Министерство образования и науки Российской Федерации
Псковский государственный университет

И.В. Антонов, Ю.В. Бруттан

ВЕБ-ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

для бакалавров направлений подготовки
09.03.02 - Информационные системы и технологии,
09.03.01 - Информатика и вычислительная техника,
09.03.04 – Программная инженерия
всех форм обучения

Рекомендовано к изданию редакционно-издательским советом
Псковского государственного университета

Псков
Псковский государственный университет
2017

УДК 681.3.06
ББК 32.97
А72

Рекомендовано к изданию редакционно-издательским советом Псковского государственного университета

Рецензенты:
– Лехин С.Н. – к.т.н., доцент, декан факультета вычислительной техники и электроэнергетики Псковского государственного университета
– Герасименко В.В. – к.т.н., доцент, инженер Псковского областного института повышения квалификации работников образования

Антонов И.В., Бруттан Ю.В.
А72 Веб-программирование
Учебно-методическое пособие для бакалавров по направлениям подготовки 09.03.02 "Информационные системы и технологии", 09.03.01 "Информатика и вычислительная техника", 09.03.04 "Программная инженерия" всех форм обучения / И.В. Антонов, Ю.В. Бруттан. – Псков: Псковский государственный университет, 2017. — 72 с.
ISBN

В учебно-методическом пособии изложены основы проектирования веб-приложений ASP.NET MVC в Visual Studio 2017. Материал пособия содержит также общие рекомендации по изучению дисциплины, контрольные вопросы, рекомендации по выполнению самостоятельной работы, задание для выполнения контрольной работы, рекомендации по выполнению контрольной работы, задания по выполнению лабораторных работ и рекомендации по их выполнению, задание и рекомендации по выполнению курсового проекта, список рекомендуемой литературы.

Для бакалавров направлений подготовки 09.03.02 "Информационные системы и технологии", 09.03.01 "Информатика и вычислительная техника", 09.03.04 "Программная инженерия" всех форм обучения.

УДК 681.3.06
ББК 32.97

ISBN

© Антонов И.В., Бруттан Ю.В. 2017
© Псковский государственный университет, 2017

СОДЕРЖАНИЕ

стр.

ВВЕДЕНИЕ **5**
РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ДИСЦИПЛИНЫ **6**
РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ САМОСТОЯТЕЛЬНОЙ РАБОТЫ **7**
КОНТРОЛЬНЫЕ ВОПРОСЫ **9**
КОНТРОЛЬНАЯ РАБОТА **10**
Задание 10
Общие требования к оформлению контрольной работы 10
ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ **11**
1. *Разработка веб-приложений на основе ASP.NET MVC* 11
2. *Веб-формы и валидация данных в ASP.NET MVC* 23
3. *Использование баз данных в ASP.NET MVC* 30
4. *Аутентификация и авторизация в ASP.NET MVC* 36

ЛАБОРАТОРНЫЕ РАБОТЫ **39**
ЛАБОРАТОРНАЯ РАБОТА №1 .... **39**
Задание 39
Рекомендации по выполнению работы 39
ЛАБОРАТОРНАЯ РАБОТА №2 .... **42**
Задание 42
Рекомендации по выполнению работы 42
ЛАБОРАТОРНАЯ РАБОТА №3 .... **43**
Задание 43
Рекомендации по выполнению работы 43
ЛАБОРАТОРНАЯ РАБОТА №4 .... **46**
Задание 46
Рекомендации по выполнению работы 46
ЛАБОРАТОРНАЯ РАБОТА №5 .... **50**
Задание 50
Рекомендации по выполнению работы 50
ЛАБОРАТОРНАЯ РАБОТА №6 .... **54**
Задание 54
Рекомендации по выполнению работы 54
ЛАБОРАТОРНАЯ РАБОТА №7 .... **57**
Задание 57
Рекомендации по выполнению работы 57
КУРСОВОЙ ПРОЕКТ **59**
Обобщенное задание 59
Содержание отчета по курсовому проекту 60
Оформление отчета по курсовому проекту 61

Рекомендации по выполнению курсового проекта 65
СПИСОК РЕКОМЕНДУЕМОЙ УЧЕБНОЙ ЛИТЕРАТУРЫ **67**
СПИСОК БИБЛИОГРАФИЧЕСКИХ ИСТОЧНИКОВ **68**
ПРИЛОЖЕНИЕ А **69**
ПРИЛОЖЕНИЕ Б **70**

ВВЕДЕНИЕ

Учебная дисциплина «Веб-программирование» по направлениям подготовки 09.03.02 – «Информационные системы и технологии» и 09.03.01 – «Информатика и вычислительная техника» относится к обязательным дисциплинам вариативной части Блока 1 «Дисциплины (модули)», а по направлению подготовки 09.03.04 – «Программная инженерия» относится к базовой части Блока 1.

Основной целью изучения дисциплины «Веб-программирование» является освоение технологии проектирования веб-приложений на основе современных средств программирования и платформ.

Задачами дисциплины является изучение веб-протоколов, языков разметки веб-страниц (HTML, CSS, DHTML), программных средств построения веб-приложений серверной и клиентской стороны (JavaScript, PHP, ASP.NET), средств доступа к базам данных в локальных и глобальных сетях.

В результате изучения данной дисциплины студент должен:

♦ *Знать* язык разметки веб-страниц HTML, управление стилями через CSS, основы языков JavaScript и PHP, веб-компоненты платформы .NET;

♦ *Уметь* использовать современные средства проектирования приложений, создаваемых на базе веб-технологий, проектировать сетевые приложения с веб-доступом, работающие с базами данных на основе архитектуры клиент-сервер;

♦ *Владеть* средствами проектирования программ, работающих в локальных и глобальных сетях TCP/IP.

РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ДИСЦИПЛИНЫ

Учебная дисциплина «Веб-программирование» изучается студентами в 6 и 7 семестрах.

Программой предусмотрено проведение лекционных занятий, написание контрольной работы, выполнение лабораторных работ и курсового проекта. В конце 6 семестра предусмотрен экзамен.

Контрольная работа должна быть сдана преподавателю на рецензию до начала зачётной недели. Задание для контрольной работы, требования к её оформлению, а также контрольные вопросы к экзамену и список рекомендуемой литературы приведены в данном учебно-методическом пособии в соответствующих разделах.

Для подготовки к экзамену и выполнения контрольной работы необходимо воспользоваться материалами лекций, описанием теоретических вопросов и литературой, приведенных в данном пособии, а также литературой и интернет-источниками, найденными студентом самостоятельно.

РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студентов по дисциплине «Веб-программирование» заключается в самостоятельной проработке перечисленных ниже теоретических вопросов. Для этого необходимо воспользоваться приведенным в данном учебно-методическом пособии списком рекомендуемой учебной литературы, а также требуется по возможности использовать информационные ресурсы сети Интернет для получения дополнительной информации по темам данной дисциплины.

Перечень теоретических вопросов для самостоятельной работы:

1. Язык HTML
2. Язык Java Script
3. Язык XML
4. Динамический HTML (DHTML)
5. Технология AJAX
6. Библиотека JQUERY
7. Доступ к базам данных на основе веб-технологий
8. Средства обеспечения информационной безопасности в глобальных сетях
9. Язык PHP
10. Язык Java
11. Язык Phyton
12. Язык Ruby

Список источников для выполнения самостоятельной работы

1. Django: разработка веб-приложений на Python / Форсье Джефф , Биссекс Пол; Чан Уэсли / Джефф Форсье, Пол Биссекс, Уэсли Чан ; [пер. с англ. А. Киселева]. – Москва: Символ-Плюс, 2013. – 451 с.
2. Java: объектно-ориентированное программирование для магистров и бакалавров: базовый курс по объектно-ориентированному программированию / Васильев Алексей Николаевич / А. Н. Васильев. – СПб: Питер, 2011. – 395 с. – ISBN 978-5-49807-948-6
3. XML. Работа с XML / Дэвид Хантер, Джефф Рафтер, Джо Фаусетт, Эрик ван дер Влист, и др. 4-е издание. — М.: Диалектика, 2009. — 1344 с. — ISBN 978-5-8459-1533-7.
4. Гибкая разработка веб-приложений в среде Rails / Руби Сэм, Томас Дэйв; Хэнссон Дэвид Хэйнемеер / Сэм Руби, Дэйв Томас, Дэвид Хэнссон ; [пер. с англ. Н. Вильчинский]. – 4-е изд. – СПб: Питер, 2013. – 463 с.
5. Изучаем jQuery. Перейдите на новый уровень работы с JavaScript, используя мощь jQuery / Каслдайн Эрл, Шарки Крэйг; Черник В. / Эрл Каслдайн, Крэйг Шарки; пер. с англ. В. Черник. – 2-е изд. – СПб: Питер, 2012. – 400 с. – ISBN 978-5-459-01619-2

6. Никсон Р. Создаем динамические веб-сайты с помощью PHP, MySQL и JavaScript / Р. Никсон; [пер. с англ. Н. Вильчинский]. – СПб: Питер, 2013. – 496 с.

7. Фрэйн Б. HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств / Б. Фрэйн; [перевод с английского В. Черник]. – СПб: Питер, 2014. – 298 с.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Основные свойства языка HTML.
2. Виды и свойства HTML-тегов.
3. Стандартная структура HTML-документа.
4. Состав секции HEAD HTML-документа.
5. Теги разметки текста.
6. Теги гиперссылок и ссылок на изображения.
7. Теги разметки таблиц.
8. Формы и элементы форм в HTML.
9. Навигационные карты в HTML.
10. Фреймы в HTML.
11. Язык XML.
12. Каскадные таблицы стилей CSS.
13. Управление компоновкой веб-страниц средствами CSS.
14. Основные характеристики языка Java Script.
15. Основные объекты объектной модели Java Script.
16. Обработка событий в Java Script.
17. Назначение и особенности технологии AJAX.
18. Возможности библиотеки JQUERY.
19. Основные свойства языка PHP.
20. Особенности языка Ruby.
21. Общая характеристика языка Phyton.
22. Возможности языка Java.

КОНТРОЛЬНАЯ РАБОТА

Задание

1. Написать реферат на тему «Сравнительный анализ языков программирования серверных веб-приложений. Выбор языка программирования в соответствии с вариантом из таблицы 1.

Таблица 1

Вариант	Номер последней цифры зачетной книжки	Тема
1	1, 5	PHP и Perl
2	2, 4	Java и ASP.NET
3	3, 6	Ruby и Python
4	7, 9	Ruby и PHP
5	8, 0	Python и Perl

Общие требования к оформлению контрольной работы

1) Контрольную работу требуется представить на рецензию преподавателю в печатной форме, на белой бумаге формата А4 (210x297мм).

2) Контрольная работа должна состоять из следующих компонентов:

♦ титульный лист (пример титульного листа приведен в *приложении А* данного пособия);

♦ задание;

♦ содержательную часть в соответствии с заданием;

♦ список использованной литературы.

3) Контрольная работа должна быть напечатана в текстовом редакторе (например, Microsoft Word) в соответствии со следующими требованиями:

♦ шрифт: Times New Roman;

♦ размер:14 пт;

♦ межстрочный интервал: 1,5;

♦ выравнивание основного текста: по ширине;

♦ поля: верхнее – 3,0 см; нижнее – 3,0 см; левое – 2,5 см; правое – 1,5 см;

♦ нумерация страниц: внизу, по центру.

4) В верхнем колонтитуле должна содержаться фамилия студента с инициалами.

5) Рисунки должны быть ориентированы по центру страницы и подписаны. Например:

Рис.1 Макет компоновки страниц сайта

ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ

В данном учебно-методическом пособии рассматривается разработка веб-приложений *ASP.NET MVC* на платформе *ASP.NET Core*. Платформа *ASP.NET Core* построена на основе *.NET Core*, которая представляет собой межплатформенную версию *.NET*. За счет принятия межплатформенного подхода компания *Microsoft* расширила область охвата *.NET*, сделав возможным развертывание приложений *ASP.NET Core* на более широком наборе сред размещения, а также предоставила разработчикам возможность создавать веб-приложения *ASP.NET Core* на машинах Linux и OS X/ macOS [1, с.23].

1. Разработка веб-приложений на основе ASP.NET MVC

1.1 Основные компоненты ASP.NET MVC

Аббревиатура MVC расшифровывается как *Model-View-Controller* (*Модель-Представление-Контроллер*) и представляет собой архитектурный паттерн, очень популярный в области веб-разработки.

Microsoft разрабатывала инструменты и фреймворки для веб-разработки на протяжении довольно длительного периода. *ASP.NET MVC* стала прорывом, поскольку, в отличие от предыдущих разработок, делает упор на чистый код, концепцию разделения и тестируемость. Компания Microsoft встроила в инфраструктуру *MVC Framework* инструменты с открытым кодом, такие как библиотека *jQuery*, учла проектные решения и передовой опыт конкурирующих платформ, а также выпустила в свет исходный код *MVC Framework* для изучения разработчиками [1, с.22].

MVC паттерн подразумевает, что *MVC* приложение будет разделено как минимум на три части:

- *Модели*, которые содержат или представляют данные, с которыми работают пользователи. Это могут быть простые модели представления, которые только представляют данные, передаваемые от контроллера представлению, или они могут быть моделями, которые содержат операции, преобразования и правила работы с данными.
- *Представления*, которые используются для того, чтобы обработать некоторые части модели в качестве пользовательского интерфейса.
- *Контроллеры*, которые обрабатывает входящие запросы, выполняют операции для модели и выбирают представления для показа пользователю.

Модель данных

Наиболее важной частью MVC приложения является *модель данных*. Модели создаются путем выявления данных, объектов, операций и правил, которые существуют в определенной области деятельности. Они должны поддерживаться создаваемым приложением. Программное представление выявленных понятий предметной области и их отношений в MVC реализуется через *модель*. При создании ASP.NET приложений MVC модель представляет

собой набор типов C# (классы, структуры и т.д.). Операции модели представлены методами, определенными в этих типах, а правила выражаются в логике внутри методов. На основе типов модели создаются экземпляры (объекты), представляющие определенные фрагменты данных.

