проверить, не существует ли уже какого-нибудь пользователя с таким же логином. Если нет, добавляем в БД и переходим дальше на Home. Данные передаются с помощью класса RegistrationModel и метода POST.

Отображение ошибок реализовано с помощью ModelState.

Выбор в пользу создания отдельной модели RegistrationModel, которая даже не хранится в базе данных, обусловлен тем, что можно удобно средствами Visual Studio создать представление, которое уже будет настроено так как нам нужно, отталкиваясь от модели и аннотаций над полями.

[HttpPost]

public ActionResult Register(RegistrationModel regModel)

{

try

{

foreach (var login in db.Logins)

{

if (login.UserLogin == regModel.Login)

{

ModelState.AddModelError("", "This nickname was already taken");

return View();

}

}

var log = new Login() { UserLogin = regModel.Login, Password = regModel.Password };

db.Logins.Add(log);

var user = new User() { Login = log, Email = regModel.Email, FirstName = regModel.FirstName, LastName = regModel.LastName, RoleId = 2 };

db.Users.Add(user);

db.SaveChanges();

return RedirectToAction("Index", "Home", new { userId = user.Id });

}

catch (DbEntityValidationException e)

{

ViewBag.Error = "Something went wrong!";

}

return View();

}

Листинг 3.2. Метод Register

Что касается HomeController, пользователь получает доступ к его методам только после авторизации. Доступ неавторизированным пользователям ограничен с помощью аннотаций «[Authorize]» над методами, которые возвращают представления. В них и будет основная логика приложения.

4. Разработка базы данных для системы

База данных приложения реализована с использованием Microsoft SQL Server и Entity Framework Code First. Последнее означает, что мы не занимаемся созданием скриптов, которые бы добавляли или обеспечивали работу с таблицами. За нас это делает EF. Все необходимое для его работы мы сделали при создании моделей.

На данном этапе, диаграмма таблиц (Рис. 4.1), обеспечивающих работу с пользователями выглядит как дерево, вершиной которого является таблица пользователей, а от них уже идут «ветви» на питомцев, ролей и так далее.

