

Разработка Web приложений с использованием ASP.NET MVC

СОДЕРЖАНИЕ

Лабораторная работа 1. Введение в разработку ASP.NET MVC Web Application	3
Упражнение 1. Создание веб-приложения ASP.NET MVC на основе контроллера.....	3
Упражнение 2. Передача параметров в контроллер	7
Упражнение 3. Реализация взаимодействия контроллера и модели	7
Лабораторная работа 2. Создание веб-приложения ASP.NET MVC	22
Упражнение 1. Реализация взаимодействия контроллера и представления.....	22
Упражнение 2. Реализация взаимодействия модели, представления и контроллера в шаблоне MVC	25
Упражнение 3. Реализация модели-репозитория.....	29
Лабораторная работа 3. Разработка представлений.....	32
Упражнение 1. Создание формы для ввода данных с помощью HTML Helpers.....	32
Упражнение 2. Применение шаблонных страниц	33
Упражнение 3. Использование CSS	35
Лабораторная работа 4. Разработка модели.....	36
Упражнение 1. Создание приложения с реализацией хранения данных	36
Упражнение 2. Настройка работы с данными.....	39
Лабораторная работа 5. Применение контролеров для формирования шаблонов данных	44
Упражнение 1. Добавление шаблонного контроллера.....	45
Упражнение 2. Применение аннотации данных в модели.....	48
Упражнение 3. Применение кеширования	49
Лабораторная работа 6. Создание интерактивных страниц в ASP.NET MVC.....	50
Упражнение 1. Использование AJAX и частичных страниц.....	50
Лабораторная работа 7. Реализация авторизации и аутентификации в приложении ASP.NET MVC 5	53

Упражнение 1. Использование типа аутентификации Individual User Accounts	54
Упражнение 2. Ограничение входа для не зарегистрированных пользователей.....	54
Упражнение 3. Создание ролей и их использование для разграничения доступа.....	55
Литература	58

Лабораторная работа 1. Введение в разработку ASP.NET MVC Web Application

В этой работе вы создадите простой проект на основе контроллера и модели, изучите основные элементы веб-приложения. Более подробно язык программирования C# и шаблон MVC будет рассматриваться в следующих работах.

Упражнение 1. Создание веб-приложения ASP.NET MVC на основе контроллера

В этом упражнении вы создадите веб-приложение, содержащее только контроллер, который будет содержать метод действия передающий информацию в браузер для ее отображения.

1. Запустите Visual Studio.
2. Выберите **File** → **New Project**.
3. В открывшемся диалоговом окне **New Project** (Создание проекта) выберите в разделе Visual C# шаблоны Web, в списке версий .NET Framework оставьте значение по умолчанию и выберите тип проекта ASP.NET Web Application (как показано на рисунке 1.1).

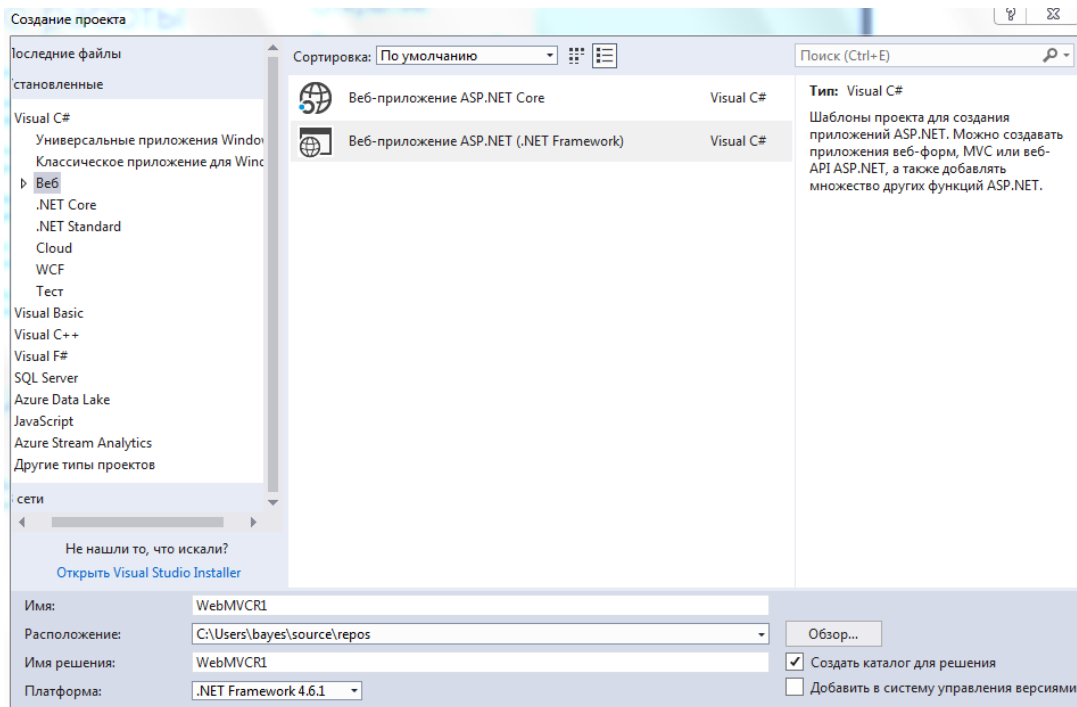


Рис.1.1 Создание проекта

4. Назовите новый проект **WebMVCRI** и нажмите кнопку ОК, чтобы продолжить.

5. В следующем окне выбора типа шаблона (см. рис. 1.2) укажите **Empty** (Пустой) и в перечне флажков добавления папок и ссылок выберите **MVC** – будет создан проект с базовой структурой папок, но без файлов, необходимых для создания MVC приложений.

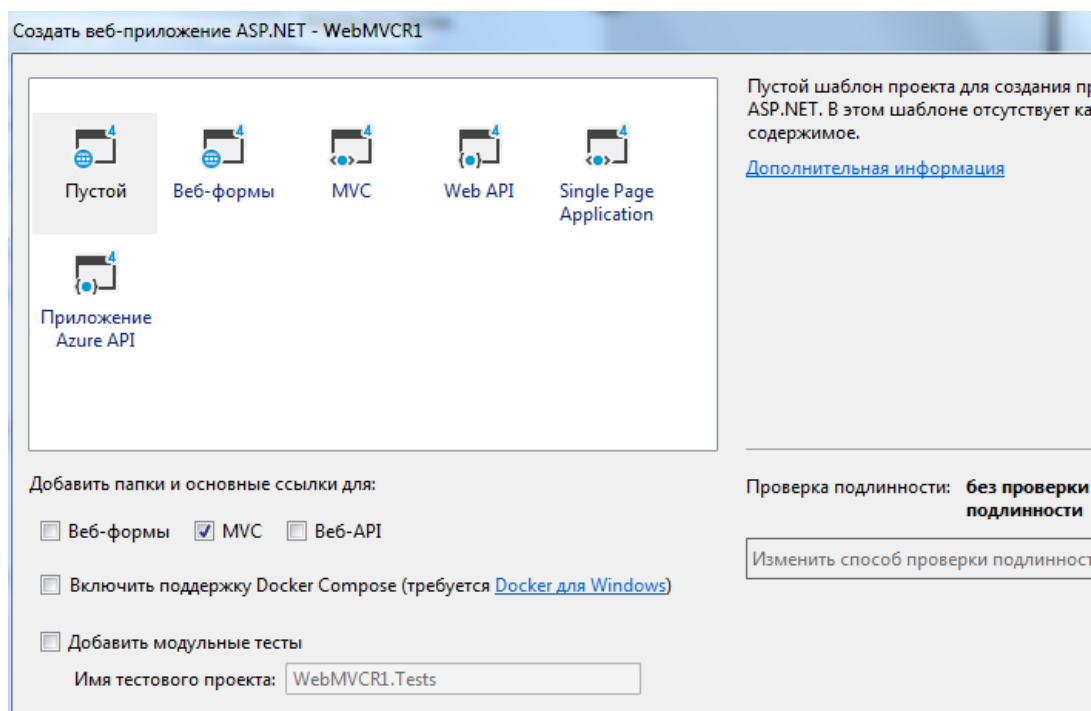


Рис. 1.2 Выбор шаблона приложения

6. В окне **Solution Explorer** (Обозреватель решений) изучите стандартную структуру MVC проекта. Поскольку был выбран пустой шаблон проекта и приложение ничего не содержит, запускать приложение пока что бессмысленно, будет ошибка 404 Not Found.