Таким образом, модель является отдельным определением бизнес данных и процессов внутри MVC приложения.

Представление

Представление (*View*) используется для формирования внешнего вида страниц приложения, в котором они непосредственно доступны пользователям. В ASP.NET MVC представления являются файлами с расширением *cshtml*, которые содержат код с интерфейсом пользователя, как правило, на языке html. Несмотря на то, что представление в основном состоит из кода html, оно не является html-страницей. При компиляции приложения на основе требуемого представления сначала генерируется класс на языке *C#*, а затем этот класс компилируется. Все добавляемые представления, как правило, группируются по контроллерам в соответствующие папки в каталоге **Views**. Представления, которые относятся к методам контроллера **Home**, будут находиться в проекте в папке **Views/Home**. Однако при необходимости есть возможность создать в каталоге **Views** папку с произвольным именем, где будут храниться дополнительные представления, не обязательно связанные с определенными методами контроллера. Чтобы произвести рендеринг представления в выходной поток, используется метод **View()**. Если в этот метод не передается имени представления, то по умолчанию приложение будет работать с тем представлением, имя которого совпадает с именем метода действия.

Контроллер

В MVC контроллеры являются *C#* классами. Каждый открытый (*public*) метод в классе, производном от **Controller**, называется *методом действия*, который связан с настраиваемым URL через систему маршрутизации (роутингом) ASP.NET. При отправке запроса на URL, связанные с методом действия, исполняются выражения в классе контроллера для того, чтобы выполнить некоторые операции над доменной моделью, а затем выбрать представление для отображения клиенту.

На рисунке 1 показано взаимодействие между контроллером, моделью и представлением.

В MVC контроллеры находятся в папке под названием **Controllers**, которую Visual Studio создает при создании проекта. Когда браузер запрашивает стартовую страницу сайта в стандартном проекте интернет-приложения он получает выходные данные метода **Index** контроллера **HomeController**.

Метод **Index** возвращает объект типа **ViewResult**, создаваемый методом **View()** и содержащий отображаемую информацию запрошенной веб-страницы. Этот объект формируется на основе файла **Index.cshtml** из папки **Views**. Расширение файла *.cshtml* обозначает C# представление, которое будет

обрабатываться движком представления *Razor*. Файлы *cshtml* содержат в основном HTML код, а также элементы, начинающиеся с символа @. В них находится код, обрабатываемый движком представления *Razor* на серверной стороне. MVC определяет, какое представление использовать для контроллера на основе соглашения об именах, заключающегося в том, что представление имеет имя метода действия и содержится в папке, названной именем контроллера без окончания **Controller**.

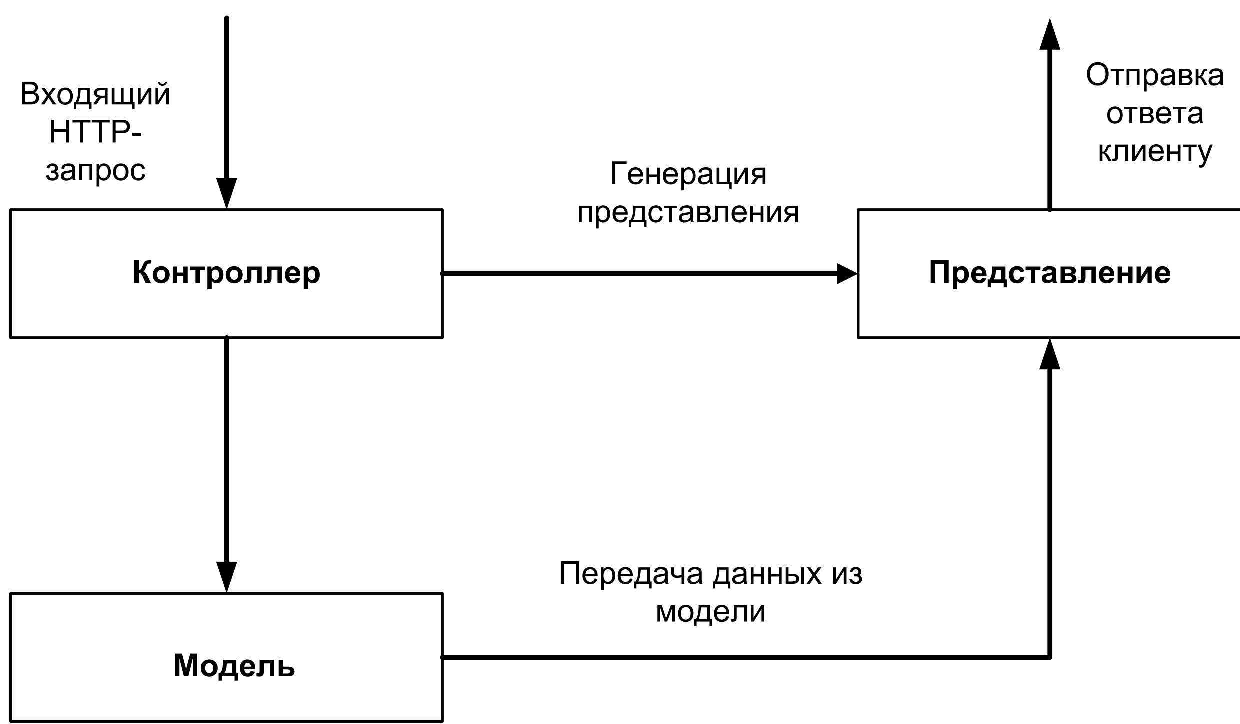

Рис. 1 Взаимодействие между контроллером, моделью и представлением

Задача контроллера MVC – создать некоторые данные и передать их представлению, которое отвечает за то, чтобы представить их в виде HTML. Одним из способов передачи данных от контроллера к представлению является использование объекта **ViewBag**.

ViewBag – это динамический объект, которому можно присвоить произвольные свойства, что делает эти значения доступными для любого представления, которое будет с ними дальше работать.

Например, **@ViewBag.Message** в стандартном файле **Index.cshtml** обеспечивает вставку в макет страницы текста из свойства **Message**, определенного в файле **HomeController.cs**.

1.2 Структура проекта MVC

В окне «**Обозреватель решений**» доступна структура проекта MVC. Стандартный проект содержит следующие разделы и компоненты:

- **Connected Services**: подключенные облачные службы.
- **Prorerties**: содержит файл настроек для отладочного запуска проекта из Visual Studio.

- **wwwroot**: этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов – изображений, скриптов *javascript*, файлов *css* и т.д., которые используются приложением. Цель добавления этой папки в проект состоит в разграничении доступа к статическим файлам, к которым разрешен доступ со стороны клиента и к которым доступ запрещен.
- **Зависимости**: все добавленные в проект пакеты и библиотеки.
- **Controllers**: папка для хранения контроллеров, используемых приложением.
- **Models**: папка для классов моделей.
- **Views**: каталог для хранения представлений.
- **appsettings.json**: хранит конфигурацию приложения.
- **bower.json**: файл, который управляет клиентскими зависимостями (библиотеки *javascript* и *css*), которые подключаются через менеджер пакетов *Bower*.
- **bundleconfig.json**: файл, который содержит задачи по минификации используемых скриптов и стилей, которые выполняются при построении проекта.
- **Program.cs**: файл, определяющий класс *Program*, который инициализирует и запускает хост с приложением.
- **Startup.cs**: файл, определяющий класс *Startup*, с которого начинается работа приложения, т.е. это точка входа в приложение.

В подкаталоге **Views\Shared** расположен файл **_Layout.cshtml**, который используется для редактирования макета страницы приложения.

1.3 Запуск приложения. Классы Program и Startup

Приложение *ASP.NET Core* является консольным приложением, которое инициализирует веб-сервер и затем посредством этого веб-сервера обрабатывает входящие запросы.

В проекте приложения содержится файл **Program.cs**, в котором определен одноименный класс **Program**, с которого начинается выполнение приложения:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace WebApplication1
```

```
{
   public class Program
   {
      public static void Main(string[] args)
      {
         BuildWebHost(args).Run();
      }
      public static IWebHost BuildWebHost(string[] args) =>
         WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>().Build();
   }
}
```

В методе **Main** вызывается метод **BuildWebHost**, в котором для создания хоста веб-приложения используется класс **WebHost** из пространства имен **Microsoft.AspNetCore**. Вызывается статический метод этого класса **CreateDefaultBuilder**, создающий экземпляр класса **WebHostBuilder**. С помощью последовательного вызова методов **WebHostBuilder** инициализирует веб-сервер для развертывания веб-приложения. Сначала вызывается метод **UseStartup<Startup>()**. Этим вызовом устанавливается стартовый класс приложения – класс **Startup**, с которого и будет начинаться обработка входящих запросов. Затем вызывается метод **Build(),** который создает хост – объект **IWebHost**, в рамках которого развертывается веб-приложение. После возврата в метод **Main** вызывается метод **Run()**. После этого приложение запущено, и веб-сервер начинает прослушивать все входящие HTTP-запросы.

Класс *Startup* должен определять метод *Configure()*, и также опционально в *Startup* можно определить конструктор класса и метод *ConfigureServices()*.

При запуске приложения сначала срабатывает конструктор, затем метод *ConfigureServices()* и в конце метод *Configure()*. Эти методы вызываются средой выполнения ASP.NET.

В проекте *ASP.NET Core* по шаблону *MVC* класс *Startup* выглядит следующим образом:

```
public class Startup
  {
   public Startup(IConfiguration configuration)
    {
      Configuration = configuration;
    }
```

```
public IConfiguration Configuration { get; }
public void ConfigureServices(IServiceCollection services)
  {
    services.AddMvc();
  }
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    else
    {
      app.UseExceptionHandler("/Home/Error");
    }
    app.UseStaticFiles();
    app.UseMvc(routes =>
    {
      routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
    });
  }
}
```

1.4 Макет на основе мастер-страницы

Мастер-страницы используются для создания единой компоновки веб-страниц для сайта в целом. Мастер-страницы представляют собой шаблоны веб-страниц, которые могут определять фиксированное содержимое и динамическое содержимое, различное на разных страницах сайта.

По структуре и содержанию мастер-страницы аналогичны файлам представлений, но они могут включать в себя другие представления. Например, можно определить на мастер-странице общие для всех остальных представлений меню, а также подключить общие стили и скрипты. В итоге в каждом отдельном представлении не потребуется указывать пути к файлам

стилей, а потом при необходимости их изменять. Специальные теги позволяют вставлять в определенное место на мастер-страницах другие представления.

По умолчанию при создании нового проекта *ASP.NET MVC Core* в проект добавляется мастер-страница под названием **_Layout.chtml**, которая находится в каталоге **Views/Shared**.

Главное отличие мастер-страниц от обычных представлений состоит в использовании метода **@RenderBody()**, который является плейсхолдером и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу.

По умолчанию представления подключают мастер-страницу за счет файла **_ViewStart.cshtml**. Этот файл находится в проекте в папке **Views**. Код этого файла добавляется в начало кода представлений при их запуске.

Файл **_ViewStart.cshtml** по умолчанию содержит следующий код:

```
@{
   Layout = "_Layout";
}
```

В каждом представлении через синтаксис *Razor* доступно свойство *Layout*, которое хранит ссылку на мастер-страницу. Здесь в качестве мастер страницы устанавливается файл **_Layout.cshtml**. При этом расширение можно не использовать.

Когда будет происходить рендеринг представления, то система будет искать мастер страницу *_Layout* по следующим путям:

/Views/[Название_контроллера]/_Layout.cshtml
/Views/Shared/_Layout.cshtml

Структура файла _Layout.chtml

Файл *_Layout.chtml* соответствует по структуре обычному *html* документу и содержит основные теги, определяющие структуру документа – *html, head, body*. Также файл включает в себя ряд серверных расширений *html*, обрабатываемых хелперами (*helpers*) на стороне сервера. Хелперы подключаются при использовании специальных тегов и атрибутов тегов. В секции *head* содержится хелперный тег *environment*, который используется для генерации разметки в зависимости от того, находится ли приложение в процессе разработки, тестирования или уже опубликовано на сервере. Как правило, данный тег-хелпер используется совместно с тегами *link* и *script*.

С использованием тегов *environment* к макету мастер-страницы подключены файлы библиотек *jQuery* и *bootstrap*, на основе использования которых реализована разметка страниц сайта. Эти библиотеки доступны, соответственно, и на всех страницах представлений, отображаемых с использованием мастер-страницы.

Тег **nav** используется для создания меню навигации по сайту. Основные элементы страницы помещены в контейнеры **div**, посредством которых назначаются классы стилей и задается компоновка страницы.

Тег **button** задает кнопку, которая отображается при малых размерах окна браузера и раскрывает свернутое в этом случае меню навигации.

Ссылки на страницы сайта в меню навигации реализованы на основе хелпер-тега гиперссылки **a** со специальными атрибутами **asp-area**, **asp-controller**, **asp-action**. Атрибут **asp-area** задает область веб-сайта. В простых проектах области обычно не используются. Атрибуты **asp-controller** и **asp-action** задают контроллер и его метод действия, который будет вызываться при активизации ссылки.

Метод **@RenderBody()** задает место вставки кода разметки представлений, которые используют данную мастер-страницу.

В конце контейнера **body** находятся теги **environment**, подключающие скрипты, которые будут доступны и в коде мастер-страницы, и в коде представлений.

Метод @**RenderSection("Scripts", required: false)** задает место вставки скриптов из использующих данную мастер-страницу представлений.

Файлы представлений стандартного макета веб-сайта

В папке **Views\Home** находятся файлы *Index.cshtml*, *Contact.cshtml*, *About.cshtml*, содержащие макеты информационных страниц сайта.

В файле *Index.cshtml* для демонстрации слайд-шоу баннеров использован компонент “**карусель**” библиотеки *bootstrap*. Компонент реализуется набором тегов *div*, которым заданы классы элементов карусели. Описание компонента с комментариями доступно по ссылке:

https://itchief.ru/lessons/bootstrap-3/bootstrap-3-carousel

Далее тег *div* с классом *row* задает *bootstrap* блок, стандартно содержащий 12 колонок. Четыре тега *div* с классом *col-md-3* задают колонки, каждая из которых по ширине занимает место трех стандартных колонок.

В колонках размещены списки ссылок на ресурсы по ASP.NET.

Файлы *Contact.cshtml* и *About.cshtml* содержат минимальные заготовки соответствующих страниц сайта.

1.5 Каскадные таблицы стилей

Для оформления и компоновки веб-страниц предназначена технология каскадных таблиц стилей (*Cascading Style Sheets, CSS*), или просто таблиц стилей. Таблица стилей содержит набор правил (стилей), описывающих оформление самой веб-страницы и отдельных ее фрагментов. Эти правила определяют: цвет текста и выравнивание абзаца, отступы между графическим изображением и обтекающим его текстом, наличие и параметры рамки у таблицы, цвет фона веб-страницы и многое другое.

Каждый стиль должен быть привязан к соответствующему элементу веб-страницы (или самой веб-странице). После привязки описываемые выбранным стилем параметры начинают применяться к данному элементу. Привязка может быть *явная* – когда указываем какой стиль к какому элементу веб-страницы привязан, или *неявная* – когда стиль автоматически привязывается ко всем элементам веб-страницы, созданным с помощью определенного тега.

Таблица стилей может включаться прямо в HTML-код веб-страницы или сохраняться в отдельном файле. Последний подход более соответствует современным концепциям, требующим, чтобы содержимое и представление веб-страницы были разделены. Кроме того, отдельные стили можно поместить прямо в тег HTML, создающий элемент веб-страницы; но такой подход используется сейчас довольно редко.

Стандарт, описывающий версию *CSS 1*, появился в 1996 году. В настоящее время широко поддерживается и применяется на практике стандарт *CSS 2* и ведется разработка стандарта *CSS 3*, ограниченное подмножество которого уже поддерживают многие веб-браузеры

Обычный формат определения стиля CSS:

<селектор> {
<атрибут стиля 1>: <значение 1>;
<атрибут стиля 2>: <значение 2>;
. . .
<атрибут стиля n-1>: <значение n-1>;
<атрибут стиля n>: <значение n>
}

Определение стиля включает **селектор** и **список атрибутов стиля с их значениями**.

Селектор используется для привязки стиля к элементу веб-страницы, на который он должен распространять свое действие. Фактически селектор однозначно идентифицирует данный стиль.

За селектором, через пробел, указывают список атрибутов стиля и их значений, заключенный в фигурные скобки.

Атрибут стиля представляет один из параметров элемента веб-страницы:

- *цвет шрифта,*
- *выравнивание текста,*
- *величину отступа,*
- *толщину рамки*

и др.

Значение атрибута стиля указывают после него через символ «**:**» (двоеточие). В некоторых случаях значение атрибута стиля заключают в кавычки.

Пары *<атрибут стиля>:<значение>* отделяют друг от друга символом «**;**» (точка с запятой).

Между последней парой *<атрибут стиля>:<значение>* и закрывающей фигурной скобкой символ «**;**» не ставят, т.к. в этом случае некоторые веб-браузеры могут неправильно обработать определение стиля.

Определения различных стилей разделяют пробелами или переводами строк.

Пример о стиля:

p { color: #0000FF }

где *p* — это селектор. Он представляет собой имя тега <p>.
color — это атрибут стиля. Он задает цвет текста.
#0000FF — это значение атрибута стиля *color*. Оно представляет код синего цвета, записанный в формате *RGB*.

Когда веб-браузер считывает описанный стиль, он автоматически применит его ко всем абзацам веб-страницы (тегам *<p>*).

В следующем примере в качестве селектора используется пользовательский идентификатор, начинающийся с символа «**.**» (точка). Такая разновидность стиля называется стилевым классом.

.redtext { color: #FF0000 }

Стилевой класс может быть привязан к любому тегу. Для этого используется атрибут **class**, значением которого является имя стилевого класса без точки в начале:

<p class="redtext">Внимание!</p>

В качестве значения атрибута *class* можно указать несколько имен стилевых классов, разделив их пробелами. В таком случае стили отдельных классов объединяются для данного тега.

Стиль уникального элемента страницы задается с использованием идентификатора этого элемента, заданного атрибутом **id** с добавлением лидирующего символа ***#*** :

#bigtext { font-size: large }

. . .

<p id="bigtext">Это большой текст.</p>

CSS позволяет создавать стили с несколькими селекторами — так называемые **комбинированные стили**. Такие стили служат для привязки к тегам, удовлетворяющим сразу нескольким условиям. Таким образом, можно указать, что комбинированный стиль должен быть привязан к тегу, вложенному в другой тег, или к тегу, для которого указан определенный стилевой класс.

Правила, которые установлены стандартом CSS при написании селекторов в комбинированном стиле:

- В качестве селекторов могут выступать имена тегов, имена стилевых классов и имена стилей уникальных элементов.
- Селекторы перечисляют слева направо и обозначают уровень вложенности соответствующих тегов, который увеличивается при движении слева направо (теги, указанные правее, должны быть вложены теги, что стоят левее).

- Если имя тега скомбинировано с именем стилевого класса, значит, для данного тега должно быть указано это имя стилевого класса, чтобы комбинированный стиль был применен.
- Селекторы разделяют пробелами.

Пример комбинированного стиля:

P.mini
{
color: #FF0000; font-size: smaller
}

Имя тега <*P*> скомбинировано с именем стилевого класса *mini*. Значит, данный стиль будет применен к любому тегу <*P*>, для которого указано имя стилевого класса *mini*. (Значение *smaller* атрибута стиля *font-size* задает уменьшенный размер шрифта.)

Стандарт CSS позволяет определить сразу несколько одинаковых стилей, перечислив их селекторы через запятую:

h1, .redtext, p em { color: #FF0000 }

Таблицы стилей, как правило, размещаются в отдельных файлах с расширением **css**. В стандартном проекте *APN.NET MVC* основная таблица стилей размещается в файле **Content\Site.css**

Обычно такие стили подключаются тегом **link** в секции **head** HTML документа:

<link rel="stylesheet" href="<адрес файла таблицы стилей>" type="text/css">

В *MVC* для оптимизации загрузки стилей и скриптов используется два механизма: **бандлинг** и **минификация**. **Бандлинг** представляет соединение скриптов или стилей в один общий файл или бандл. **Минификация** представляет сокращение содержимого скриптов или стилей. Для проведения бандлинга и минификации в ASP.NET Core используется расширение Visual Studio – **BundlerMinifier**. В проекте по умолчанию присутствует файл **bundleconfig.json**. В этом файле параметр **outputFileName** задает путь выходного файла, который будет формироваться в результате объединения файлов. Параметр **inputFiles** определяет через запятую набор файлов, которые будут объединяться. То есть, из файла "**wwwroot/js/site.js**" будет формироваться файл "**wwwroot/js/site.min.js**"

Дополнительный параметр **minify** указывает, будут ли минифицироваться включаемые в бандл файлы. Значение **enabled: true** включает минификацию. А значение "**renameLocals**": **true** позволяет сократить имена локальных переменных. Последний параметр **sourceMap** указывает, надо ли генерировать файл-карту сопоставления исходного и выходного файлов.

Также таблица стилей может размещаться непосредственно в HTML-документе. В этом случае она помещается в контейнерный тег **style** в секцию **head**.

Стили могут указываться прямо в HTML-коде веб-страницы. Это встроенные стили. В них отсутствует селектор и фигурные скобки. Встроенный стиль может быть привязан только к одному тегу — тому, в котором он находится.

Определение встроенного стиля указывают в качестве значения атрибута *style* нужного тега:

<p style="font-size: smaller; font-style: italic">Маленький курсив.</p>

На один тег могут одновременно действовать несколько стилей. В общем случае их эффекты совмещаются. Когда они противоречат друг другу, то приоритетный стиль определяется *по правилам каскадности*:

- Внешняя таблица стилей, ссылка на которую (тег *<link>*) встречается в HTML-коде страницы позже, имеет приоритет перед той, ссылка на которую встретилась раньше.
- Внутренняя таблица стилей имеет приоритет перед внешними.
- Встроенные стили имеют приоритет перед любыми стилями, заданными в таблицах стилей.
- Более конкретные стили имеют приоритет перед менее конкретными. Это значит, например, что стилевой класс имеет приоритет перед стилем переопределения тега, поскольку стилевой класс привязывается к конкретным тегам. Точно так же комбинированный стиль имеет приоритет перед стилевым классом.
- Если к тегу привязаны несколько стилевых классов, то те, что указаны правее, имеют приоритет перед указанными левее.

Атрибут стиля может быть помечен как «важный» что делает его приоритетным, даже если по правилам каскадности он таковым не является.

Чтобы сделать атрибут стиля важным, достаточно после его значения через пробел поставить слово **!important**:

EM { color: #00FF00;
font-weight: bold !important }

Теперь текст, выделенный тегом *<EM>*, всегда будет выводиться полужирным шрифтом, даже если данный параметр переопределен в более конкретном стиле.

2. Веб-формы и валидация данных в ASP.NET MVC

2.1 Веб-формы

Формы в HTML предназначены для передачи пользовательских данных на сервер. Форма состоит из элемента **<form>**, внутри которого располагаются поля ввода: *кнопки, однострочные и многострочные текстовые поля, выпадающие списки* и т.д. Функциональность формы определяется рядом атрибутов элемента *<form>*. Одним из них является **name**, в котором можно указать уникальное имя формы. В документе может быть несколько форм, но их имена не должны совпадать. С помощью имени формы подключенные к документу клиентские скрипты могут получать динамический доступ к полям формы еще до ее отправки на сервер.

Перед отправкой формы данные из полей приводятся к *MIME*-типу, соответствующему значению атрибута **enctype**. Затем они отправляются на сервер по адресу, указанному в **action**, методом, выбранным в атрибуте **method**. После этого серверное программное обеспечение может обработать полученную информацию, сформировать запрошенную страницу и отправить ее обратно браузеру.

Элементы ввода данных

Наиболее используемым элементом форм является *<input>*. С его помощью создаются поля для ввода текста, паролей и выбора файлов, а также кнопки, флажки и переключатели. В HTML 5 добавлены элементы *<input>* для ввода дат, числовых значений, телефонов, адресов, выбора цвета и т.д. Назначение элемента определяется атрибутом *type* (см. табл.2). Имя поля, определяемого элементом *<input>*, задается атрибутом *name*, а его значение по умолчанию указывается в *value*.

<input type="text" name="поле" value="значение">

Таблица 2

Значение type	Описание
Text	Значение по умолчанию. Элемент предназначен для ввода текстовой строки.
Password	Элемент предназначен для ввода паролей. Все вводимые символы заменяются жирными точками.
Reset	Кнопка очистки формы.
submit	Кнопка отправки данных на сервер.
image	Альтернативный вариант кнопки отправки данных в виде графического изображения, адрес которого указывается в атрибуте **src**, а альтернативный текст — в **alt**.

Окончание таблицы 2

Значение type	Описание
hidden	Скрытое поле. В браузере не отображается, но также может содержать значения **name** и **value**, отправляемые на сервер.
heckbox	Флажок. Отображается в виде небольшой области с установленной или снятой «галочкой». Если элемент содержит атрибут **checked="checked"**, то галочка по умолчанию будет установлена.
Radio	Переключатель, отображаемый в виде кружочка с жирной точкой (значение выбрано) или без нее (не выбрано). Выбор по умолчанию определяется атрибутом **checked="checked"**. В отличие от других типов полей, в форме может быть несколько элементов **<input type="radio">** с одинаковым name, однако выбран из них может быть только один.
File	Выбор файла. Отображается аналогично текстовому полю, но с добавленной справа кнопкой «**Обзор**». При нажатии на нее появляется диалоговое окно выбора файла.

В таблице 3 приведены значения атрибута *type*, добавленные в стандарте HTML 5.

Таблица 3

Значение type	Описание
search	Текстовое поле, предназначенное для ввода поискового запроса. Некоторые браузеры отображают на нем дополнительную кнопку очистки поля.
email	Текстовое поле для ввода адресов электронной почты.
url	Текстовое поле для ввода абсолютного *URL*
tel	Поле для ввода телефонных номеров
number	Поле числового ввода. Содержит кнопки-стрелки, позволяющие увеличивать и уменьшать значение.
range	Элемент для выбора значения из заданного диапазона. Представляет собой ползунок, минимальное и максимальное значения которого задаются в атрибутах **min** и **max** соответственно, а шаг — в атрибуте **step**.
time	Элемент для ввода времени. Аналогичен полю для ввода чисел, но с разделением на часы и минуты.
date	Элемент для выбора даты, представляющий собой выпадающий григорианский календарь.

Окончание таблицы 3

datetime -local	Комбинация двух предыдущих элементов для ввода даты и времени без учета часового пояса.
datetime	То же, что и предыдущий элемент, но с установленной временной зоной UTC.
week	Элемент для выбора недели. Визуально аналогичен элементу с **type="date"**, отличается лишь формат значения.
month	Элемент для выбора месяца. Визуально аналогичен элементу с **type="date"**, отличается лишь формат значения.
color	Элемент для выбора цвета в формате RGB.

Помимо элемента *<input>* для создания полей формы используются некоторые другие элементы. Для ввода многострочного текста предназначен тег *<textarea>*. Его размеры определяются в атрибутах *rows* и *cols*, задающих число видимых строк и символов в строке соответственно. Исходный текст для *<textarea>* задается не в атрибуте *value*, а указывается между открывающим и закрывающим тегами. Как и в *<input>*, максимальная длина значения может быть задана атрибутом *maxlength*.