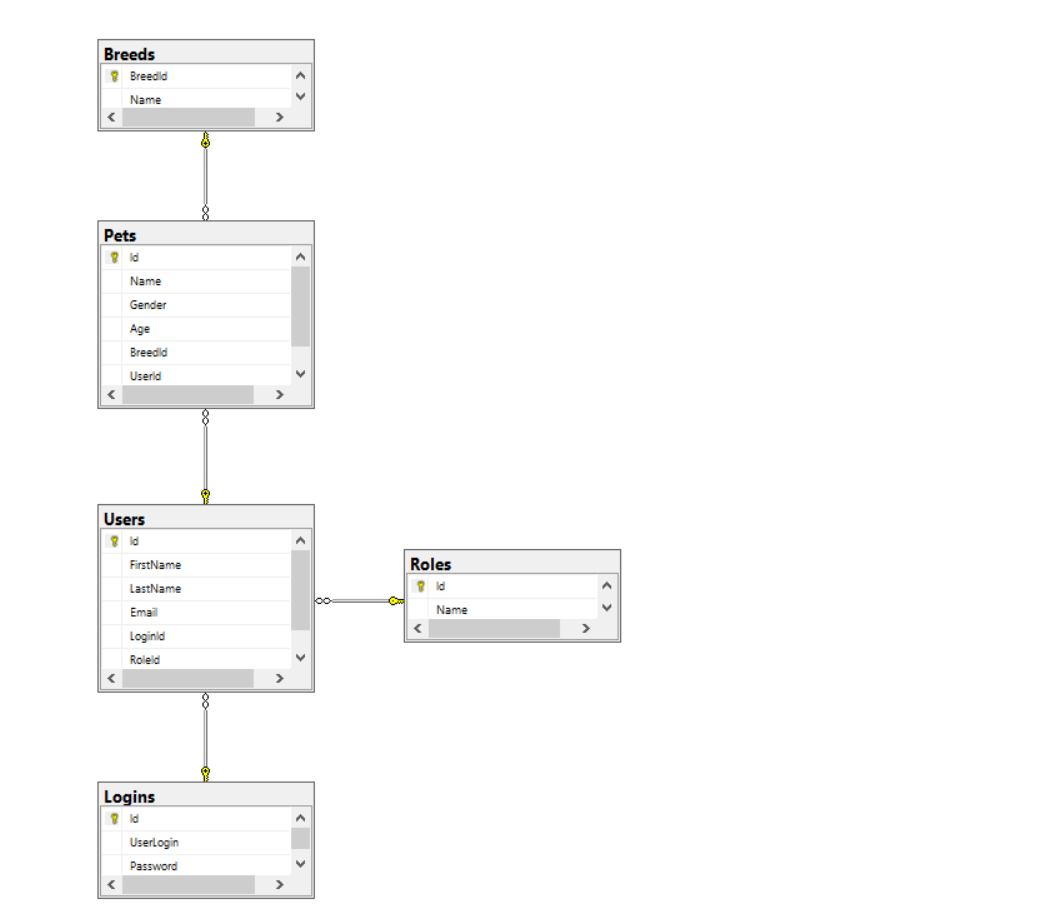


Рис. 4.1. Диаграмма таблиц

5. Разработка интерфейса системы

Приложение ASP.NET подразумевает, что интерфейс системы будет сгенерирован при помощи так называемых представлений. Это могут быть файлы с расширениями .cshtml, .vbhtml или .aspx. Какое именно расширение будет использоваться зависит от выбранного вами «движка» создания этих представлений. В нашем проекте выбор упал в сторону движка под названием Razor.

При вызове метода View контроллер не производит рендеринг представления и не генерирует разметку html. Контроллер только готовит данные и выбирает, какое представление надо возвратить в качестве объекта ViewResult. Затем уже объект ViewResult обращается к движку представления для рендеринга представления в выходной результат.

До MVC 3 использовался только движок **Web Forms**, который использовал стандартный синтаксис ASP.NET.

Введение в MVC 3 движка Razor позволило уменьшить синтаксис при вызове кода C#, сделать код более "чистым". Поэтому его использование более предпочтительно. В конечном счете, выбор движка представлений зависит от личных предпочтений, и оба движка по функциональности равноценны.

В данном случае важно понимать, что движок представлений, например, Razor - это не какой-то новый язык, это лишь способ рендеринга представлений, который имеет определенный синтаксис для перехода от разметки html к коду C#.

Кроме движка Razor можно использовать и движки сторонних производителей. Например, движок Spark, использующий декларативный синтаксис, или Brail, который использует язык Boo.

Использование синтаксиса Razor характеризуется тем, что перед выражением кода стоит знак @, после которого осуществляется переход к коду C#. Существуют два типа переходов: к выражениям кода и к блоку кода.

Для авторизации были созданы такие представления, как представление для входа в существующий аккаунт (Рис. 5.1) и представление для регистрации (Рис. 5.2).