Создание контролера

В архитектуре MVC входящие запросы обрабатываются контроллерами, которые являются классами (как правило, наследуются от `System.Web.Mvc.Controller`). В MVC контроллеры находятся в папке **Controllers**, которая была создана при создании проекта.

Каждый открытый метод в контроллере представляет собой метод действия, то есть его можно вызвать из Интернет через некоторый URL, чтобы выполнить действие.

В этом упражнении вы создадите в контроллере метод действия, который возвратит в браузер строку.

Для добавления пустого контроллера выполните следующие действия:

1. В контекстном меню папки **Controllers** выберите команду **Add** (Добавить) → **Controller** (Контроллер).
2. В открывшемся окне добавления шаблона выберите пустой контроллер (см. рис. 1.3) и добавьте его.

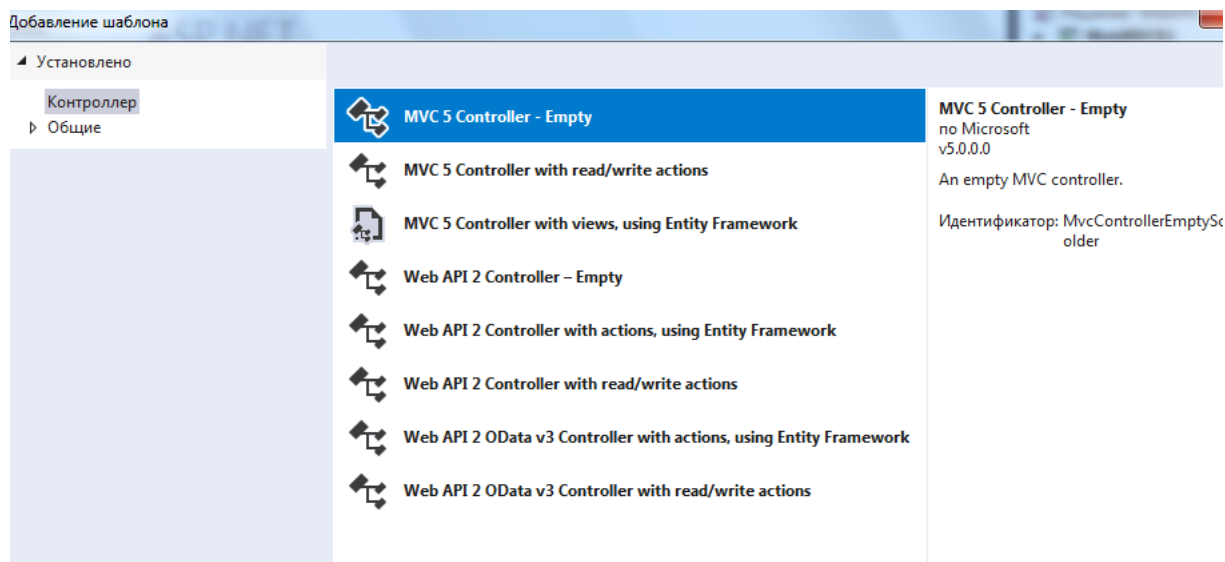


Рис. 1.3 Выбор шаблона контроллера

- В окне **Add Controller** (Добавление контроллера) укажите имя контроллера **HomeController** (рис. 1.4) и нажмите кнопку добавления контроллера:

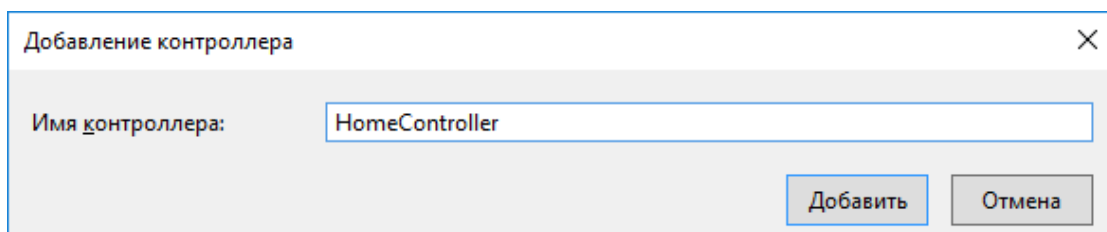


Рис. 1.4 Указание имени контроллера

- Изучите код класса **HomeController**, обратите внимание на метод действия `Index()`.
- Для тестирования контроллера закомментируйте метод `Index()` и добавьте другой вариант метода, который возвращает строку приветствия в зависимости от времени дня:

```
public string Index()
{
    int hour = DateTime.Now.Hour;
    string Greeting = hour < 12 ? "Доброе утро" : "Добрый
день";
    return Greeting;
}
```

Операция `x ? y : z` вычисляет значение `y`, если значение `x` равно `true`, и значение `z`, если значение `x` равно `false`.

- Запустите проект, выбрав в меню **Debug** команду **Start Debugging**. Браузер должен отобразить результат метода действия `Index()` – строку.

Тестирование маршрутизации

MVC-приложения используют систему маршрутизации (роутинговую систему) ASP.NET, которая решает, как URL-адреса понимают конкретные контроллеры и

действия. При создании MVC-проекта среда разработки ASP.NET добавляет некоторые роуты по умолчанию.

В отличие от традиционных приложений ASP.NET, URL-адреса MVC не соответствуют физическим файлам. Каждый метод действия имеет свой URL, и MVC использует систему маршрутизации ASP.NET чтобы перевести эти URL в действия.

1. Для тестирования маршрутизации введите в адресной строке браузера ссылки, направленные на метод `Index` контроллера **HomeController** и после каждой обновите страницу:

```
localhost:port/Home  
localhost:port//Home/Index
```

2. Проверьте, что в обоих случаях браузер отображает ту же самую строку приветствия.

Это объясняется тем, что, когда браузер запрашивает адрес сайта, он получает выходные данные метода `Index` контроллера **HomeController**.

3. Переименуйте имя класса контроллера, например, вместо `HomeController` укажите `MyController`.
4. Повторно запустите приложение, в этом случае будет ошибка, так как система маршрутизации не нашла нужный маршрут.
5. Для исправления ошибки откройте находящийся в папке `App_Start` файл `RouteConfig` и исправьте значение маршрута по умолчанию:

```
public class RouteConfig  
{  
    public static void RegisterRoutes(RouteCollection routes)  
    {  
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
        routes.MapRoute(  
            name: "Default",  
            url: "{controller}/{action}/{id}",  
            defaults: new { controller = "My", action = "Index",  
id = UrlParameter.Optional }  
        );  
    }  
}
```

6. Повторно запустите приложение, в этом случае откроется требуемая страница, так как система маршрутизации нашла нужный маршрут.
7. Теперь измените в классе контроллера имя метода `Index()` на, например, `Start()` и внесите соответствующие коррективы в класс **RouteConfig**.
8. Запустите приложение и проверьте ее работу.

Упражнение 2. Передача параметров в контроллер

В этом упражнении вы узнаете, как передавать параметры контроллеру с помощью URL-адреса и строки запроса.