```
<textarea rows="4" cols="20" name="myText">
    Здесь можно расположить большой
    многострочный текст…
</textarea>
```

Если текст не помещается в строку *<textarea>*, то он переносится на следующую строку. Если строк больше, чем вмещается в элемент, то автоматически появляется полоса прокрутки. В HTML 5 добавлен атрибут, определяющий способ передачи содержимого на сервер — это *wrap*. Установленный в значение *hard*, он добавляет код символа переноса в конец каждой строки. По умолчанию значением *wrap* является *soft*, при котором символы переноса строки не добавляются.

Оба элемента *<input>* и *<textarea>* поддерживают атрибут *readonly="readonly"*, который устанавливает их в режим «*только чтение*», запрещая редактирование содержимого.

Кнопки можно добавлять и с помощью элемента *<button>*. В *<button>* значением атрибута *type* могут быть *reset, submit* и *button*, задающие функции очистки формы, отправки данных и кнопки без определенного действия. Отличается тег *<button>* тем, что он контейнерный, и надпись на кнопке определяется не в атрибуте *value*, а в содержимом элемента.

Для организации выпадающих списков используют структуру, состоящую из элемента *<select>*, внутри которого размещаются дочерние *<option>*, представляющие варианты выбора.

```
<select name="food">
  <option value="pie">Пирог</option>
```

```
  <option value="bread" selected="selected">Хлеб</option>
  <option value="cookie" label="Печенье"></option>
</select>
```

Передаваемое на сервер имя поля указывается в атрибуте *name* элемента *<select>*, а его значение — в атрибуте *value* элемента *<option>*. В выпадающем списке, как и в случае с переключателем *<input type="radio" >*, из предлагаемых вариантов может быть выбран только один. Чтобы указать вариант по умолчанию применяется атрибут *selected="selected"*.

Если в элементе *<select>* указать атрибут *size* с некоторым числовым значением, то список становится не выпадающим, а обычным списком с указанным количеством видимых на экране вариантов. Если их на самом деле больше, то браузер добавит к элементу полосу прокрутки. С помощью атрибута *multiple="multiple"* можно разрешить пользователю выбрать несколько вариантов одновременно. Такой список также перестанет быть выпадающим, и чтобы указать необходимое количество видимых элементов, необходимо применять атрибут *multiple* в паре с *size*.

HTML 5 предоставляет возможность объединить выпадающий список с обычным элементом ввода *<input>*. Такой список может содержать, например, наиболее востребованные поисковые запросы или рекомендуемые значения заполняемого поля. Формируется он элементом *<datalist>*, в который вложены *<option>* с предлагаемыми в атрибутах *value* вариантами. Чтобы связать такой список с полем ввода, необходимо присвоить элементу *<datalist>* уникальный идентификатор *id* и указать его в значении атрибута *list* элемента *<input>*. По умолчанию *<datalist>* не отображается на странице, а появляется, только когда пользователь вводит данные в поле, к которому он привязан.

```
<input list="cars">
<datalist id="cars">
  <option value="BMW"></option>
  <option value="Ford"></option>
  <option value="Volvo"></option>
</datalist>
```

Формы в ASP.NET MVC

При создания форм в ASP.NET MVC удобно воспользоваться атрибутами тег-хелпера *asp-action* и *asp-controller*, которые задают имя метода действия, принимающего данные из формы, и класс контроллера, содержащий этот метод действия.

Например:

```
<form method="post" asp-action="Register" asp-controller="Home">
  …
</form>
```

В *ASP.NET MVC* действует соглашение, по которому методам действия контроллера данные из формы передаются через входные параметры, которые

обычно имеют тип *string*, но для числовых полей может быть использован тип *int*, а для *checkbox – bool*. Имя каждого поля формы (атрибут *name*) в этом случае должно совпадать с именем соответствующего ему параметра в объявлении метода действия. Также методу действия может передаваться объект модели, свойства которой будут содержать данные из полей отправленной на сервер формы. В этом случае имена полей формы должны совпадать с именами свойств модели.

Передача объектов моделей из контроллера в представление

Метод действия может передать объект модели в представление. В этом случае объект должен передаваться как параметр методу *View()*, вызовом которого завершается работа метода действия.

В начале кода представления в этом случае используется директива **@model**, с помощью которой указывается класс модели. Доступ к полям переданного из контроллера объекта в коде представления выполняется с использованием префикса **@Model**, например:

```
<span>e-mail:</span>@Model.Mail<br>
```

2.1 Валидация в MVC ASP.NET

Валидация позволяет проверить входные данные на наличие неправильных, корректных значений и должным образом обработать эти значения. В *ASP.NET MVC* валидация реализуется на основе сопровождения свойств модели специальными атрибутами валидации.

Атрибут **Required** из пространства имен **System.ComponentModel.DataAnnotations** задает обязательное поле. Его параметр **ErrorMessage** задает текст сообщения об ошибке. Пример:

```
[Required (ErrorMessage = "Не указано имя")]
public string Name { get; set; }
```

Атрибут **RegularExpression** предполагает, что вводимое значение должно соответствовать указанному в этом атрибуте регулярному выражению. Например, класс модели содержит свойство *Email*:

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
ErrorMessage = "Некорректный адрес")]
public string Email { get; set; }
```

Существует более компактный вариант – специальный атрибут **EmailAddress**:

```
[EmailAddress (ErrorMessage = "Некорректный адрес")]
public string Email { get; set; }
```

Также доступны специальные атрибуты **CreditCard**, **Phone**, **Url**. Атрибут **StringLength** первым параметром получает максимальную допустимую длину строки. Именованные параметры, в частности **MinimumLength** и **ErrorMessage**, позволяют задать дополнительные опции. Атрибут **Range** определяет минимальные и максимальные ограничения для числовых данных.

```
[Required]
[Range(1, 110, ErrorMessage = "Недопустимый возраст")]
public int Age { get; set; }
```

Атрибут **Compare** гарантирует, что два свойства объекта модели имеют одно и то же значение. Если, например, надо, чтобы пользователь ввел пароль дважды:

```
[Required]
public string Password { get; set; }

[Compare("Password", ErrorMessage = "Пароли не совпадают")]
public string PasswordConfirm { get; set; }
```

Атрибут **Remote** из пространства имен **Microsoft.AspNetCore.Mvc** для валидации свойства выполняет запрос на сервер к определенному методу контроллера. Если требуемый метод контроллера вернет значение *false*, то валидация не пройдена. Например:

```
[Remote(action: "CheckEmail", controller: "Home", ErrorMessage ="Email
уже используется")]
public string Email { get; set; }
```

В данном случае указано, что для проверки значения атрибут будет обращаться к методу **CheckEmail** контроллера **Home**:

```
public IActionResult CheckEmail(string email)
{
    if (email == "admin@mail.ru")
        return Json(false);
        else
    return Json(true);
}
```

Метод валидации должен возвращать объект **JsonResult**, который создается методом **Json()**. Параметр *true* означает, что валидация успешно пройдена.

Валидация на стороне сервера выполняется методом действия контроллера, принимающего объект модели, через проверку свойства **ModelState.IsValid**:

```
[HttpPost]
public IActionResult Create(Person person)
{
   if (ModelState.IsValid)
     {
       // Успех
         }
```

```
    else
        {
          // Неудача
        }
}
```

Если в объекте **ModelState** имеются какие-нибудь ошибки, то свойство **ModelState.IsValid** возвратит *false*.

Валидация на стороне клиента позволяет уменьшить количество обращений к серверу и произвести все действия по проверке значений непосредственно при вводе данных.

Для активизации валидации на стороне клиента необходимо подключить к представлению специальные скрипты валидации, добавив в конец файла представления, содержащего форму, код подключения частичного представления с этими скриптами:

```
@section Scripts {
   @{Html.RenderPartial("_ValidationScriptsPartial"); }
}
```

Для задания полей формы с валидацией в тегах полей формы используется атрибут тег-хелпера **asp-for** для указания имени поля формы и специальный тег **span** с атрибутом **asp-validation-for,** также указывающим имя поля формы. Тег **span** задает элемент разметки, в котором в дальнейшем выводится сообщение об ошибке при неудачной валидации:

```
<input type="text" asp-for="Name">
<span asp-validation-for="Name"></span>
```

В начало кода формы после тега **form** добавляется тег для отображения общих ошибок валидации, получаемых с сервера:

```
<div class="validation" asp-validation-summary="ModelOnly"></div>
```

3. *Использование баз данных в ASP.NET MVC*

ASP.NET MVC поддерживает создание баз данных на основе моделей данных, определенных в виде классов. Эту возможность обеспечивает платформа **Entity Framework** (*EF*), содержащая реализацию объектно-реляционного отображения (*ORM*), то есть, отображения структур данных, описанных в программном коде, в таблицы реляционных баз данных, а также обратное отображение. Чтобы воспользоваться этой возможностью необходимо определить соответствующие классы для моделей данных, на основе которых будут созданы таблицы базы данных. После этого можно получать данные из базы с помощью запросов к классам, а взаимодействие с базой, включая сохранение изменений, будет обеспечивать платформа *Entity Framework*.

В простейшем случае используемые модели данных представляют собой классы, содержащие автоматические свойства (с пустыми аксессорами *get* и *set*). Например, так может выглядеть модель записей о книгах для интернет-магазина:

```
public class Book
{
    // ID книги
public int Id { get; set; }

    // название книги
public string Name { get; set; }

    // автор книги
public string Author { get; set; }

   // цена
public int Price { get; set; }
}
```

Для таких моделей действует ряд соглашений об интерпретации их полей. Свойство с именем *Id* используется для первичных ключей создаваемых таблиц. При этом регистр символов не учитывается.

На основе классов моделей определяют класс контекста данных, представляющий базу данных в целом.

Класс контекста данных, содержащий одну таблицу, может выглядеть следующим образом:

```
public class ShopContext : DbContext
{
public DbSet<Book> Books { get; set; }
}
```

Для привязки класса контекста данных к базе данных может использоваться передача объекта конфигурации *DbContextOptions* в конструктор базового класса *DbContext*. При создании объекта конфигурации

для него задается источник данных и строка инициализации соединения с базой данных.

После этого для доступа к базе создается экземпляр класса контекста данных и вызываются его методы. Пример добавления записи о книге в базу данных:

```
ShopContext db = new ShopContext();
Book book = new Book();
book.Name = …;
book.Author = …;
book.Price = …;
db.Books.Add(book);
db.SaveChanges();
```

Доступ к записям из базы данных:

```
foreach (Book book in db.Books)
{
…
if(book.Price < 400)
{
…
}
…
}
```

Доступ по Id:

```
Book book = db.Books.Find(id);
```

Получение отсортированной по названиям коллекции книг на языке запросов *LINQ*:

```
var query = from b in db.Books
orderby b.Name
select b;
```

или используя функции EF:

```
var    query = db.Books.OrderBy(b => b.Name);
```

Поля элементов коллекции query, полученной в результате выполнения запроса, могут включаться в представления (View) и выводиться на страницах сайта.

3.1 Модели связанных таблиц базы данных

Для создания моделей связанных таблиц в *Entity Framework* в классы моделей сущностей добавляются особые поля. Эта модификация затрагивает оба класса моделей таблиц, участвующих в отношении.

Для организации связи **один-ко-многим** в класс, представляющий таблицу, содержащую внешний ключ, добавляются два поля – *внешний ключ и*

навигационное свойство. Навигационное свойство – это поле, представляющее объект из таблицы, на которую ссылается внешний ключ.

Имя навигационного свойства обычно задают совпадающим с *именем класса связанной сущности*. *Имя внешнего ключа* должно быть задано как *имя_навигационного_свойства+имя_ключа из связанной сущности*.

В класс, представляющий таблицу, на которую ссылаются через внешние ключи, также добавляется навигационное свойство – коллекция ссылающихся на нее объектов. Имя поля формируется как множественное число от имени типа ссылающихся объектов.

Приведем пример добавления связи один-ко-многим для классов *Client* (информация о зарегистрированном пользователе) и *Comment* (отзыв пользователя). **Жирным шрифтом выделены строки, реализующие связь один-ко-многим.**

```
public class Client
{
public int Id { get; set; }
public string UserName { get; set; }

public List<Comment> Comments { get; set; }
}

public class Comment
{
public int Id { get; set; }
public DateTime CommentDate { get; set; }
public string CommentText { get; set; }

public int ClientId { get; set; }
public  Client Client { get; set; }
}
```

Для организации связи **один-к-одному** в подчиненном классе надо задавать внешний ключ и навигационное свойство аналогично предыдущему примеру.

В основной класс добавляется только навигационное поле, ссылающееся на экземпляр подчиненного класса.

Приведем пример добавления связи один-к-одному для классов *Client* (информация о зарегистрированном пользователе) и *UserProfile* (дополнительные данные пользователя). **Жирным шрифтом выделены строки, реализующие связь один-к-одному.**

```
public class Client
{
public int Id { get; set; }
public string UserName { get; set; }

public  UserProfile UserProfile { get; set; }
}
```

```
public class UserProfile
{
public int Id { get; set; }
public string FIO { get; set; }
public string Address { get; set; }
public string Phone { get; set; }
public string Mail { get; set; }
public DateTime RegDate { get; set; }

public int ClientId { get; set; }
public  Client Client { get; set; }
}
```

Entity Framework (EF) поддерживает реализацию отношения **многие-ко-многим** между таблицами в базе данных. Такая связь в реляционных базах данных обеспечивается за счет использования промежуточной таблицы, с которой связываются две другие таблицы через отношение один-ко-многим. В результате между этими двумя таблицами образуется связь многие-ко-многим.

После создания связей при запросах данных из базы через классы *EF* можно получать данные из связанных объектов, обращаясь к навигационным свойствам полученных объектов.

Однако необходимо учитывать, что данные из связанных объектов для гарантированного доступа к ним в коде должны подгружаться явно. Для этого можно использовать метод **Load**, который вызывается через объект контекста базы данных.

Ниже показано использование метода **Load** для загрузки связанного объекта через навигационное свойство **Client**.

```
var query = from b in db.Comments
orderby b.CommentDate descending select b;
foreach (var item in query)

db.Entry(item).Reference(p => p.Client).Load();
```

Если через навигационное свойство загружается коллекция связанных объектов, то вместо метода **Reference** вызывается метод **Collection**, которому также передается делегат, возвращающий навигационное свойство.