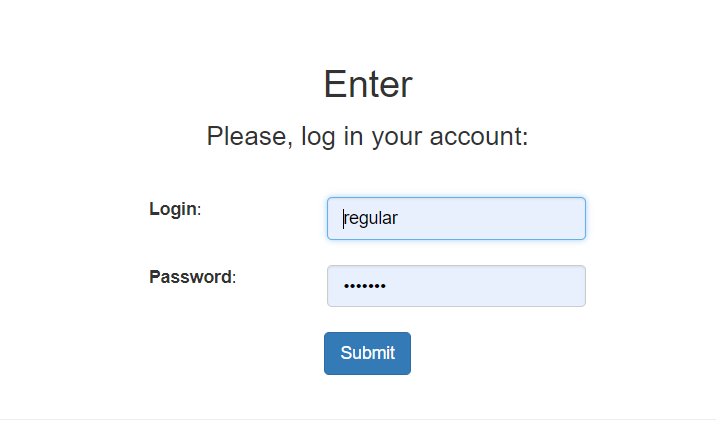


Рис. 5.1. Представление для входа в аккаунт

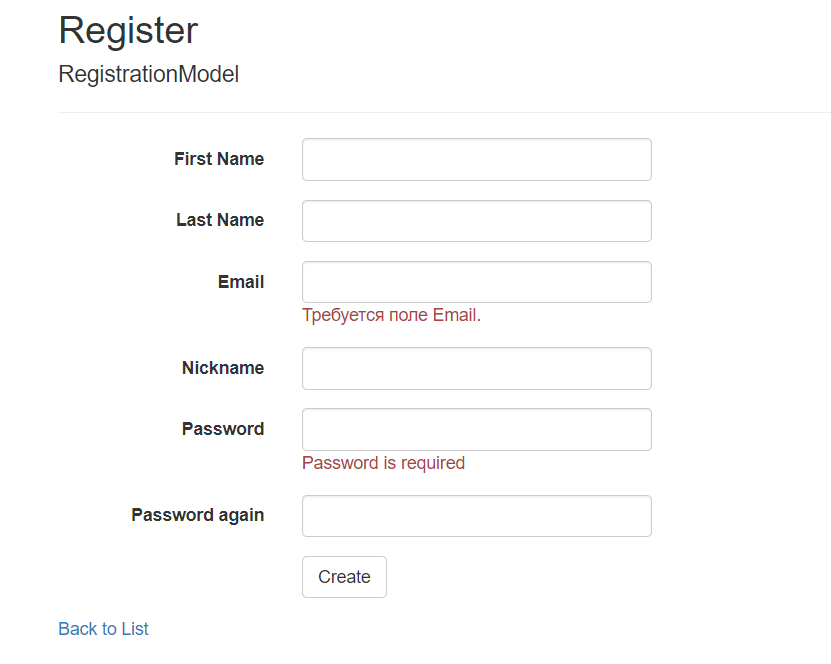


Рис. 5.2. Представление регистрации

По этому же принципу создаются остальные представления.

Для создания единообразного вида сайта применяются мастер-страницы. Мастер-страницы - это по сути те же самые представления. Мы можем определить на мастер-странице некоторые элементы, которые будут отображаться на всех страницах сайта. А также мы можем определить заполнители или плейсхолдеры, содержание которых обеспечивают другие представления.

На вид это обычное представление за одним исключением - вызова метода @RenderBody(). Этот вызов является плейсхолдером, на место которого другие представления, которые используют эту мастер-страницу, будут подставлять свое содержимое. И таким образом, мы можем легко установить для представлений веб-приложения единообразный стиль.

6. Руководство пользователя

Для того чтобы воспользоваться вебсайтом, необходимо перейти по ссылке и попасть на страницу авторизации. Если у вас есть ваша учетная запись, введите логин и пароль. Если они верны, вы попадете на главную страницу, откуда уже можно воспользоваться основным функционалом. Если же у вас нет учетной записи, перейдите на страницу регистрации, введите необходимые данные и нажмите кнопку подтвердить внизу.

На главной странице есть ссылка на страницу с коллекцией ваших питомцев. Оттуда можно добавить, редактировать или удалить питомца.

ВЫВОДЫ

В ходе проделанной работы был разработан вебсайт «Petoo», который является местом коммуницирования между собой для владельцев собак.