При отправке GET-запроса значения передаются через строку запроса. Стандартный get-запрос принимает примерно следующую форму:

`/метод?параметр1=значение1&параметр2=значение2`

1. Откройте класс **HomeController**.
2. Измените метод `Index()`, добавив передаваемый ему строковый параметр, а в теле метода присвойте строковой переменной сообщение с передаваемым параметром:

```
public string Index(string hel)
{
    int hour = DateTime.Now.Hour;
    string Greeting = hour < 12 ? "Доброе утро" :
    "Добрый день" + ", " + hel;
    return Greeting;
}
```

3. Запустите приложение. Должна загрузиться страница `Home`.
4. Для проверки работы метода действия, принимающий параметр измените URL на следующий адрес (`port` – номер вашего порта) и учтите, что теперь имя контроллера – `MyController`:

`localhost:port/My/Index?hel=Иван`

Обновите страницу, должна появиться строка с переданным параметром.

Упражнение 3. Реализация взаимодействия контроллера и модели

Контроллер, реализованный в предыдущих упражнениях помимо обработки входных данных и вывода в браузер занимался и несвойственным ему действием – рассчитывал значение переменной, т.е. обрабатывал некую логику. В этом упражнении вы добавите в созданное веб-приложение, содержащее контроллер, модель, в которую будет перенесена логика приложения.

Реализация модели

Модель содержит данные и алгоритмы, которые будут определять логику приложения.

1. Добавьте в папку **Models** класс **ModelClass**.
2. Добавьте в класс статический метод и добавьте в его код расчета приветствия. В результате должно получиться следующее:

```
public class ModelClass
{
    public static string ModelHello()
    {
        int hour = DateTime.Now.Hour;
```

```

        string Greeting = hour < 12 ? "Доброе утро" :
"Добрый день";
        return Greeting;
    }
}

```

3. В методе контроллера `Index(string hel)` присвойте переменной строкового типа результат вызова метода модели `ModelHello()`:

```

public string Index(string hel)
{
    string Greeting = ModelClass.ModelHello() + ", " + hel;
    return Greeting;
}

```

4. Запустите приложение. Протестируйте работу метода действия, принимающий параметр, функциональность не должна измениться.

Лабораторная работа 2. Основы языка C#

Для изучения основ языка программирования вы будете использовать созданный ранее контроллер, метод действия которого будет возвращать результат в браузер.

1. Добавьте в папку **Models** класс **StudyCsharp**, который будет реализовывать все последующие действия по изучению языка.

Упражнение 1. Создание и использование размерных типов данных

В этом упражнении вы создадите перечисление для представления различных типов банковских счетов (например, `Checking` и `Deposit`). Вы создадите структуру, которую можно использовать для моделирования банковских счетов: переменные для хранения номеров счетов (тип данных `int`), балансов счетов (тип данных `decimal`) и типов счетов (перечисление).

Создание перечисления

Перечисляемый тип представляет собой тип значений, содержащий конечное число именованных констант. Ниже объявляется и используется перечисляемый тип `AccountType`, содержащий два постоянных значения `Checking` и `Deposit`.

1. Откройте файл `StudyCsharp.cs` и перед описанием класса добавьте перечисление `AccountType` со значениями `Checking` и `Deposit`:

```

namespace WebMVC1.Models
{
    public enum AccountType
    {
        Checking,
        Deposit
    }

    public class StudyCsharp
    {

```



```
    }
}
```

- В контроллере HomeController объявите новый метод ExeEnum() и объявите две переменные типа AccountType:

```
public string ExeEnum()
{
    AccountType goldAccount;
    AccountType platinumAccount;
```

- Далее присвойте первой переменной значение Checking, а второй – Deposit:

```
goldAccount = AccountType.Checking;
platinumAccount = AccountType.Deposit;
```

- Объявите две строковые переменные, с помощью метода Format() сформируйте строки результата и верните их в браузер:

```
string res1 = String.Format("Тип банковского счета {0}",
goldAccount);
string res2 = String.Format("Тип банковского счета {0}",
platinumAccount);
string res = res1 + "<p>" + res2;
return res;
```

Тестирование перечисления

- В методе действия Index() объявите строковую переменную res и присвойте ей вызов метода ExeEnum(), замените в выражении return переменную Greeting на новую переменную res:

```
string res = ExeEnum();
return res;
```

- Запустите проект, браузер должен отобразить результат метода действия Index() – две строки с указанием типа банковских счетов.

Метод действия ExeEnum() можно было также вызвать указав в строке браузера localhost:port/Home/ExeEnum

Создание и использование структуры

Структуры представляют собой структуры данных, содержащие члены-данные и члены-функции (методы структуры). Структуры имеют тип значений и не требуют выделения памяти в куче. Переменная типа структуры непосредственно хранит данные структуры.

- В файле StudyCsharp.cs после перечисления добавьте **public** (общедоступную) структуру **BankAccount**, содержащую следующие поля:

Модификатор доступа	Тип	Имя переменной
public	long	accNo

public	decimal	<i>accBal</i>
public	AccountType	<i>accType</i>

В итоге структура должна выглядеть следующим образом:

```
public struct BankAccount
{
    public long accNo;
    public decimal accBal;
    public AccountType accType;
}
```

2. В контроллере HomeController объявите новый метод ExeStruct() и в нем объявите переменную типа **BankAccount**:

```
public string ExeStruct()
{
    BankAccount goldBankAccount;
```

3. Далее присвойте значения полям *accNo*, *accBal* и *accType* переменной goldAccount:

```
goldBankAccount.accType = AccountType.Checking;
goldBankAccount.accBal = (decimal)3200.00;
goldBankAccount.accNo = 123;
```

4. Сформируйте строку информации о счете и верните ее в браузер:

```
string res = String.Format("Номер счета {0}, баланс {1},  
тип {2}", goldBankAccount.accNo, goldBankAccount.accBal,  
goldBankAccount.accType);

return res;
```

Тестирование структуры

1. В методе действия Index() закомментируйте строковую переменную res, которая вызывала метод перечисления, объявите новую переменную res и присвойте ей вызов метода ExeStruct():

```
string res = ExeStruct();
return res;
```

2. Запустите проект, браузер должен отобразить результат метода действия Index() – строку с указанием информации о банковском счете.

Переопределение метода ToString()

Любой тип языка C# является производным типом от типа **Object**, в котором определен метод ToString(), возвращающий строковое представление объекта. Для производных типов данный метод рекомендуется переопределить, указав правило представления объекта в виде строки.

1. В структуру **BankAccount** добавьте переопределение (ключевое слово override) метода ToString(), в котором укажите способ преобразования:

```
public struct BankAccount
```

```

    {
        public long accNo;
        public decimal accBal;
        public AccountType accType;

        public override string ToString()
        {
            string res = String.Format("Номер счета {0},  
баланс {1}, тип {2}", accNo, accBal, accType);
            return res;
        }
    }

```

2. В методе `ExeStruct()` контроллера `HomeController` замените выражение вычисления строки `res` на следующее выражение, использующее преобразование объекта в строку:

```

string res = String.Format("Информация о банковском  
счете: {0}", goldBankAccount);

```

3. Запустите проект, браузер должен отобразить результат метода действия `Index()` – строку с указанием информации о банковском счете согласно логике метода `ToString()`.

Использование C# версии 6.0 для формирования строки

Начиная с версии 6.0 для формирования строки допускается функция интерполяции строк, позволяющая внедрить выражения в строку формата – строка начинается с символа `$`, а затем в кавычках указываются данные, например:

```

string str = $"Номер счета {accNo}, баланс {accBal}, тип  
{accType}";

```

Примените новый синтаксис к указанному выше коду и протестируйте его работу. В дальнейшем вы можете использовать любой вариант синтаксиса на ваше усмотрение.

Упражнение 2. Создание и использование методов

В этом упражнении вы реализуете различные алгоритмы с помощью методов. Обратите внимание на реализуемые в этих методах управляющие конструкции (ветвление, циклы).