3.2 Редактирование и удаление записей с использованием Entity Framework

Entity Framework Core позволяет выполнять стандартные операции с данными таблиц, такие как создание, получение, обновление и удаление данных.

Удаление производится с помощью метода **Remove**:

```
db.Users.Remove(user);
db.SaveChanges();
```

Данный метод установит статус объекта в **Deleted**, благодаря чему *Entity Framework* при выполнении метода **db.SaveChanges()** сгенерирует *SQL-выражение DELETE*.

Если необходимо удалить сразу несколько объектов, то можно использовать метод **RemoveRange():**

```
User user1 = db.Users.FirstOrDefault();
User user2 = db.Users.LastOrDefault();
db.Users.RemoveRange(user1, user2);
```

Для редактирования объекта достаточно изменить его данные и вызвать метод **db.SaveChanges()**. В результате будет сформировано *SQL-выражение UPDATE* для данного объекта, которое обновит объект в базе данных. Однако, если ссылка не изменяемый объект была получена в другом методе, требуется дополнительно вызывать метод метод **Update** перед вызовом **SaveChanges**:

```
db.Users.Update(user);
db.SaveChanges();
```

При необходимости обновить одновременно несколько объектов, применяется метод **UpdateRange()**:

```
db.Users.UpdateRange(user1, user2);
```

Каскадное удаление

Каскадное удаление представляет собой автоматическое удаление зависимой сущности после удаления главной. По умолчанию для сущностей применяется каскадное удаление, если наличие связанной сущности обязательно. Например:

```
public class Team

{
   public int Id { get; set; }
   public string Name { get; set; } // название команды

   public List<Player> Players { get; set; }
}

public class Player
{
   public int Id { get; set; }
   public string Name { get; set; }

   public int TeamId { get; set; } // внешний ключ
   public Team Team { get; set; }  // навигационное свойство
}
```

Здесь свойство внешнего ключа имеет тип **int**, оно не допускает значения **null** и требует наличия конкретного значения **id** связанного объекта **Team**. То есть для объекта **Player** обязательно необходимо наличия связанного объекта

Team. Соответственно, при удалении такого объекта **Team** будут удалены все объекты **Players**, содержащие его **TeamId**.

Можно изменить модели, указав необязательность наличия объекта Team:

```
public class Team
{
   public int Id { get; set; }
   public string Name { get; set; } // название команды
   public List<Player> Players { get; set; }
}

public class Player
{
   public int Id { get; set; }
   public string Name { get; set; }
   public int? TeamId { get; set; } // внешний ключ
   public Team Team { get; set; }  // навигационное свойство
}
```

Теперь внешний ключ имеет тип *Nullable<int>,* то есть он допускает значение *null*. Когда игрок не будет принадлежать ни одной команде, это свойство будет иметь значение *null*. Удаление команды, к которой принадлежал игрок, теперь не приведет к каскадному удалению игрока.

За дополнительной информацией об использовании *Entity Framework Core* следует обратиться к справочной документации по ссылке:

https://metanit.com/sharp/entityframeworkcore/

4. Аутентификация и авторизация в ASP.NET MVC

Аутентификация — это проверка учетных данных, предоставляемых пользователем. После успешной аутентификации пользователя, во все последующие запросы добавляется *cookie-файл* с идентификатором пользователя.

ASP.NET Core имеет встроенную поддержку аутентификации на основе *Cookies*. Для этого в *ASP.NET* определен специальный компонент, который сериализует данные пользователя в зашифрованные аутентификационные *Cookies* и передает их на сторону клиента. При получении запроса от клиента, в котором *содержатся* аутентификационные *Cookies*, происходит их валидация, десериализация и инициализация свойства **User** объекта **HttpContext**.

Поддержка аутентификации на основе *Cookies* включается вызовом соответствующего метода в методе **ConfigureServices** класса **Startup**:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).
AddCookie(o => o.LoginPath = new PathString("/Home/Login"));
```

Кроме того, в метод **Configure** класса **Startup** необходимо добавить вызов

```
app.UseAuthentication();
```

В ASP.NET MVC используется атрибут авторизации **[Authorize]**, который применяется к контроллерам и методам действий для того, чтобы ограничить доступ неавторизованным пользователям.

Для проверки статуса пользователя атрибут использует свойство **User** объекта **HttpContext**, которое устанавливается инфраструктурой ASP.NET.

Свойство **HttpContext.User** представляет объект интерфейса **IPrincipal**, который определен в пространстве имен **System.Security.Principal**. Этот интерфейс определяет метод **IsInRole()** и свойство **Identity**.

Свойство **Identity** возвращает объект интерфейса **IIdentity**, который связан с текущим запросом.

Метод **IsInRole()** в качестве параметра принимает роль и возвращает true, если текущий пользователь принадлежит данной роли.

Объект **IIdentity**, в свою очередь, предоставляет информацию о текущем пользователе через следующие свойства:

- **AuthenticationType**: тип аутентификации в строковом виде.
- **IsAuthenticated**: возвращает *true*, если пользователь аутентифицирован.
- **Name**: возвращает имя пользователя. Как правило, в качестве подобного имени используется логин, по которому пользователь входит в приложение.

Как было отмечено выше, объект **HttpContext.User** представляет интерфейс **IPrincipal** и у этого объекта есть свойство **Identity**, которое представляет интерфейс **IIdentity**. Для обоих этих интерфейсов ASP.NET Core предоставляет реализации по умолчанию – классы **ClaimPrincipal** и

ClaimsIdentity. То есть свойство **User** (**HttpContext.User**) фактически представляет объект **ClaimsPrincipal**.

Ключевым моментом этих двух классов является то, что они позволяют работать с объектами *claim*.

Объекты *claim* представляют некоторую информацию о пользователе, которую можно использовать для авторизации в приложении. Например, у пользователя может быть определенный возраст, город, страна проживания, любимая музыкальная группа и прочие признаки. Все эти признаки могут представлять отдельные объекты *claim*. И в зависимости от значения этих *claim* можно предоставлять пользователю доступ к тому или иному ресурсу. Таким образом, *claims* представляют более общий механизм авторизации нежели стандартные логины или роли, которые привязаны лишь к одному определенному признаку пользователя.

Каждый объект *claim* представляет класс **Claim**, который определяет следующие свойства:

- **Issuer**: "издатель" или название системы, которая выдала данный *claim.*
- **Subject**: возвращает информацию о пользователе в виде объекта **ClaimsIdentity.**
- **Type**: возвращает тип объекта *claim.*
- **Value**: возвращает значение объекта *claim.*

В простейшем случае для создания для создания аутентификационных *Cookies* может использоваться следующий объект *Claim*:

```
var claims = new List<Claim>
{
   new Claim(ClaimsIdentity.DefaultNameClaimType, userName)
};
```

Для создания *claim* его конструктору передается тип и значение. Тип **ClaimsIdentity.DefaultNameClaimType** фактически представляет логин. А **userName**, в данном случае будет представлять значение, которое затем можно будет получить через выражение **User.Identity.Name**.

Для работы с объектами **Claim** в классе **ClaimsPrincipal** есть следующие свойства и методы:

- **Identity**: возвращает объект **ClaimsIdentity**, который реализует интерфейс **IIdentity** и представляет текущего пользователя;
- **FindAll(type)** / **FindAll(predicate)**: возвращает все объекты *claim*, которые соответствуют определенному типу или условию;
- **FindFirst(type)** / **FindFirst(predicate)**: возвращает первый объект claim, который соответствуют определенному типу или условию;
- **HasClaim(type, value)** / **HasClaim(predicate):** возвращает значение true, если пользователь имеет *claim* определенного типа с определенным значением;

- **IsInRole(name)**: возвращает значение **true**, если пользователь принадлежит роли с названием **name**.

С помощью объекта **ClaimsIdentity**, который возвращается свойством **User.Identity**, можно управлять объектами *claim* у текущего пользователя. В частности, класс **ClaimsIdentity** определяет следующие свойства и методы:

- **Claims**: свойство, которое возвращает набор ассоциированных с пользователем объектов *claim*;
- **AddClaim(claim)**: добавляет для пользователя объект *claim*;
- **AddClaims(claims)**: добавляет набор объектов *claim*;
- **FindAll(predicate)**: возвращает все объекты *claim*, которые соответствуют определенному условию;
- **HasClaim(predicate)**: возвращает значение **true**, если пользователь имеет *claim*, соответствующий определенному условию;
- **RemoveClaim(claim)**: удаляет объект *claim*.

Для создания объекта **ClaimsIdentity** в его конструктор передается набор *claim*, тип аутентификации (**ApplicationCookie**), тип для *claim*, представляющего логин, и тип для *claim*, представляющего роль.

Созданный объект **ClaimsIdentity** передается в конструктор **ClaimsPrincipal**. Этот объект **ClaimsPrincipal** будет в дальнейшем доступен через свойство **HttpContext.User**.

```
// создаем объект ClaimsIdentity
ClaimsIdentity id = new ClaimsIdentity(claims, "ApplicationCookie",
ClaimsIdentity.DefaultNameClaimType,
ClaimsIdentity.DefaultRoleClaimType);
// установка аутентификационных cookies
await HttpContext.SignInAsync("Cookies", new ClaimsPrincipal(id));
```

Для установки *Cookies* применяется асинхронный метод контекста **HttpContext.SignInAsync().** В качестве параметра он принимает схему аутентификации, которая была использована при создании конфигурации в классе **Startup**. В примере это строка "**Cookies**". В качестве второго параметра передается объект **ClaimsPrincipal**, который представляет пользователя. После вызова этого метода пользователь является авторизованным.

Для выхода пользователя (*удаления аутентификационных Cookies*) применяется асинхронный метод **HttpContext.SignOutAsync()**, который также принимает в качестве параметра схему аутентификации.

ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторные работы по дисциплине «Веб-программирование» посвящены проектированию веб-приложений ASP.NET MVC в среде Visual Studio 2017.

В рамках выполнения курса лабораторных работ каждым студентом должен быть разработан веб-ресурс, представляющий фирму, организацию, частный или общественный проект. В проекте должно использоваться сохранение структурированной информации в базе данных и отображение этой информации на страницах сайта – каталог товаров (или услуг) и т.д.

Для упрощения части работы, связанной с информационным наполнением страниц и поиском графического материала, можно сделать аналог какого-либо существующего сайта с теми изменениями, которые требуются по заданиям лабораторных работ.

ЛАБОРАТОРНАЯ РАБОТА №1

Создание ASP.NET MVC проекта. Размещение на сайте страниц, включающих гиперссылки и изображения

Задание

1. Согласовать с преподавателем тему для разработки веб-сайта.
2. Подобрать первоначальную текстовую информацию и графические изображения по теме сайта.
3. Создать проект веб-сайта на основе типа проекта MVC.
4. Заменить заголовки и тексты стандартных макетов страниц шаблона сайта на тексты, соответствующие теме сайта.
5. Поместить графический логотип сайта в верхней области страниц. Клик по логотипу должен приводить к переходу на главную страницу сайта.
6. Добавить к проекту сайта страницу «*Полезные ресурсы*», на которой разместить краткие аннотации трех веб-сайтов, тематика которых связана с содержанием сайта. Аннотации должны включать гиперссылки и графические изображения.

Рекомендации по выполнению работы

<u>Пункт 3.</u>

1. Запустите *Visual Studio 2017* и создайте в папке группы проект веб-сайта. Имя проекта будет определяться именем папки проекта. Дальнейшие примеры предполагают использование имени *WebApplication1*:

Файл –> Создать –> Проект –>
    Visual C# –>
        Веб-приложение ASP.NET Core –>
            Веб-приложение (модель-представление-контроллер)

2. В настройке способа отладочного запуска выберите запуск приложения в браузере *Internet Explorer*.

3. Запустите приложение.

4. Завершите приложение и изучите структуру проекта в «*Обозревателе решений*».

Пункт 4.

1. Для изменения информационного содержания сайта отредактируйте макеты страниц сайта в папке *Views\Home* и код контроллера *Controllers\HomeController.cs*.

2. При создании разметки страниц используйте информацию по тегам HTML5 из справочника:

https://webref.ru/html

Справочник по библиотеке **bootstrap** доступен по ссылке:

https://webref.ru/layout/bootstrap

Пункт 5.

1. Добавьте картинку логотипа сайта в папку *Images*. Добавьте в файле *Views\Shared_Layout.cshtml* в конец блока *<div class="container">* следующий блок:

```
<div>
        <a asp-controller="Home" asp-action="Index">
<img src="~/images/имя_файла_с_логотипом" alt="">
  </a>
</div>
```

где вместо *имя_файла_с_логотипом* укажите имя файла картинки.

В файле стилей **wwwroot\css\site.css** для тега **body** подберите значение параметра **padding-top**, которое обеспечит отсутствие перекрытия шапки сайта с содержанием страниц.

Пункт 6.

1. Добавьте в папку *Views\Home* (используя команду «*Добавить-Создать элемент – Страница представления MVC*» контекстного меню папки *Views\Home*) файл *Links.cshtml*.

2. Разместите в этом файле *html* код, содержащий краткое описание, гиперссылки и логотипы веб-ресурсов.

3. Добавьте в файл *HomeController.cs* по аналогии с имеющимися там методами метод с заголовком *public IActionResult Links()*.

4. В файле *Views\Shared_Layout.cshtml* добавьте в список пунктов меню (тег *<ul>*) строку для страницы ссылок на другие ресурсы.

5. В файле *Startup.cs* добавьте перед вызовом метода создания маршрута *MapRoute* дополнительный вызов этого метода, обеспечивающий генерацию *URL без префикса /Home* для страниц сайта:

```
routes.MapRoute(
        name: "OnlyAction",
        template: "{action}",
    defaults: new { controller = "Home", action = "Index" });
```

6. Проверьте работоспособность выполненных изменений и дополнений.

ЛАБОРАТОРНАЯ РАБОТА №2

Использование таблиц стилей CSS для оформления страниц веб-сайта

Задание

1. Дополнить информационное наполнение макета сайта, созданного в первой лабораторной работе.
2. Используя стилевые таблицы *CSS* реализовать индивидуальное оформление веб-сайта. Применить индивидуальное оформление для:

- меню сайта;
- заголовков;
- текстов;
- таблиц;
- гиперссылок;
- изображений;

При выполнении задания использовать стилевые свойства из всех основных категорий:

- Текст
- Фон
- Поля и отступы
- Границы
- Позиционирование
- Таблицы
- Списки
- Эффекты

Рекомендации по выполнению работы

Пункт 2.

1. Изучите информацию по стилевому оформлению страниц веб-сайта из соответствующего раздела теоретической части данного пособия.
2. Изучите структуру и состав файла *Site.css*
3. Дополните файл *Site.css* собственными определениями стилей, выделив для них отдельную секцию в конце файла.
4. Используйте собственные определения стиля на страницах сайта в соответствии с заданием к работе.
5. При создании стилевого оформления сайта используйте информацию о стилевых свойствах из справочника: **https://webref.ru/css**

ЛАБОРАТОРНАЯ РАБОТА №3

Размещение на страницах веб форм и обработка данных форм. Валидация полей ввода

Задание

1. Реализовать с помощью элементов управления ввод и отображение страницы анкеты клиента или пользователя сайта. Состав элементов формы должен учитывать тематическую направленность сайта и включать в себя:

- поля ввода,
- флажки,
- радиокнопки,
- выбор из списка,
- ввод блока текста.

Обработчик формы на сервере должен возвращать в браузер страницу с информацией об успешном получении данных и с форматированным списком полученных данных.

2. Организовать валидацию на стороне клиента, включая проверку полей ввода на непустое значение и на соответствие данных стандартному формату ввода. Для поля ввода "*Имя Фамилия Отчество*" валидатор должен проверять, чтобы вводилось три слова. Номер мобильного телефона должен соответствовать формату "xxx-xxx-xx-xx". Адрес электронной почты должен проверяться на корректность.

Рекомендации по выполнению работы

<u>Пункт 1.</u>

1. Создайте в папке ***Models*** *cs-файл* класса модели анкетных данных пользователя сайта. Набор данных должен соответствовать содержанию формы из задания к работе. Класс может быть, например, назван *ClientModel*.

2. Добавьте в проект (в папку *Views\Home*) страницу представления с анкетой пользователя, например, *Register.cshtml*. Разместите на добавленной странице форму, содержащую элементы, соответствующие заданию к работе. Имена полей формы задайте в соответствии с именами свойств модели. В атрибуте *method* тега *form* укажите способ отправки *post*. В начало файла представления добавьте директиву @*model* с указанием полного (с пространством имен) имени класса модели. Например:

@*model WebApplication1.Models.ClientModel*

3. В *HomeController.cs* добавьте методы действия, вызываемые при запросе страницы анкеты и при отправке заполненной анкеты. Оба метода будут иметь имя, совпадающее с именем файла представления. Первый из них вызывается по запросу *GET*, а второй – при отправке формы методом *POST*. Сопроводите методы соответствующими атрибутами *[HttpGet]* и *[HttpPost]*

GET-версия не будет получать входных параметров. POST-версия должна получать входной параметр, имеющий тип модели данных и передавать его в качестве параметра методу *View()*.

Пример возможного объявления второго метода:

```
[HttpPost]
public IActionResult Register(ClientModel User)
  {
        …
        return View(User);
  }
```

4. На страницу представления с анкетой добавьте код ветвления по наличию объекта модели. Если данные модели присутствуют, значит, была принята заполненная форма и следует выводить отформатированную страницу с информацией для пользователя о принятых от него данных. При отсутствии объекта модели выполняется начальная загрузка анкеты и следует отображать на странице форму для ввода информации.

Соответствующее ветвление может быть организовано в *cshtml* файле представления следующим образом:

```
@if (Model == null)
{
// Здесь выводится форма
}
else
{
// Здесь выводятся данные пользователя
}
```

5. Добавьте ссылку на созданную страницу в меню навигации сайта.
6. Убедитесь, что анкета отображается, заполняется и отчет о приеме данных отображается после отправки анкеты.

<u>Пункт 2.</u>

1. Для доступности скриптов валидации в коде представления добавьте в конец *cshtml* файла код подключения частичного представления со скриптами валидации:

```
@section Scripts {
  @{Html.RenderPartial("_ValidationScriptsPartial"); }
}
```

2. Дополните свойства модели атрибутами валидации в соответствии с информацией из теоретического раздела данного пособия. Для контроля формата ввода имени, фамилии и отчества и номера телефона используйте следующие регулярные выражения.

- Три слова на русском языке:

'^[А-ЯЁ][а-яё]+ [А-ЯЁ][а-яё]+ [А-ЯЁ][а-яё]+$'

- Номер телефона формата xxx-xxx-xx-xx:

'^[0-9]{3}-[0-9]{3}-[0-9]{2}-[0-9]{2}$'

3. В тегах полей формы, подлежащих валидации, используйте атрибут *asp-for* вместо атрибута *name*. Добавьте теги *span* с атрибутом *asp-validation-for* для вывода сообщений об ошибках валидации.
4. Добавьте в файл *site.css* стили для выделения цветом сообщений об ошибках и рамок полей с ошибками:

```
.field-validation-error {
   color: #b94a48;
}
.input-validation-error {
   border: 1px solid #b94a48;
}
.validation-summary-errors {
   color: #b94a48;
}
```

5. Проверьте валидацию данных, вводимых в поля формы, и отображение страницы с информацией о результатах заполнения анкеты.

ЛАБОРАТОРНАЯ РАБОТА №4

Сохранение информации в базе данных. Формирование страниц сайта на основе выборок из базы данных

Задание

1. Добавить в проект возможность сохранения данных анкет пользователей в базе данных сайта.
2. Добавить на сайт страницу с таблицей, содержащей упорядоченную по именам в алфавитном порядке информацию о заполнивших анкету пользователях сайта.
3. Оформить страницу с таблицей со сведениями о пользователях в соответствии со стилем сайта.

Рекомендации по выполнению работы

<u>Пункт 1.</u>

1. Используя диспетчер пакетов *NuGet* добавьте в проект библиотеки платформы *Entity Framework*. Для этого запустите консоль менеджера (**"Средства"–"Диспетчер пакетов NuGet"–"Консоль диспетчера пакетов"**) и введите там следующие команды:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
```

2. Добавьте в класс модели данных пользователя свойства для первичного ключа и для сохранения даты регистрации. Первым полем сделайте:

```
public int Id { get; set; }
```

Дополните модель информацией о дате и времени заполнения анкеты (тип поля *DateTime*).

3. Добавьте в папку *Models* класс модели базы данных. Класс должен наследовать от класса *Microsoft.EntityFrameworkCore.DbContext*. Класс и файл для этого класса можно назвать *SiteContext*. Код класса может быть следующим:

```
using Microsoft.EntityFrameworkCore;

namespace WebApplication1.Models
{
   public class SiteContext : DbContext
   {
      public DbSet<ClientModel> Users { get; set; }
      public SiteContext(DbContextOptions<SiteContext> options)
         : base(options)
      {
      }
```

```
    }
}
```

4. Чтобы подключаться к базе данных следует задать параметры подключения. Для этого измените файл *appsettings.json*, добавив в него определение строки подключения:

```
{
 "ConnectionStrings": {
"DefaultConnection": "Server=(localdb)\\mssqllocaldb;
Database=sitedb;AttachDbFileName=%CONTENTROOTPATH%\\App_Data\\
sitedb.mdf;Trusted_Connection=True;MultipleActiveResultSets=true"
 },
 // остальное содержимое файла
}
```

Содержание строки подключения задает размещение базы данных в папке *App_Data*, которую надо добавить в проект.

В данном случае будет использоваться сервер баз данных **LocalDB**, который представляет собой облегченную версию *SQL Server Express*, предназначенную специально для разработки приложений.

Имя базы данных, указанное в строке подключения после *Database=* может потребоваться изменить, если база данных с этим именем уже существует. Лучше сразу указать другое имя.

5. Измените файл *Startup.cs*.

- К директивам *using* добавьте:

```
using WebApplication1.Models;
using Microsoft.EntityFrameworkCore;
```

- Добавьте в класс *Startup* поле:

```
private string _contentRootPath = "";
```

В этом поле будет сохраняться путь к корневой папке проекта.

- Замените конструктор класса *Startup* на следующий:

```
public Startup(IHostingEnvironment env)
{
  var builder = new ConfigurationBuilder()
  .SetBasePath(env.ContentRootPath)
  .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
  .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
  .AddEnvironmentVariables();
  Configuration = builder.Build();
 _contentRootPath = env.ContentRootPath;
 }
```

Этот вариант инициализации конфигурации позволяет получить и сохранить строку с путем к папке с файлами проекта.

- В метод *ConfigureServices* перед вызовом *services.AddMvc()* добавьте строки:

```
    string connection =
Configuration.GetConnectionString("DefaultConnection");
    if (connection.Contains("%CONTENTROOTPATH%"))
      {
        connection = connection.Replace("%CONTENTROOTPATH%",
_contentRootPath);
      }
      services.AddDbContext<SiteContext>(options =>
options.UseSqlServer(connection));
```

Добавление контекста данных в виде сервиса позволит затем получать его в конструкторе контроллера как входной параметр.

Через вызов *Replace* в строку инициализации подставляется актуальный путь к папке с базой данных.

6. Пересоберите проект.
7. В консоли диспетчера пакетов *NuGet* выполните команды:

```
Add-Migration Initial
Update-Database
```

Первая из этих команд добавляет в проект код класса для создания базы данных. Вторая выполняет создание базы данных.

8. Измените файл *HomeController.cs*.

- К директивам *using* добавьте:

```
using Microsoft.EntityFrameworkCore;
```

- Добавьте в класс *HomeController* поле для контекста базы данных и конструктор, инициализирующий это поле:

```
private  SiteContext db;
public HomeController(SiteContext context)
      {
        db = context;
     }
```

- В метод действия, принимающий данные из формы, добавьте сохранение записи о пользователе в базе данных. Для заполнения поля с датой используйте статическое свойство *Now* структуры *DateTime*, которое возвращает текущую дату и время.
- Для сохранения записи в базе добавьте строки перед завершением метода:

```
db.Users.Add(User);
db.SaveChanges();
```

9. Убедитесь, что добавленные анкеты пользователей теперь сохраняются в базе данных. Для этого просмотрите данные таблицы *Users* в *Обозревателе объектов SQL Server*, доступном через меню «*Вид*» Visual Studio.

<u>Пункт 2.</u>

1. Добавьте к проекту веб-страницу, на которую будет выводиться список и данные заполнивших анкеты пользователей в алфавитном порядке. Для этого потребуется добавить соответствующее представление в папку *Home* и обработчик запроса (метод действия) в *HomeController*. Ссылку на эту страницу добавьте в меню навигации по сайту. Метод действия в *HomeController* может передавать коллекцию отсортированных данных пользователей в представление через параметр метода *View*:

```
return View(db.Users.OrderBy(b => b.Name).ToList());
```

В этом случае представление должно принимать в качестве входной модели динамический массив объектов:

```
@model List<WebApplication1.Models.ClientModel>
```

В представлении для заполнения строк таблицы можно использовать цикл *foreach*:

```
@foreach(var Client in  Model)
{
  <tr>
  <td>@Client.Name</td>
  <td>@Client.Address</td>
  <td>@Client.Phone</td>
…
  </tr>
}
```

2. Убедитесь, что добавленная в базу данных информация о пользователях отображается на странице пользователей упорядоченной в алфавитном порядке.

ЛАБОРАТОРНАЯ РАБОТА №5

Авторизация пользователей сайта.
Разграничение прав доступа пользователей

Задание

1. Реализовать на сайте регистрацию пользователей. Совместить заполнение анкеты с процедурой регистрации пользователей сайта, включая ввод логина и пароля. При вводе для регистрации уже используемого логина или e-mail форма не должна успешно проходить валидацию.
2. Реализовать вход и выход зарегистрированного пользователя и добавить соответствующий пункт в меню навигации сайта (с отображением логина вошедшего пользователя).
3. Сделать страницу сайта со списком сведений о зарегистрированных пользователях видимой в меню и доступной только администратору сайта. Дополнить сведения о пользователях в списке пользователей информацией о логине пользователя.

Рекомендации по выполнению работы

Пункт 1.

1. Через менеджер *NuGet* добавьте в проект библиотеку *Microsoft.AspNetCore.Authentication.Cookies* .

2. Добавьте в файл Startup.cs директивы использования пространств имен:

Microsoft.AspNetCore.Http
Microsoft.AspNetCore.Authentication
Microsoft.AspNetCore.Authentication.Cookies

3. В метод **ConfigureServices()** класса *Startup* в соответствии со справкой к работе добавьте вызов метода *AddAuthentication().*

4. В метод **Configure()** класса *Startup* добавьте вызов:

app.UseAuthentication();

5. Дополните класс модели пользователя полями для ввода логина и пароля. Задайте им атрибуты валидации.

6. Соберите проект. Обновите структуру базы данных, используя команды диспетчера пакетов Nuget:

Add-Migration Update1
Update-Database

7. Добавьте в класс модели данных пользователя дополнительное поле для повторного (проверочного) ввода пароля. Это поле не должно сохраняться в базе данных. Для этого его надо сопроводить атрибутом *[NotMapped]* из пространства имен *System.ComponentModel.DataAnnotations.Schema,* которое

должно быть подключено в начале файла директивой *using*. Для поля проверки ввода пароля добавьте атрибут валидации:

[Compare("Password", ErrorMessage = "Пароль введен неверно")]

8. В файл с классом *HomeController* добавьте директивы использования пространств имен *System.Security.Claims, Microsoft.AspNetCore.Authorization, Microsoft.AspNetCore.Authentication.*

9. Дополните метод действия, принимающий регистрационную форму, проверкой уникальности вводимых логина и e-mail. В случае ошибки следует устанавливать состояние ошибки модели через вызов *ModelState.AddModelError* и не передавать в представление объект модели. Возможная реализация проверки уникальности логина:

```
var found = db.Users.FirstOrDefault(u => u.Login == User.Login);
    if(found != null)
    {
 ModelState.AddModelError("", $"Логин {User.Login} уже используется!");
            return View();
    }
```

10. Проверьте, чтобы при вводе не уникального логина или пароля регистрация не выполнялась и над формой регистрации выводилось сообщение об ошибке.