Данное приложение реализует весь заявленный функционал, а именно возможность зарегистрироваться и войти в свою учетную запись, добавить своих питомцев или изменить их данные, функционал для общения и поиска пары питомцу.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. <https://habr.com/>
2. <https://stackoverflow.com/>
3. Dominick, Baier Developing More–Secure Microsoft ASP.NET 2.0 Applications / Dominick Baier. - Москва: **ИЛ**, 2006. - 480 c.
4. Eric, A. Smith Active Server® Pages Bible / Eric A. Smith. - Москва: **Машиностроение**, **2003**. - **714** c.
5. Esposito Programming Microsoft® ASP.NET 4 / Esposito. - Москва: **Высшая школа**, 2011. - **152** c.
6. Аньен, Ф. Основы ASP.NET с примерами на Visual Basic .NET / Ф. Аньен. - Москва: **Гостехиздат**, 2005. - 400 c.
7. Венц, Кристиан Программирование в ASP.NET Ajax / Кристиан Венц. - М.: Символ-плюс, 2008. - 512 c.
8. Гаряка, А. А. Основы ASP.NET 2.0 / А.А. Гаряка. - М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2007. - 296 c.
9. Дари, Кристиан Поисковая оптимизация на ASP.NET для профессионалов. Руководство разработчика по SEO / Кристиан Дари , Джейми Сирович. - М.: Диалектика, 2008. - 400 c.
10. Камерон, Роб ASP.NET 3.5, компоненты AJAX и серверные элементы управления для профессионалов / Роб Камерон , Дэйл Михалк. - М.: Вильямс, 2009. - 608 c.
11. Мак-Дональд, Мэтью Microsoft ASP.NET 2.0 с примерами на C# 2005 для профессионалов / Мэтью Мак-Дональд , Марио Шпушта. - М.: Вильямс, 2007. - **192** c.
12. Рихтер, Джеффри CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# / Джеффри Рихтер. - М.: Питер, 2016. - 896 c.
13. Столбовский, Д. Н. Основы разработки Web-приложений на ASP.NET / Д.Н. Столбовский. - М.: Бином. Лаборатория знаний, Интернет-университет информационных технологий, 2014. - 304 c.
14. Фримен, Адам ASP.NET 4.5 с примерами на C# 5.0 для профессионалов / Адам Фримен. - М.: Вильямс, 2014. - **256** c.
15. Фримен, Адам ASP.NET MVC 4 с примерами на C# 5.0 для профессионалов / Адам Фримен. - М.: Вильямс, 2013. - 688 c.
16. Фримен, Адам ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов / Адам Фримен. - М.: Вильямс, 2015. - 736 c.
17. Шапошников, И. Самоучитель ASP.NET / И. Шапошников. - М.: БХВ-Петербург, 2002. - 368 c.
18. Эвери, Дж. Microsoft ASP.NET. Конфигурирование и настройка / Дж. Эвери. - М.: Эком, 2005. - **123** c.
19. Эспозито, Дино Программирование с использованием Microsoft ASP.NET 3.5 / Дино Эспозито. - М.: Питер, Русская Редакция, 2009. - **708**c.
20. Эспозито, Дино Разработка веб-приложений с использованием ASP.NET и AJAX / Дино Эспозито. - М.: Питер, 2012. - 400 c.

ПРИЛОЖЕНИЯ

Приложение А

Класс User

public class User

{

public int Id { get; set; }

[Required]

public string FirstName { get; set; }

[Required]

public string LastName { get; set; }

[EmailAddress]

public string Email { get; set; }

[ForeignKey("Login")]

public int LoginId { get; set; }

public Login Login { get; set; }

[ForeignKey("Role")]

public int RoleId { get; set; }

public Role Role { get; set; }

public virtual ICollection<Pet> Pets { get; set; }

public User()

{

Pets = new List<Pet>();

}

}

Приложение Б

Класс RegistrationModel

public class RegistrationModel

{

[Display(Name = "First Name")]

[Required]

public string FirstName { get; set; }

[Display(Name = "Last Name")]

[Required]

public string LastName { get; set; }

[Required]

[Display(Name = "Email")]

[EmailAddress]

public string Email { get; set; }

[Required]

[Display(Name = "Nickname")]

public string Login { get; set; }

[Display(Name = "Password")]

[DataType(DataType.Password)]

[Required(ErrorMessage = "Password is required")]

public string Password { get; set; }

[Display(Name = "Password again")]

[Compare("Password")]

[DataType(DataType.Password)]