1. Откройте файл `StudyCsharp.cs`.

Реализация выбора

Оператор `if` выбирает оператор для выполнения на основании значения логического выражения.

2. В классе `StudyCsharp` объявите статический метод, принимающий целочисленный параметр и рассчитывающий в зависимости от значения этого параметра статус разработчика:

```

public static string SetStatus(int age)

```

```

{
    string status = "junior developer";

    if ((age > 2) && (age < 7)) status = "middle
developer";
    else if ((age >= 7) && (age < 15)) status = "senior
developer";
    else if ((age >= 15)) status = "sensei";

    return status;
}

```

3. В методе Index() котроллера HomeController укажите выражение вычисления строки res, использующее вызов метода SetStatus() (ранее вычисляемую переменную res закомментируйте):

```
string res = StudyCsharp.SetStatus(3);
```

4. Запустите проект, браузер должен отобразить результат метода действия Index() – строку с указанием информации о вычисленном статусе.

Оператор switch выбирает для выполнения список операторов, метка которого соответствует значению **switch**-выражения.

1. В классе StudyCsharp объявите статический метод, принимающий строковый параметр (статус) и определяющий в зависимости от значения этого параметра возможные действия:

```

public static string ExeSwitch(string status)
{
    string res;
    switch (status)
    {
        case "junior developer":
            res = "Набирайся знаний"; break;
        case "middle developer":
            res = "Набирайся опыта"; break;
        case "senior developer":
            res = "Руководи другими"; break;
        case "sensei":
            res = "Учи других"; break;
        default:
            res = "Не знаю, что делать";
            break;
    }

    return res;
}

```

2. В методе Index() котроллера HomeController укажите выражение вычисления строки res, использующее вызов метода ExeSwitch(), принимающий в качестве параметра метод SetStatus() (ранее вычисляемую переменную res закомментируйте):

```
string res = StudyCsharp.ExeSwitch(Stud
yCsharp.SetStatus(3));
```

3. Запустите проект, браузер должен отобразить результат метода действия `Index()` – строку с указанием действий в зависимости от статуса.

Реализация цикла – расчет значения функции на интервале

4. В классе `StudyCsharp` объявите статический метод, принимающий два вещественных параметра (границы интервала) и возвращающий значения функции в виде строки:

```
public static string GetFunction(double x1, double x2)
{
    StringBuilder str = new StringBuilder();
    double x = x1;
    do
    {
        str.AppendFormat("x = {0:0.##} : y = {1:0.##};<br>", x, Math.Pow(x, 3));
        x = x + 0.5;
    }
    while (x <= x2);

    return str.ToString(); ;
}
```

Обратите внимание на использование для построения результата объект класса **StringBuilder**, применяемого для изменяемых строк и использующий метод `AppendFormat()`.

5. В методе `Index()` контроллера `HomeController` укажите выражение вычисления строки `res`, использующее вызов метода `GetFunction()` (ранее вычисляемую переменную `res` закомментируйте):

```
string res = StudyCsharp.GetFunction(0, 9);
```

6. Запустите проект, браузер должен отобразить результат метода действия `Index()` – строку с расчетными значениями.

Использование возвращаемых параметров в методах

В этом задании вы создадите метод **Factorial()**, принимающий целочисленную переменную и рассчитывающий ее факториал по итерационному алгоритму.

1. В классе `StudyCsharp` объявите статический метод:
 - a. принимающий два параметра (первый параметр типа **int** передается по значению (это число, для которого рассчитывается факториал), второй параметр типа **out int** используется для возвращения результата)
 - b. возвращающий значения типа **bool**, отражающее успешность выполнения метода, так как может произойти переполнение и выброс исключения:

```
public static bool Factorial(int n, out int answer)
{
```

```
}
```

2. Внутри метода напишите код расчета факториала для передаваемого на вход значения:

- a. Объявите три переменных: целочисленные `k` (будет использоваться в цикле в качестве счетчика) и `f` (будет использоваться внутри цикла, присвойте начальное значение 1), а также булевого типа `ok` (будет использоваться для отслеживания ошибок):

```
int k;  
int f = 1;  
bool ok = true;
```

- b. Создайте цикл **for**. Начальное значение `k = 2`, итерации продолжаются до тех пор, пока не будет достигнуто значение параметра `n`. На каждом шаге значение `k` увеличивается на единицу. В теле цикла `f` умножается на `k` и сохраняется результат в `f`. Значение факториала растет достаточно быстро, поэтому реализуйте проверку на арифметическое переполнение в блоке **checked**:

```
checked  
{  
    for (k = 2; k <= n; ++k)  
    {  
        f = f * k;  
    }  
}
```

- c. Реализуйте перехват возможных исключений, все исключения обрабатываются одинаково: обнуляется результат и переменной `ok` присваивается **false**:

```
try  
{  
    checked  
    {  
        for (k = 2; k <= n; ++k)  
        {  
            f = f * k;  
        }  
    }  
}  
catch (Exception)  
{  
    f = 0;  
    ok = false;  
}
```

- d. Итоговое значение переменной `f` присвойте возвращаемому параметру `answer`:

```
answer = f;
```

```
return ok;
```

Если метод отработал успешно, он возвращает значение **true**, если произошло арифметическое переполнение (выброс исключения), то возвращается значение **false**.

Тестирование метода расчета факториала

1. В контроллере HomeController объявите новый метод, принимающий целочисленную переменную *x* для расчета факториала:

```
public string ExeFactorial(int x)
{
```

2. В этом методе объявите две переменные:

- a. переменную *f* типа **int** для хранения факториала числа,
- b. переменную *ok* типа **bool** и присвойте ей результат вызова метода **Factorial()**, передав число *x* и переменную *f* в качестве параметров:

```
int f;
bool ok = StudyCsharp.Factorial(x, out f);
```

3. Реализуйте возврат из метода значения переменных *x* и *f* в случае, если переменная *ok* принимает значение **true**. В противном случае передайте сообщение об ошибке:

```
if (ok)
    return String.Format("Факториал числа {0} равен {1} ",
x, f);
else
    return "Невозможно вычислить факториал";
```

4. В методе Index() контроллера HomeController укажите выражение вычисления строки *res*, использующее вызов метода для расчета факториала числа 5 – ExeFactorial(5) (переменную *res* ранее вычисляемую функцию GetFunction закомментируйте):

```
string res = ExeFactorial(5);
```

5. Запустите проект, браузер должен отобразить результат метода действия Index() – строку с указанием информации о вычисленном значении факториала.

Упражнение 3. Реализация класса

В этом упражнении вы создадите класс – треугольник, описываемый сторонами и имеющий периметр и площадь.

1. В пространстве имен namespace WebMVCRI.Models в любом месте (например, после класса StudyCsharp) объявите класс Triangle:

```
public class Triangle
{
}
}
```

2. В классе Triangle объявите автоматические свойства, определяющие стороны треугольника:

```
public class Triangle
{
    public double Sta { get; set; }
    public double Stb { get; set; }
    public double Stc { get; set; }
```

3. Далее реализуйте еще три свойства – имя, периметр и площадь. Все эти свойства должны быть определены только для чтения поскольку зависят от значений сторон:

```
public string Name
{
    get { return String.Format("\"Треугольник со
сторонами {0}, {1} и {2}\"", Sta, Stb, Stc); }
}

public double Perimeter
{
    get
    {
        double p = Sta + Stb + Stc;
        return p;
    }
}

public double Area
{
    get
    {
        double sq = Math.Sqrt(Perimeter / 2 * (Perimeter / 2 -
Sta) * (Perimeter / 2 - Stb) * (Perimeter / 2 - Stc));
        return sq;
    }
}
```

Использование C# версии 6.0 для формирование свойств только для чтения

Начиная с версии 6.0 сократить синтаксис, написав член класса, представляющий выражение с применением оператора “=>”. Например, для свойства периметра и имени новый синтаксис будет выглядеть следующим образом:

```
public double Perimeter => Math.Round(Sta + Stb + Stc);
public string Name => $"\"Треугольник со сторонами
{Sta},{Stb} и {Stc}\"";
```

Примените новый синтаксис к указанному выше коду и протестируйте его работу. В дальнейшем вы можете использовать любой вариант синтаксиса на ваше усмотрение (это работает для методов и свойств, доступных только для чтения).