Пункт 2.

1. Создайте модель данных для формы входа: класс *LoginModel* с атрибутами валидации.
2. Создайте представление для формы входа, работающее с моделью *LoginModel*.
3. По аналогии с формой регистрации реализуйте *GET* и *POST* версии метода действия *Login*.
4. Метод, принимающий данные для логина, должен быть асинхронным, так как асинхронным является вспомогательный метод аутентификации *SignInAsync*. Возможная реализация этого метода:

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel model)
{
   if (ModelState.IsValid)
   {
      ClientModel user = db.Users.FirstOrDefault(u => u.Login == 
model.Login && u.Password == model.Password);
      if (user != null)
      {
         await Authenticate(model.Login); // аутентификация
         return RedirectToAction("Index", "Home");
      }
```

```
            ModelState.AddModelError("", "Некорректные логин и(или)
пароль");
        }
        else
            ModelState.AddModelError("", "Некорректные логин и(или)
пароль");
        return View(model);
    }
    private async Task Authenticate(string userName)
    {
         var claims = new List<Claim>
        {
            new Claim(ClaimsIdentity.DefaultNameClaimType, userName),
            new Claim(ClaimsIdentity.DefaultRoleClaimType, userName ==
"admin" ? "admin": "user")
        };
        // создаем объект ClaimsIdentity
        ClaimsIdentity id = new ClaimsIdentity(claims, "ApplicationCookie",
ClaimsIdentity.DefaultNameClaimType,
            ClaimsIdentity.DefaultRoleClaimType);
        // установка аутентификационных cookies
        await HttpContext.SignInAsync("Cookies", new ClaimsPrincipal(id));
    }
```

В примере для пользователя с логином *admin* создается роль *admin*, а для остальных пользователей – роль *user*.

Подразумевается, что администратор должен быть зарегистрирован на сайте до начала его эксплуатации.

Аналогично асинхронным методом должен быть *Logout*:

```
    public async Task<IActionResult> Logout()
    {
        await HttpContext.SignOutAsync("Cookies");
        return RedirectToAction("Index", "Home");
    }
```

5. Добавьте в меню навигации пункт для входа пользователя. После успешного входа этот пункт должен преобразовываться в "*Выход (Логин_пользователя)*". Для динамического формирования пункта меню можно проверять свойство *User.Identity.IsAuthenticated* и получать имя пользователя через свойство *User.Identity.Name*.

Пункт 3.

1. Для динамического формирования состава меню навигации в зависимости от статуса пользователя можно использовать в представлении код C#, включающий в страницу теги пункта меню по условию:

```
@if(User.Identity.IsAuthenticated && User.Identity.Name =="admin")
{
 ...
}
```

2. Для блокировки доступа к списку пользователей всем, кроме администратора, метод действия, возвращающий список пользователей, можно сопроводить следующим атрибутом:

```
[Authorize(Roles = "admin")]
```

ЛАБОРАТОРНАЯ РАБОТА №6

Использование Entity Framework со связанными таблицами базы данных

Задание

1. Добавить на сайт книгу отзывов, в которой:
- администратор может скрывать нежелательные записи;
- зарегистрированные пользователи – добавлять отзывы;
- все посетители – просматривать отзывы.

Отзывы должны выводиться в порядке убывания даты. Таблица пользователей должна быть связана отношением "*один ко многим*" с таблицей отзывов.

Рекомендации по выполнению работы

<u>Пункт 1.</u>

1. Создайте класс модели отзыва пользователя. Класс может иметь следующую структуру:

```
public class Comment
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public DateTime CommentDate { get; set; }
    public string CommentText { get; set; }
    public bool Hidden { get; set; }
}
```

Поле *Hidden* может использоваться для пометки скрытых от показа записей.

2. Добавьте к полю текста комментария атрибут валидации, требующий заполнения этого поля.
3. Добавьте в классы моделей пользователей и отзывов поля, организующие связь "*один ко многим*" в соответствии со справкой из теоретического раздела данного пособия.
4. Добавьте в класс контекста базы данных *SiteContext* поле для таблицы отзывами:

```
public DbSet<Comment> Comments { get; set; }
```

5. Соберите проект. В консоли диспетчера пакетов выполните следующие команды:

```
Add-Migration Update2
Update-Database
```

6. Добавьте в проект в папку *Views\Home* страницу с классом представления для отображения страницы отзывов и ввода отзыва. Представление должно работать с моделью отзыва.

7. В класс *HomeController* добавьте методы действия для вывода книги отзывов и сохранения добавляемых отзывов.

- Метод действия, который отображает книгу отзывов, может иметь следующий заголовок:

```
[HttpGet]
public ActionResult Comment()
```

- Метод действия, сохраняющий добавленный отзыв, может иметь следующий заголовок:

```
[HttpPost]
public ActionResult Comment(Comment comment)
```

- Список отзывов, полученный из базы данных, может передаваться в представление через поле *ViewBag.Comments*:

```
var query = from b in db.Comments
        orderby b.CommentDate descending
        select b;
ViewBag.Comments = query;
foreach (var item in query)
    db.Entry(item).Reference(p => p.ClientModel).Load();
```

8. В файл представления для отзывов добавьте код для отображения списка отзывов (*дата, логин / имя пользователя, текст отзыва*) и поля ввода нового отзыва. Цикл для вывода отзывов можно реализовать следующим образом:

```
@foreach(var comment in  ViewBag.Comments)
{
   ...
}
```

Для администратора справа от выводимого текста следует отображать ссылку для скрытия отзыва (или показа ранее скрытого отзыва). Признак входа под логином администратора можно получить, проверив условие:

```
@if (User.Identity.IsAuthenticated && User.Identity.Name == "admin")
```

Тогда в представлении можно ввести ячейку таблицы, заполняемую только для администратора по этому условию:

```
<a asp-controller="Home" asp-action="HideComment" asp-route-id =
"@comment.Id">
                @if(comment.Hidden)
                {
                    <span>Показать</span>
                }
                else
                {
```

```
            <span>Скрыть</span>
        }
</a>
```

В этом примере используется ссылка на метод действия *HideComment*, которому передается параметр *comment.Id*. Метод должен изменить признак видимости комментария. Возможна следующая реализация этого метода действия:

```
public ActionResult HideComment(int id)
{
   Comment comment = db.Comments.Find(id);
   if (comment != null)
   {
      comment.Hidden = !comment.Hidden;
      db.SaveChanges();
   }
   return RedirectToAction("Comment", "Home");
}
```

Вызов метода *RedirectToAction* в конце этого метода выполняет переадресацию на метод действия *Comment* контроллера *Home*.

9. Скрытые администратором отзывы не должны выводиться другим посетителям. Для этого вывод строк таблицы с текстом отзывов в представлении следует выполнять в теле цикла с проверкой свойства *comment.Hidden*.

10. Форма ввода отзыва должна отображаться только вошедшим под своим логином на сайт пользователям. Остальным должно отображаться сообщение о необходимости входа для добавления отзыва.

11. Запустите проект на выполнение. Убедитесь в работоспособности ввода и отображения отзывов.

ЛАБОРАТОРНАЯ РАБОТА №7

Редактирование данных. Администрирование сайта

Задание

1. Предоставить администратору сайта возможность редактировать отзывы.
2. Добавить возможность удаления администратором нежелательных отзывов из книги отзывов сайта.
3. Предоставить администратору возможность блокировать доступ к сайту нежелательным пользователям и удалять пользователей.

Рекомендации по выполнению работы

Пункт 1.

1. Для редактирования отзывов пользователей администратором добавьте в код представления страницы отзывов вывод для администратора рядом с отзывами ссылки "**Правка**", при активизации которой может вызываться метод действия, получающий *Id* отзыва и передающий его в представление через свойство объекта *ViewBag*. В представлении при совпадении этого свойства с *Id* отзыва для администратора должна выводиться форма редактирования выбранного отзыва вместо его текста. Чтобы поле *textarea* с атрибутом *asp-for* заполнялось не пустым содержанием, представлению должен передаваться объект модели комментария, содержащий редактируемый комментарий. Альтернативная реализация может заключаться в использовании в теге *textarea* атрибута *name* вместо *asp-for*. Тогда содержание текста блока может задаваться текстом внутри контейнера *textarea* и этот текст может быть получен из переменной цикла по отзывам в представлении *@comment.CommentText*.

- Возможная реализация метода действия:

```
public ActionResult EditComment(int id)
{
    var query = from b in db.Comments
            orderby b.CommentDate descending
            select b;
    ViewBag.Comments = query;
    foreach (var item in query)
        db.Entry(item).Reference(p => p.ClientModel).Load();
    CommentModel comment = db.Comments.Find(id);
    ViewBag.EditId = id;
    return View("~/Views/Home/Comment.cshtml", comment);
}
```

Использованный вариант вызова метода *View()* получает в качестве первого аргумента путь к файлу представления, что позволяет выводить через метод действия произвольное представление.

При отправке формы с измененным отзывом должно выполняться обновление текста отзыва в базе данных.

<u>Пункт 2.</u>

1. Для удаления отзывов добавьте для администратора ссылку «**Удалить**» рядом с отзывами. При переходе по этой ссылке должен вызываться метод действия, удаляющий отзыв из базы данных.

<u>Пункт 3.</u>

1. Добавьте в модель пользователя логическое поле – признак блокировки пользователя. Пересоберите проект и обновите структуру базы данных с использованием соответствующих команд диспетчера пакетов *NuGet*.
2. В списке пользователей, выводимом администратору, добавьте ссылки для блокировки и удаления пользователей. Реализуйте соответствующие методы действия.
3. При обработке формы входа на сайт, отправленной заблокированным пользователем, устанавливайте признак ошибки валидации модели:

ModelState.AddModelError("", "Вам заблокирован вход на сайт. Обратитесь к администратору.");

КУРСОВОЙ ПРОЕКТ

Обобщенное задание

Разработать веб-сайт организации (компании / фирмы) с элементами электронной коммерции.

ТЕМУ СОГЛАСОВАТЬ С ПРЕПОДАВАТЕЛЕМ.

1. Проект должен быть реализован на основе технологии *ASP.NET MVC*.
2. Информация о товарах (услугах), зарегистрированных пользователях, новостях, используемая и отображаемая на страницах ресурса, должна извлекаться из таблиц базы данных. Страницы проекта должны быть оформлены в едином стиле, используя таблицы стилей (*CSS*).
3. В оформлении страниц должны быть использованы различные графические элементы, цвета и шрифты.
4. Меню навигации по разделам должно быть доступно на всех страницах сайта.
5. Проект должен обеспечивать четыре уровня доступа к своим информационным ресурсам, в соответствии с возможными ролями посетителей:
   - Гость;
   - Клиент;
   - Менеджер;
   - Администратор.
6. Права на интерактивный доступ к данным проекта должны зависеть от статуса посетителя:
   a. *Гость* должен иметь возможность только просматривать некоторые страницы сайта, а также иметь возможность зарегистрироваться для получения статуса клиента;
   b. *Клиент* должен иметь возможность просматривать страницы сайта и выполнять заказ продукции или услуг (в зависимости от тематики сайта), а также просматривать, редактировать или удалять свои заказы;
   c. *Менеджер* должен иметь возможность просматривать страницы сайта и управлять заказами продукции или услуг клиентов (в зависимости от тематики сайта);
   d. *Администратор* через web-интерфейс должен выполнять функции удалённого управления информационным наполнением ресурса. Должны быть реализованы следующие функции администрирования: добавление, редактирование, удаление данных проекта (каталога продукции или услуг, новостей), а также добавление и удаление пользователей с ролью «Менеджер». Для получения административного доступа должна производиться аутентификация администратора.
7. Сайт должен содержать следующие основные информационные элементы:
   a. *Главная страница*, которая представляет организацию и должна содержать следующие элементы:

- название организации,
- логотип организации,
- краткая аннотация области функционирования организации,
- список из 5 последних новостей организации.

b. Страницу «*Новости*», которая должна содержать список всех новостей организации.

c. Страницу «*Контакты*», которая должна содержать адрес организации, контактные телефоны и информацию о разработчике сайта.

d. Страницу «*Каталог*» (каталог товаров либо услуг, либо других ресурсов, соответствующих тематике проекта). Каталог должен состоять из набора страниц, сочетающих графическую и текстовую информацию. Компоновка страниц должна происходить на основе выборок из базы данных. Каталог должен иметь иерархическую структуру (не менее 3 категорий по 5 наименований в каждой).

e. Страницу «*Корзина*», доступную только для «Клиента» (для просмотра и редактирования текущего заказа клиента).

f. Страницу «*Мои заказы*» доступную только для «Клиента» (для просмотра истории заказов клиента).

g. Страницу «*Заказы*», доступную только для «Менеджера» (для выполнения работ по управлению заказами «Клиентов»).

h. Страницу «*Регистрация*» (содержащую форму регистрации).

i. Набор страниц «Администратора» (конкретизировать самостоятельно).

Примечание 1. Содержание страниц подробно описать в информационных требованиях технического задания.

Примечание 2. Количество дополнительных страниц, соответствующих тематике сайта не ограничено.

Содержание отчета по курсовому проекту

Отчет по курсовому проекту должен содержать следующие разделы.

1. Техническое задание

Содержание технического задания (ТЗ) сформулировать с учетом особенностей конкретной темы курсового проекта и требований к содержанию ТЗ.

1.1. Название проекта. Назначение веб-сайта

1.2. Требования к веб-сайту

1.2.1. Функциональные требования

Содержит перечень конкретных функций, которые будут реализованы в соответствии с конкретными ролями пользователей.

1.2.2. Информационные требования

Содержит описание информационных элементов веб-сайта с их атрибутами.

2. Проектирование веб-сайта

2.1. Проектирование карты сайта

Содержит рисунок с картой сайта (см. Пример в приложении Б данного пособия) и пояснения к рисунку.

2.2. Проектирование базы данных WEB-сайта

2.2.1. Структура базы данных

Содержит рисунок «Схема базы данных» и пояснения к нему.

2.2.2. Описание таблиц базы данных

Содержит описание структуры таблиц базы данных, представленных в табличном виде.

2.3. Проектирование шаблонов страниц

Содержит рисунки шаблонов страниц с пояснениями.

2.4. Проектирование общей структуры модулей

Содержит рисунок со структурой модулей сайта с пояснениями.

3. Разработка модулей WEB-сайта

Содержит рисунки блок-схем алгоритмов модулей с пояснениями к ним.

3.1. Разработка модуля оформления заказа

3.2. Разработка модуля управления заказами зарегистрированных пользователей

3.3. Разработка модулей администрирования

Оформление отчета по курсовому проекту

Отчет по курсовому проекту является технической документацией, поэтому должен быть оформлен в соответствии с ГОСТ ЕСКД, ЕСПД.

Требования к оформлению отчета:

1. Общие

а) Отчет должен быть напечатан в текстовом редакторе (например, Microsoft Word) в соответствии со следующими требованиями:

- шрифт: Times New Roman;
- размер:14 пт;
- межстрочный интервал: 1,5;
- выравнивание (для основного текста): по ширине
- поля: верхнее – 3 см; нижнее – 2,0 см; левое – 2,5 см; правое – 1,5 см;
- отступ первой строки абзаца – 10 мм;
- нумерация страниц: внизу, по центру;
- в верхнем колонтитуле должна содержаться тема курсового проекта.

б) Каждый раздел отчета должен начинаться с новой страницы. Заголовок раздела отделяется от текста дополнительным межстрочным интервалом. Перенос слов в заголовке разделов не допускается, заголовки подразделов помещаются на отдельной строке. **Точка в конце заголовка не ставится!** Не допускается размещать заголовки разделов и подразделов на одной странице, а относящийся к ним текст на следующей странице. Заголовки подразделов можно выделять другим начертанием шрифта того же размера.

2. Оформление рисунков

а) Рисунки должны быть пронумерованы. Они нумеруются в пределах каждого раздела арабскими цифрами с указанием номера раздела, к которому иллюстрация относится (например, Рис.2.1.)

б) Рисунки должны быть ориентированы по центру страницы и подписаны. Точка в конце названия рисунка не ставится. Например:

Рис.2.1 Название рисунка

в) В тексте отчета обязательны ссылки на иллюстрации. Они задаются, например, в круглых скобках или без них в зависимости от контекста (например –"... схема (рис.4.1.) ", или – "на рисунке 1.1 представлен ...").

3. Оформление таблиц

а) Обозначение "Таблица..." ставится над заголовком таблицы в правом верхнем углу.

б) Название таблицы должно быть ориентировано по центру страницы.

в) Таблицы нумеруются в пределах каждого раздела арабскими цифрами с указанием номера раздела, к которому она относится (например, Таблица 2.1).

г) В тексте отчета обязательны ссылки на таблицы. Они задаются, например, в круглых скобках или без них в зависимости от контекста (например –"... описание структуры таблицы USER базы данных DB(таблица 2.1.) ", или – "из таблицы 8.5 видно ...").

4. Оформление блок-схем алгоритмов модулей.

Оформление алгоритмов программ должно соответствовать определенным требованиям. Единая система программной документации (ЕСПД) устанавливает правила разработки, оформления программ и программной документации. Правила выполнения схем алгоритмов, программ данных и систем регламентируются ГОСТ 19.701-90 ЕСПД.

Операции обработки данных и носители информации изображаются на схеме соответствующими блоками (символами). Большая часть из них представляет собой четырехугольники с размерами **a** и **b** или фигуры вписанные в четырехугольники с размерами а и b. Минимальное значение а равно 10 мм, ее увеличение производится на число, кратное 5 мм. Размер **b=1,5a**. В пределах одной схемы рекомендуется изображать блоки одинаковых размеров. Блоки могут нумероваться. Вид и назначение основных блоков представлен в таблице 4:

Таблица 4

Основные символы блок-схем алгоритмов

Наименование	*Обозначение*	*Функции*
1	*2*	*3*
	Символы данных	
Данные	a b	Символ отображает входные либо выходные данные, носитель которых не определен
Запоминающее устройство с прямым доступом		Символ отображает данные, хранящиеся в запоминающем устройстве с прямым доступом (магнитный диск, гибкий магнитный диск)
Документ		Символ отображает данные, представленные на носителях (документ для оптического или магнитного считывания, микрофильм, рулон ленты с итоговыми данными, бланки ввода данных)
	Символы данных	
Ручной ввод		Символ отображает данные, вводимые вручную во время обработки с устройств любого типа (клавиатура, переключатели, кнопки, световое перо, считыватели штрихового кода).
Дисплей		Символ отображает данные, представленные на отображающих устройствах (экран для визуального наблюдения, индикаторы ввода информации).
	Символы процесса	
Процесс	b a	Символ отображает функцию обработки данных любого вида (выполнение определенной операции или их группы, приводящее к изменению значения, формы или размещения информации, или к определению и т.п.)
Пред-определенный процесс		Символ отображает предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте (в подпрограмме, модуле).

Окончание таблицы 4

1	*2*	*3*
Ручная операция	b a a	Символ отображает любой процесс, выполняемый человеком.
Подготовка	b a	Символ отображает модификацию команды или группы команд с целью воздействия на некоторую последующую функцию. Часто используется для задания параметров оператора цикла.
Решение	b a	Символ отображает решение или функцию переключательного типа, имеющую один вход и ряд альтернативных выходов, из которых только один может быть активизирован после вычисления условий, определенных внутри этого символа. Соответствующие результаты вычисления могут быть записаны по соседству с линиями, отображающими эти пути.
	Специальные символы	
Соединитель	0.5a	Символ отображает выход в часть схемы и вход из другой части этой схемы и используется для обрыва линии и продолжения ее в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение.
Терминатор	b 0.5a	Символ отображает выход во внешнюю среду и вход из внешней среды (начало или конец программы, источник или пункт назначения данных).
Комментарий	a	Символ используют для добавления описательных комментариев или пояснительных записей в целях объяснения или примечаний. Пунктирные линии в символе комментария связаны с соответствующим символом или могут обводить группу символов. Текст комментариев или примечаний должен быть помещен около ограничивающей фигуры.

Линии потока, соединяющие блоки и указывающие последовательность связей между ними, должны быть проведены параллельно вертикальным или горизонтальным краям чертежа.

Стрелка в конце линии может не ставиться, если линия направлена слева направо или сверху вниз. При смене направления потока стрелка ставится обязательно. Если две или более линии объединяются в одну линию, место объединения должно быть смещено. Линии в схемах должны:

1) подходить к символу сверху, а исходить — снизу;
либо
2) подходить — слева, а исходить — справа.

Схему алгоритма следует выполнять как единое целое, однако, в случае необходимости допускается обрывать линии, соединяющие блоки. При обрыве линии используется специальный символ соединитель – окружность диаметром **0,5а**. Внутри соединителей одной линии указывается один и тот же идентификатор, например цифра.

Если схема занимает более одного листа, то ссылки к страницам могут быть приведены совместно с символом комментария для их соединителей. В символе комментарий на листе схемы, где линия прерывается, указывается «к стр.2», на листе схемы, где эта линия продолжается «из стр.1».

Блок схема алгоритма обязательно открывается символом (терминатором) «**начало**», завершается блоком (терминатором) «**конец**» и чертится вертикально. Если она не помещается в один столбец, то ее можно разорвать, поставив в местах разрыва соединители.

Рекомендации по выполнению курсового проекта

Технология проектирования веб-сайта состоит из нескольких этапов. Результаты каждого этапа проектирования должны быть продемонстрированы преподавателю.

На защите курсового проекта по дисциплине «Веб-программирование» студенту необходимо продемонстрировать преподавателю разработанный веб-сайт и сдать отчет, оформленный в соответствии указанными в данном пособии требованиями.

При реализации web-сайта следует использовать практические навыки и теоретические знания, полученные при выполнении лабораторных работ по данной дисциплине, а также уметь пользоваться справочными сведениями к лабораторным работам, интернет-источниками.

Перечислим основные этапы проектирования веб-сайта:

1. Выбор конкретной темы курсового проекта — это выбор конкретной организации (фирмы или компании), для которой будет разрабатываться сайт. **Тему обязательно согласовать с преподавателем**.
2. Формулирование технического задания на курсовой проект в соответствии с выбранной темой (содержание технического задания представлено в соответствующем разделе данного пособия – стр.60).

3. Проектирование карты сайта (см. стр.61).
4. Проектирование базы данных сайта (см. стр.61).
5. Проектирование шаблонов страниц сайта (см. стр.61).
6. Проектирование общей структуры модулей (см. стр.61).
7. Разработка блок-схем алгоритмов модулей сайта (см. стр.61)
8. Разработка базы данных сайта. Заполнение базы данных тестовыми данными.
9. Реализация компоновки, оформления страниц сайта. Создание ролей пользователей сайта. Написание, отладка программного кода, обеспечивающего функционирование сайта. Заполнение базы данных проекта в соответствии с техническим заданием. Тестирование сайта.
10. Демонстрация реализованных функциональных возможностей разработанного в соответствии с техническим заданием веб-сайта.

СПИСОК РЕКОМЕНДУЕМОЙ УЧЕБНОЙ ЛИТЕРАТУРЫ

1. ASP.NET Core MVC с примерами на C# для профессионалов / Адам Фримен, 6-е изд.: Пер. с англ. – СПб.: ООО "Альфа-книга", 2017. – 992 с.– ISBN 978-5-9908910-4-3
2. C# 6.0. Справочник. Полное описание языка / Албахари Джозеф, Албахари Бен, 6-е изд.: Пер. с англ. – М.: ООО "И.Д. Вильямс", 2016. – 1040 с. ISBN 978-5-8459-2087-4

СПИСОК БИБЛИОГРАФИЧЕСКИХ ИСТОЧНИКОВ

1. ASP.NET Core MVC с примерами на C# для профессионалов / Адам Фримен, 6-е изд. : Пер. с англ. – СПб.: ООО "Альфа-книга", 2017. – 992 с. – ISBN 978-5-9908910-4-3
2. Антонов И.В., Бруттан Ю.В. Технология проектирования Интранет-приложений. Проектирование web-приложений на основе ASP.NET. Учебно-методическое пособие. Псков: Изд-во ПсковГУ, 2014. – 72с. – ISBN 978-5-91116-322-8

ПРИЛОЖЕНИЕ А

Пример титульного листа

Псковский государственный университет

кафедра «Информационные системы и технологии»

Контрольная работа №1

по учебной дисциплине

ВЕБ-ПРОГРАММИРОВАНИЕ

Выполнил: Иванов А.А.

Группа: 0083-02

Проверил: Бруттан Ю.В.

Псков

2017

ПРИЛОЖЕНИЕ Б

Пример карты сайта

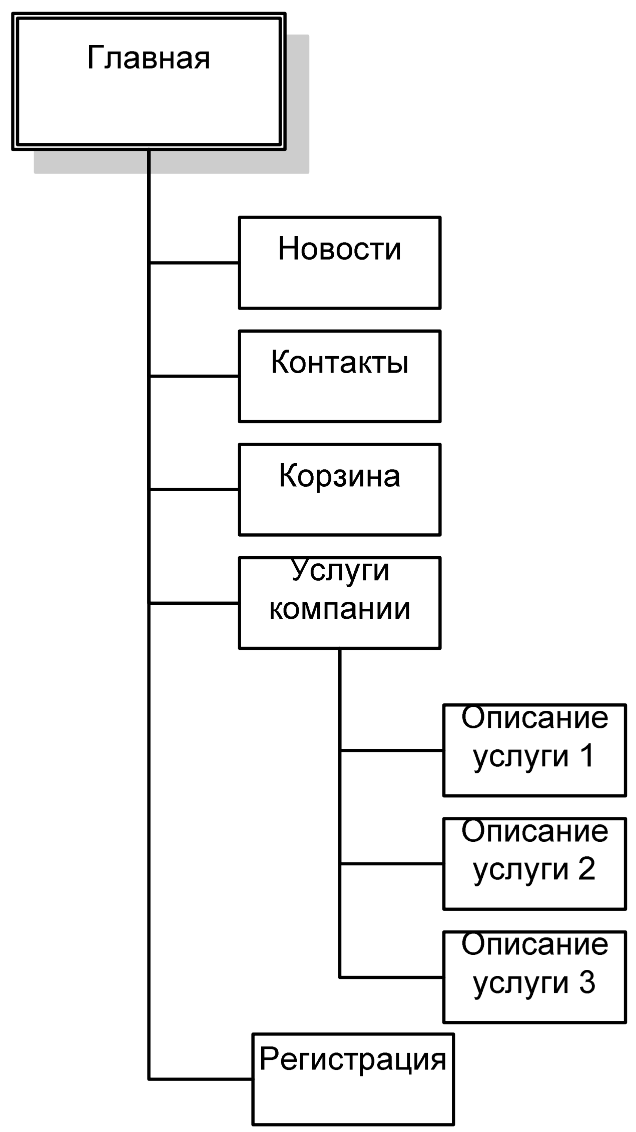

Рис. П.1. Карта сайта

Для заметок

***Антонов** Игорь Вадимович*

***Бруттан** Юлия Викторовна*

ВЕБ-ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Технический редактор: Ю.В. Бруттан
Компьютерная верстка: Ю.В. Бруттан
Корректор: С.Н. Емельянова

Подписано в печать 24.01.2018 г. Формат 60×90/16.
Гарнитура Times New Roman. Усл. п.л. 4,5.
Тираж 100 экз. Заказ №5454.

Изготовлено на Versant 2100.

Адрес издательства:
Россия, 180000, Псков, ул. Л. Толстого, д. 4[а], корп. 3[а].
Издательство Псковского государственного университета