[Required(ErrorMessage = "Confirm Password is required")]

public string PasswordAgain { get; set; }

}

Приложение В

Класс PetooDbInitializer

public class PetooDbInitializer : DropCreateDatabaseAlways<AccountContext> //TODO create if not exists

{

protected override void Seed(AccountContext context)

{

var admin = new Role { Id = 1, Name = "Admin" };

var user = new Role { Id = 2, Name = "User" };

context.Roles.AddRange(new List<Role> { admin, user });

var regLogin = new Login { Password = "regular", UserLogin = "regular" };

var admLogin = new Login { Password = "admin", UserLogin = "admin" };

context.Logins.AddRange(new List<Login> { regLogin, admLogin });

var regularUser = new User() { FirstName = "Reqular", LastName = "User", Login = regLogin, Role = user };

var adminUser = new User() { FirstName = "Admin", LastName = "Administrator", Login = admLogin, Role = admin };

context.Users.AddRange(new List<User> { regularUser, adminUser });

var golden = new Breed { Name = "Golden Retriever" };

var shitsu = new Breed { Name = "Shih Tzu" };

var spaniel = new Breed { Name = "Cocker Spaniel" };

var bigl = new Breed { Name = "Bigl" };

context.Breeds.AddRange(new List<Breed>() { golden, shitsu, spaniel, bigl });

var pet = new Pet { Breed = golden, Name = "Archie", User = regularUser, Age = 3, Gender = true };

var pet1 = new Pet { Breed = shitsu, Name = "Kora", User = regularUser, Age = 5, Gender = false };

var pet2 = new Pet { Breed = spaniel, Name = "Bonya", User = regularUser, Age = 11, Gender = true };

var pet3 = new Pet { Breed = bigl, Name = "Sherlock", User = regularUser, Age = 6, Gender = true };

context.Pets.AddRange(new List<Pet>() { pet, pet1, pet2, pet3 });

base.Seed(context);

}

}

Приложение Г

Класс CustomMembershipProvider

public class CustomMembershipProvider : MembershipProvider

{

public override bool ValidateUser(string username, string password)

{

bool isValid = false;

using (var db = new AccountContext())

{

try

{

//User user = (from u in \_db.Users

// where u.Email == username

// select u).FirstOrDefault();

var user = (from u in db.Users

from l in db.Logins

where l.UserLogin == username && l.Password == password && u.LoginId == l.Id

//where u.Login == username && u.Password == password

select u).FirstOrDefault();

if (user != null /\*&& Crypto.VerifyHashedPassword(user.Password, password)\*/) // сделать шифрование/дешифрование паролей

{

isValid = true;

}

}

catch

{

isValid = false;

}

}

return isValid;

}

public MembershipUser CreateUser(string lastName, string firstName, string userName, string password, int roleId)

{

MembershipUser membershipUser = GetUser(userName, false);

if (membershipUser == null)

{

try

{

using (var db = new AccountContext())

{

//user.Password = Crypto.HashPassword(password);

db.Logins.Add(new Login { UserLogin = userName, Password = password });

db.SaveChanges();

var user = new User { LastName = lastName, FirstName = firstName, LoginId = db.Logins.AsEnumerable().Last().Id, RoleId = roleId };

//if (db.Roles.Find(2) != null)

//{

// user.RoleId = 2;

//}

db.Users.Add(user);

db.SaveChanges();

return GetUser(userName, false); // получаю MembershipUser

}

}

catch

{

return null;

}

}

return null;

}

public override MembershipUser GetUser(string username, bool userIsOnline)

{

try

{

using (var db = new AccountContext())

{

var user = (from u in db.Users

where u.Login.UserLogin == username

select u).FirstOrDefault();

Login login = (from l in db.Logins

where l.UserLogin == username

select l).FirstOrDefault();

if (login != null)

{

return new MembershipUser("CustomMembershipProvider", login.UserLogin, null, null, null, null, false, false, DateTime.Now, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue);

}

}

}

catch

{

return null;

}

return null;

}

#region NotImplemented

…

#endregion

}