4. Реализуйте в классе конструктор с тремя параметрами – сторонами треугольника:


```

public Triangle(double a, double b, double c)
{
    Sta = a;
    Stb = b;
    Stc = c;
}

```

Тестирование класса

1. В контроллере HomeController объявите новый метод ExeTriangle() и в нем создайте объект – треугольник с заданными сторонами – 3, 5, 6:

```

public string ExeTriangle()
{
    Triangle tr1 = new Triangle(3, 5, 6);

```

2. Далее в этом методе выведите на экран значения имени объекта и его площади:

```

        string sql = String.Format("Площадь фигуры {0}
равна: {1:0.##}", tr1.Name, tr1.Area);

        return sql;
    }

```

3. В методе действия Index() закомментируйте строковую переменную res, которая вызывала предыдущий метод, объявите новую переменную res и присвойте ей вызов метода ExeTriangle():

```

        string res = ExeTriangle();

```

4. Запустите проект, браузер должен отобразить результат метода действия Index() – строку с указанием информации о треугольнике.

Реализация класса – Окружность

1. В пространстве имен namespace WebMVC.R1.Models в любом месте (например, после класса Triangle) объявите класс Circle, реализующий окружность, определяемую радиусом.
2. В классе объявите автоматическое свойство, определяющее радиус.
3. Далее реализуйте еще три свойства – имя, длина окружности и площадь. Все эти свойства должны быть определены только для чтения поскольку зависят от значения радиуса.
4. В классе реализуйте конструктор с одним параметром – радиусом окружности.
5. Реализация класса в итоге может быть следующей:

```

public class Circle
{
    public double St{ get; set; }

    public string Name
    {

```

```

        get { return String.Format("\"Окружность с
радиусом {0}\"", St; }
    }

    public Circle(double a)
    {
        St = a;
    }

    public double Dlina
    {
        get
        {
            double p = 2 * Math.PI * St;
            return p;
        }
    }

    public double Area
    {
        get
        {
            double sq = Math.PI * St*St;
            return sq;
        }
    }
}

```

6. Постройте приложение.

Тестирование класса

1. В контроллере HomeController объявите новый метод ExeCircle() и в нем создайте объект – окружность с заданным радиусом – 3.
2. Далее в этом методе выведите на экран значения имени объекта и его площади.
3. Реализация метода в итоге может быть следующей:

```

public string ExeCircle()
{
    Circle cir1 = new Circle(3);
    string sq = String.Format("Площадь фигуры {0} равна:
{1:0.##}", cir1.Name, cir1.Area);
    return sq;
}

```

4. В методе действия Index() закомментируйте строковую переменную res, которая вызывала предыдущий метод, объявите новую переменную res и присвойте ей вызов метода ExeCircle():

```

string res = ExeCircle();

```

5. Запустите проект, браузер должен отобразить результат метода действия Index() – строку с указанием информации об окружности.

Упражнение 4. Наследование и реализация полиморфизма

В этом упражнении вы создадите на основе классов предыдущего упражнения иерархию – наследование, создадите базовый класс, в котором объявите функциональность, наследуемую производными классами.

1. В пространстве имен namespace WebMVC.R1.Models в любом месте (например, после класса Circle) объявите класс Shape, реализующий фигуру.
2. В классе объявите автоматическое свойство, определяющее сторону:

```
public class Shape
{
    public double St { get; set; }.
```

3. Далее объявите свойство – имя, которое укажите виртуальным (виртуальные свойства и методы в производных классах могут быть переопределены):

```
public class Shape
{
    public double St { get; set; }

    virtual public string Name
    {
        get { return String.Format("\"Фигура\"); }
    }
}
```

4. Внесите изменения в класс окружности:

- a. Укажите, что класс Circle наследуется от класса Shape:

```
public class Circle : Shape
{
```

- a. Удалите объявление свойства стороны – радиуса. Теперь сторона St будет наследоваться у базового класса Shape.
- b. Укажите, что версия свойства Name класса окружности переопределяет базовую версию, добавив ключевое слово override:

```
    override public string Name
    {
        get { return String.Format("\"Окружность с радиусом {0}\", St); }
    }
}
```

5. Внесите изменения в класс треугольника:

- a. Укажите, что класс Triangle наследуется от класса Shape:

```
public class Triangle : Shape
{
```

- c. Удалите объявление свойства стороны – Sta. Теперь эта сторона будет наследоваться у базового класса Shape.
- d. Переименуйте имя переменной Sta на St во всем коде класса.

- е. Укажите, что версия свойства Name класса треугольника переопределяет базовую версию, добавив ключевое слово override:

```
override public string Name
{
    get { return String.Format("\"Треугольник со
сторонами {0}, {1} и {2}\"", St, Stb, Stc); }
}
```

Тестирование полиморфизма

1. В контроллере HomeController объявите новый метод ExePolim().
2. Создайте объект класса изменяемых строк:

```
StringBuilder str = new StringBuilder();
```

3. Далее в этом методе объявите массив типа Shape, но инициализируйте его объектами производных классов, например:

```
Shape[] sh = {
    new Triangle(1, 2, 3),
    new Circle(5),
    new Triangle(5, 6, 8)
};
```

4. В цикле foreach реализуйте обращение к массиву и вызовите для каждого элемента массива свойство Name (так как оно в базовом классе объявлено виртуальным и переопределено в производных классах, то вызов будет полиморфным):

```
foreach (Shape item in sh)
{
    str.AppendFormat("Это фигура {0}", item.Name + "<p>");
}
```

5. В выражении return укажите сформированную строку:

```
return str.ToString();
```

6. В методе действия Index() закомментируйте строковую переменную res, которая вызывала предыдущий метод, объявите новую переменную res и присвойте ей вызов метода ExePolim():

```
string res = ExePolim();
```

7. Запустите проект, браузер должен отобразить результат метода действия Index() – три строки с указанием информации о фигурах.

Упражнение 5. Применение коллекций для группировки объектов

В этом упражнении вы создадите для коллекцию для организации объектов. В отличие от массивов, группа объектов в коллекции может динамически возрастать и сокращаться в соответствии с потребностями приложения.

Коллекция является классом, поэтому необходимо объявить новую коллекцию перед добавлением в неё элементов.

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен `System.Collections.Generic` (типизированные классы коллекций). Такая коллекция обеспечивает безопасность типов, так что другие типы данных не могут быть в нее добавлены. При извлечении элемента из этой коллекции нет необходимости определять или преобразовывать его тип данных.

В созданной в этом упражнении коллекции можно будет сортировать объекты. Для задания свойства объекта по которому будет выполняться сортировка требуется реализовать специальный интерфейс. В библиотеке классов определено множество стандартных интерфейсов, задающих желаемую функциональность объектов. В частности, интерфейс **Comparable** задает метод сравнения объектов по принципу больше и меньше, что заставляет переопределить соответствующие операции для объектов класса.

1. Перейдите в файл `StudyCsharp.cs` и для класса `Circle` добавьте реализацию интерфейса `Comparable<Circle>`:

```
public class Circle : Shape, Comparable<Circle>
{ ...
```

Интерфейс **Comparable** и содержит единственный метод **CompareTo**, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра. Реализация данного метода должна возвращать 0 – если текущий объект и параметр равны; отрицательное число, если текущий объект меньше параметра; положительное число, если текущий объект больше параметра. Этот метод является только определением, и для его использования необходимо реализовать его в определенном классе. Смысл, вкладываемый в понятия "меньше, чем", "равно" и "больше, чем", зависит от конкретной реализации.

2. Добавьте к классу `Circle` реализацию метода `CompareTo()` для сравнения объектов по площади:

```
public int CompareTo(Circle other)
{
    if (this.Area == other.Area) return 0;
    else if (this.Area > other.Area) return 1;
    else return -1;
}
```

3. В контроллере `HomeController` объявите новый метод `ExeCollection()`.
4. В этом методе создайте коллекцию – список однотипных объектов – окружностей, используя класс **List<T>**, заполненный некоторым количеством окружностей:

```
List<Circle> cirs = new List<Circle>
{
    new Circle(12),
```

```

        new Circle(5),
        new Circle(15),
        new Circle(6)
    };

```

5. С помощью метода **Add(T item)** добавьте в список новую окружность:

```
cirs.Add(new Circle(7));
```

6. С помощью метода **Sort()** отсортируйте коллекцию:

```
cirs.Sort();
```

7. Реализуйте вывод в браузер содержимое отсортированной коллекции:

```

StringBuilder str = new StringBuilder();
foreach (Shape item in cirs)
{
    str.AppendFormat("Это фигура {0}", item.Name + "<p>");
}

return str.ToString();

```

8. В методе действия **Index()** закомментируйте строковую переменную **res**, которая вызывала предыдущий метод, объявите новую переменную **res** и присвойте ей вызов метода **ExeCollection()**:

```
string res = ExeCollection();
```

9. Запустите проект, браузер должен отобразить результат метода действия **Index()** – строки с указанием информации об окружностях, отсортированные по площади.
10. Аналогичным образом реализуйте коллекцию треугольников с возможностью сортировки по их периметру.
11. Протестируйте внесенные изменения.

Лабораторная работа 3. Создание веб-приложения ASP.NET MVC

В этой работе вы добавите в проект WebMVCR1 представление, модель и реализуете взаимодействие с ними в рамках шаблона MVC.

Упражнение 1. Реализация взаимодействия контроллера и представления

Результатом всех предыдущих упражнений является текстовая строка. Для того чтобы создать на запрос браузера HTML ответ, необходимо создать представление.

Настройка контроллера

1. Снимите комментарий для метода **Index()**, возвращающий **ActionResult** и закомментируйте метод, возвращающий строку.
2. Измените тип возвращаемого параметра метода **Index()** на **ViewResult**:

```

public ViewResult Index()
{
    return View();
}

```

Работа контроллера состоит в том, чтобы 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**, который является членом базового класса **Controller**. **ViewBag** – это динамический объект, которому можно присвоить произвольные свойства, что делает эти значения доступными для любого представления, которое будет дальше их использовать.

Другим способом является использование свойства **ViewData**, который предоставляет экземпляр класса **ViewDataDictionary**. Для передачи данных в представление нужно сначала добавить его в свойство контроллера **ViewData** в методе действия, который используется для отображения представления.

3. В новом методе реализуйте функциональность приветствия, результат проверки времени дня присвойте свойству динамического объекта **ViewBag** (имя свойства может быть произвольным, так как свойство не существует до того момента, пока ему не будет присвоено значение), а свойству присвойте **ViewData** произвольное значение:

```
public IActionResult Index()
{
    int hour = DateTime.Now.Hour;
    ViewBag.Greeting = hour < 12 ? "Доброе утро" :
"Добрый день";
    ViewData["Mes"] = "хорошего настроения";
    return View();
}
```

Создание представления

1. Для добавления представления для метода действия `Index` в контекстном меню этого метода (см. рис. 3.1) выберите команду **Add View** (Добавить представление):

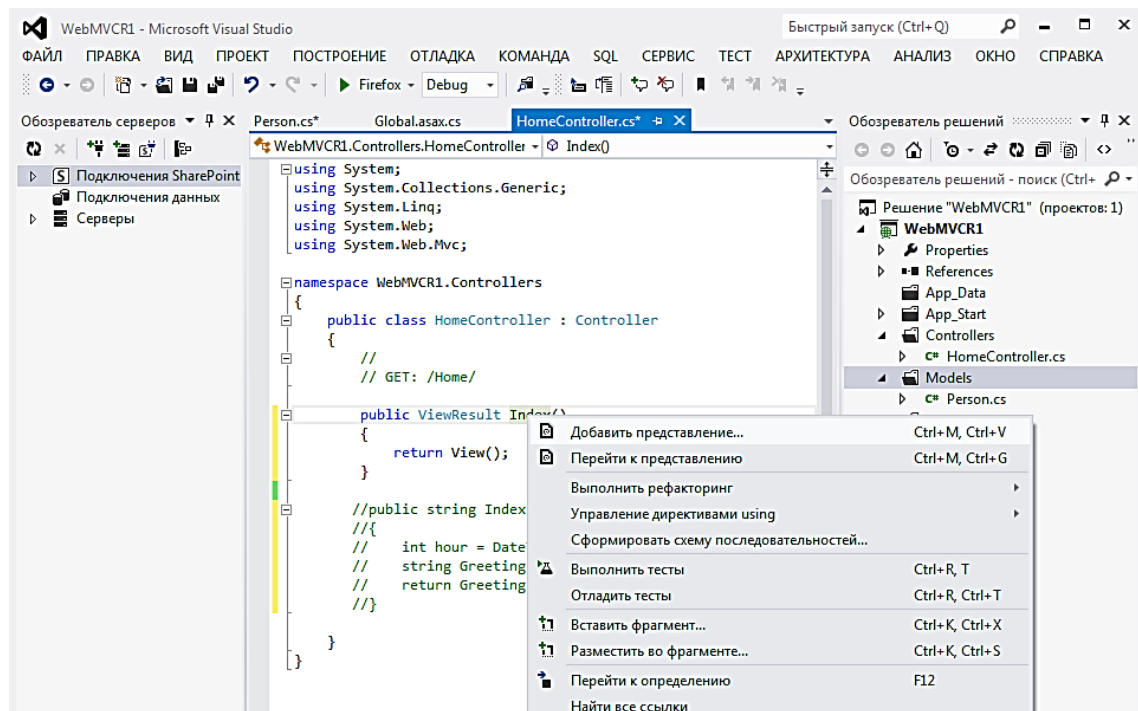


Рис. 3.1 Команды добавления представления

- В окне добавления представления оставьте имя – **Index**, в списке шаблона укажите **Empty** (без модели) и снимите флажок *Use a layout page* (Использовать страницу макета):

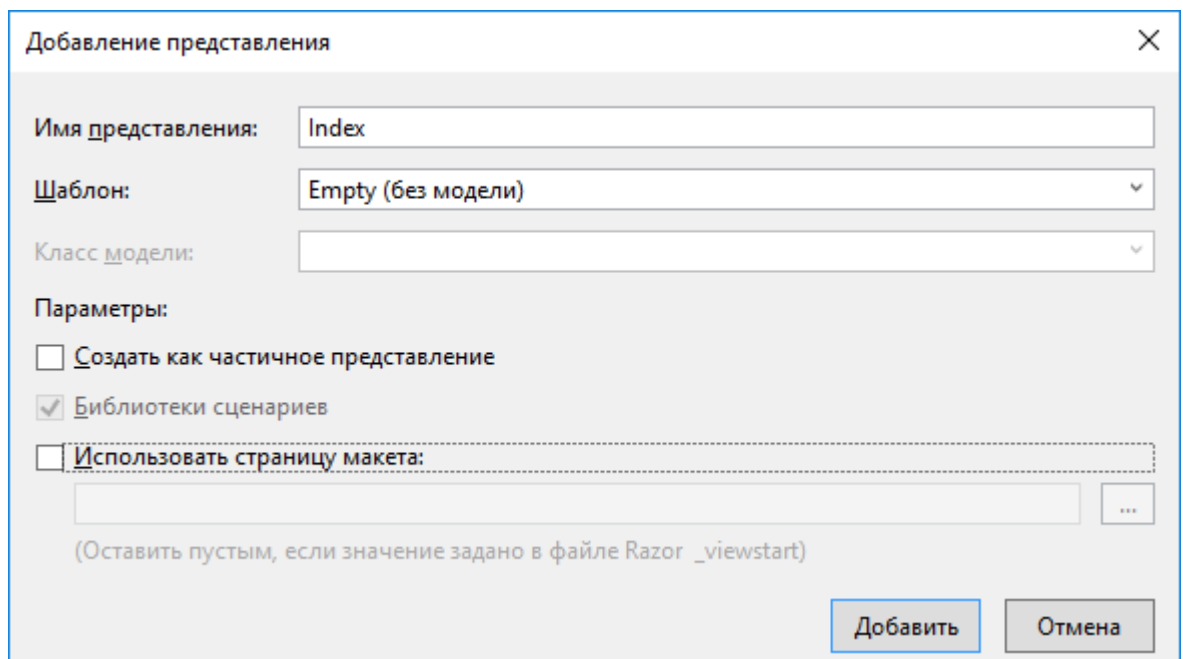


Рис. 3.2 Добавление представления

- После добавления представления откроется файл Index.cshtml для редактирования. Внутри тега `<div>` добавьте текст для вывода:

```
<div>
    @ViewBag.Greeting, спасибо, что зашли и @ViewData["Mes"]
</div>
```


4. Запустите приложение. Обработка выражения обозначает вставку значения, которое динамически присвоилось ViewBag.Greeting и ViewData["Mes"] метода действия в представление, в итоге должна открыться страница с указанным текстом.

Упражнение 2. Реализация взаимодействия модели, представления и контроллера в шаблоне MVC

В этом упражнении вы добавите основные элементы шаблона MVC – модель и представление, а также настроите контроллер для взаимодействия с ними в веб-приложение ASP.NET MVC.

Реализация модели

Данные, с которыми будет работать клиент, должны быть представлены моделями. Для этого упражнения модель будет определяться классом, описывающим персону с двумя свойствами – имя и фамилия.

1. Добавьте в папку **Models** класс **Person**, который будет реализовывать модель данных о человеке.
2. Добавьте в класс поля, описывающие имя и фамилию персоны, а также переопределите метод ToString():

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override string ToString()
    {
        string s = FirstName + " " + LastName;
        return s;
    }
}
```

3. Постройте приложение.

Связь главной формы и формы ввода данных

Ввод имени персоны будет выполняться на другой форме, на которую можно будет перейти из главной формы.

1. Для реализации возможности перехода на форму ввода (она будет создана позднее) с помощью метода ActionLink (принимающего два параметра: первый – текст для отображения в ссылке, например, "Введите имя", а второй – выполняемое действие InputData, когда пользователь нажимает на ссылку) добавьте ссылку на нее из представления Index.cshtml:

```
<div>
    @ViewBag.Greeting, спасибо, что зашли и
    @ViewData["Mes"]
<p>
```

```
@Html.ActionLink("Введите свои данные", "InputData")
</p>
</div>
```

2. Запустите приложение. Проверьте отображение ссылки.

Настройка контроллера

3. В файл контроллера HomeController добавьте метод действия InputData, которому будет соответствовать адрес формы:

```
public ActionResult InputData()
{
    return View();
}
```

Добавление строго типизированного представления

Для метода действия InputData необходимо добавить строго типизированное представление, которое будет обрабатывать определенный тип (модель).

1. Для добавления представления для метода действия InputData в контекстном меню этого метода выберите команду **Add View** (Добавить представление).
2. В окне добавления представления (см. рис. 3.3) оставьте имя – **InputData**, в списке *Шаблон* укажите **Empty**, в списке *Класс модели* выберите класс **Person (WebMVC1.Models)** и снимите флажок *Use a layout page* (Использовать страницу макета):

Рис. 3.3 Добавление строго типизированного представления

После добавления представления откроется файл InputData.cshtml для редактирования.

3. Обратите внимание на первую строку

```
@model WebMVC1.Models.Person
```

Razor-оператор `@model` определяет тип модели, с которым связано представление. В этом упражнении будет применяться обычная разметка для ввода данных. Позднее будут рассмотрены специальные элементы (html-хелперы) для ввода информации.

Создание формы для ввода данных

1. В файле `InputData.cshtml` внутри тега `<div>` добавьте текст для ввода данных с помощью запроса `POST` (`<form method="post">`) и реализуйте передачу с запросом `post` данных о модели с помощью полей ввода `<input type="text" name="FirstName"/>` и `<input type="text" name="LastName"/>` (их свойства `name` соответствуют именам свойств модели **Person** и при нажатии кнопки и отправки запроса передаются значения этих полей):

```
<div>
    <form method="post">
        <table>
            <tr>
                <td><p>Введите имя:</p></td>
                <td><input type="text" name="FirstName" /> </td>
            </tr>
            <tr>
                <td><p>Введите фамилию:</p></td>
                <td><input type="text" name="LastName" /> </td>
            </tr>
            <tr><td><input type="submit" value="Отправить" /></td><td></td></tr>
        </table>
    </form>
</div>
```

2. Запустите приложение. Проверьте работу ссылки.

Обработка данных на форме

Для получения и обработки отправленных данных формы применяются два метода действия, которые вызываются одним и тем же URL, но MVC гарантирует, что будет вызван соответствующий метод в зависимости от того, какой был запрос – GET или POST:

- метод, который отвечает на HTTP GET запросы: GET запрос является тем, с чем браузер имеет дело после каждого клика по ссылке. Этот вариант действий будет отвечать за отображение начальной пустой формы, когда кто-то первый раз посетит `/Home/InputData`.
- метод, который отвечает на HTTP POST запросы. Запросы по умолчанию, формы отправляются браузером как POST запросы. Этот вариант действий

будет отвечать за получение отправленных данные и решать, что с ними делать.

1. В класс `HomeController` импортируйте пространство имен

```
using WebMVC1.Models;
```

2. Добавьте атрибут `HttpGet` для существующего метода действия `InputData`, это означает, что данный метод должен использоваться только для GET запросов:

```
[HttpGet]
public ActionResult InputData()
{
    return View();
}
```

3. Добавьте перегруженную версию метода действия `InputData`, который принимает параметр `Person` и применяет атрибут `HttpPost`, это означает, что новый метод будет иметь дело с POST запросами:

```
[HttpPost]
public ActionResult InputData(Person p)
{
    return View("Hello", p);
}
```

Данный вариант перегруженного метода действий `InputData` показывает, как можно указать MVC обрабатывать конкретное представление, а не представление по умолчанию, в ответ на запрос. Этот вызов метода `View` говорит MVC найти и обработать представление, которое называется `Hello` и передать представлению объект `p`.

4. Создайте представление `Hello` – для этого щелкните правой кнопкой мыши внутри одного из методов `HomeController` и выберите команду **Add View** (Добавить представление) (см. рис. 3.4):
 - a. Установите имя представления – `Hello`.
 - b. В списке **Шаблон** укажите `Empty`.
 - c. В списке **Класс модели** выберите класс `Person (WebMVC1.Models)`.
 - d. Снимите флажок *Use a layout page* (Использовать страницу макета).

Рис. 3.4 Добавление представления для ответа

После добавления представления откроется файл Hello.cshtml для редактирования.

5. Внутри тега `<div>` добавьте текст для вывода данных свойства модели – имени и ссылку на главную страницу:

```
<div>
    <h1>Здравствуйте, @Model.ToString()!</h1>
    <p>
        @Html.ActionLink("На главную", "Index")
    </p>
</div>
```

6. Запустите и протестируйте работу приложения.

Упражнение 3. Реализация модели-репозитория

В этом упражнении вы создадите модель – репозиторий для хранения коллекции персон, настроите контроллер и создадите представление для вывода списка персон.

Добавление модели - репозитория

1. Добавьте в папку **Models** класс **PersonRepository**, который будет реализовывать модель коллекции.
2. Добавьте в класс поле, описывающее коллекцию – список персон:

```
public class PersonRepository
{
    private List<Person> persons = new List<Person>();
```

3. Далее в классе укажите два свойства только для чтения – первое должно возвращать количество персон, второе – саму коллекцию:

```
public int NumberOfPerson
```

```

    {
        get
        {
            return persons.Count();
        }
    }

    public IEnumerable<Person> GetAllResponses
    {
        get
        {
            return persons;
        }
    }
}

```

4. Для возможности добавления персон в коллекцию реализуйте в классе советующий метод:

```

    public void AddResponse(Person pers)
    {
        persons.Add(pers);
    }

```

5. Постройте приложение.

Настройка контроллера

1. В поле класса контроллера **HomeController** добавьте код создания статического объекта репозитория:

```
private static PersonRepository db = new PersonRepository();
```

2. В перегруженную версию метода действия `InputData`, который принимает параметр `Person` и применяет атрибут `HttpPost` укажите код добавления объекта `Person` в коллекцию `db` (выделено жирным шрифтом):

```

[HttpPost]
public ActionResult InputData(Person p)
{
    db.AddResponse(p);
    return View("Hello", p);
}

```

3. Добавьте новый метод действия, реализующий получение коллекции и информации о ее размере в динамический объект `ViewBag`:

```

public ActionResult OutputData()
{
    ViewBag.Pers = db.GetAllResponses;
    ViewBag.Count = db.NumberOfPerson;
    return View("ListPerson");
}

```

В данном случае предполагается вывод списка персон на другую страницу.

Добавление строго типизированного представления для отображения коллекции

1. Создайте представление `ListPerson` – для этого щелкните правой кнопкой мыши внутри метода `OutputData()` и выберите команду **Add View** (Добавить представление) (см. рис. 3.5):
 - a. Установите имя представления на `ListPerson`,
 - b. В списке **Шаблон** укажите `Empty`.
 - c. В списке **Класс модели** выберите класс `PersonRepository` (`WebMVCRI.Models`)
 - d. Снимите флажок *Use a layout page* (Использовать страницу макета).

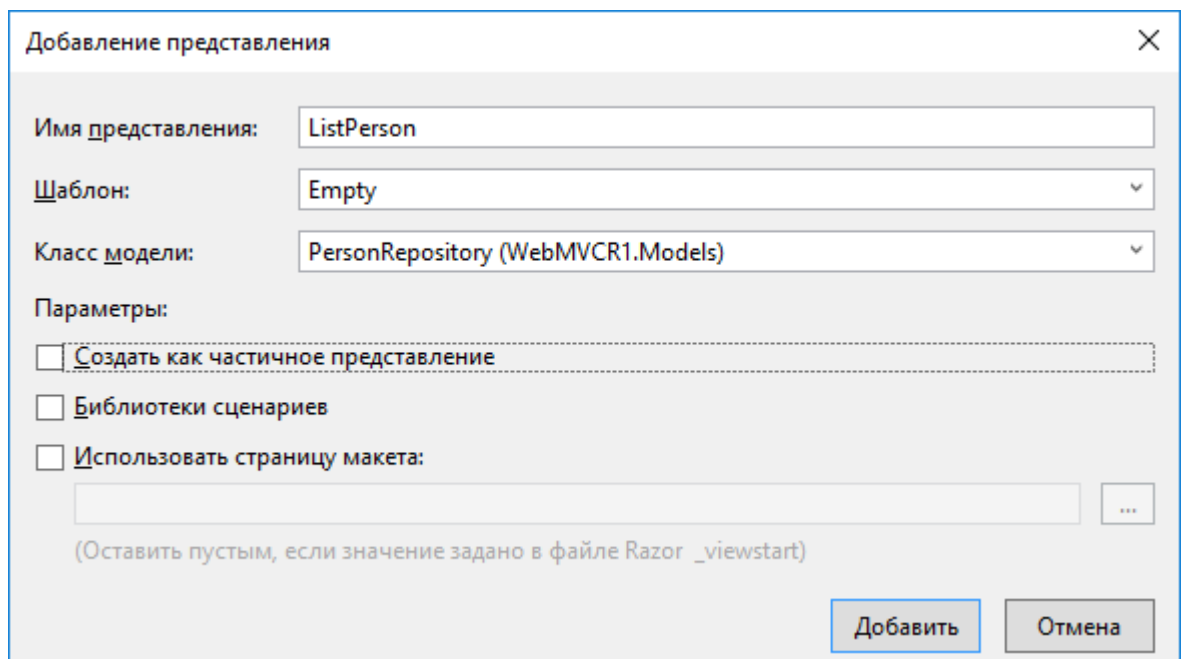


Рис. 3.5 Добавление представления для вывода данных

После добавления представления откроется файл `ListPerson.cshtml` для редактирования.

2. Внутри тега `<div>` добавьте текст для вывода данных свойства модели – имени и ссылку на главную страницу:

```
<div>
    <h3>Список участников</h3>
    <p>Персон всего @ViewBag.Count</p>

    <ul>
        @foreach (var c in ViewBag.pers)
        {
            <li>@c.ToString() </li>
        }
    </ul>

    <p>
        @Html.ActionLink("На главную", "Index")
    </p>
</div>
```

3. Для возможности просмотра списка персон (перехода на страницу после заполнения коллекции) добавьте в файл Index.cshtml соответствующую ссылку:

```
<p>
    @Html.ActionLink("Посмотрите, кто уже здесь",
"OutputData")
</p>
```

4. Постройте приложение. Запустите и протестируйте его работу. Введите данные о нескольких человек и просмотрите их список, перейдя по ссылке, добавленной в предыдущем пункте.

Лабораторная работа 4. Разработка представлений

В этой работе вы разработаете представление с использованием шаблонных страниц и таблицы стилей.

Упражнение 1. Создание формы для ввода данных с помощью HTML Helpers

В этом упражнении вы примените HTML хэлперы для генерации HTML разметки.

1. Откройте страницу InputData.cshtml. Внутри тега <div> замените разметку для ввода данных с помощью метода-хелпера Html.BeginForm и двух html-хелперов для ввода информации пользователем (второе поле использует лямбда-выражение):

```
@using (Html.BeginForm())
{
    <table>
    <tr>
        <td><p>Введите имя:</p></td>
        <td>@Html.TextBox("FirstName")</td>
    </tr>
    <tr>
        <td><p>Введите фамилию:</p></td>
        <td>@Html.TextBoxFor(x => x.LastName)</td>
    </tr>
    <tr><td><input type="submit" value="Отправить" />
</td><td></td></tr>
    </table>
}
```

Оба варианта ввода данных идентичен, но использование лямбда-выражений в строго типизированном хелпере считается предпочтительным.

2. Запустите приложение. Проверьте работу страницы ввода данных.

Упражнение 2. Применение шаблонных страниц

В этом упражнении вы используете для шаблонные страницы (макеты) для задания общего форматирования сайта. Автономные представления, разработанные в предыдущих упражнениях удобны для простых приложений, но реальный проект может иметь множество представлений, и макеты являются эффективными шаблонами, которые содержат разметку, используемую для создания логичности и постоянства в веб приложении.

1. Откройте любой файл представления, например, `Index.cshtml` и найдите в начале разметки следующее Razor-выражение:

```
@{
    Layout = null;
}
```

Этот блок кода устанавливает значение свойства `Layout` на `null`, в результате чего представление является автономным, и оно будет показывать все содержимое, которое должно вернуться клиенту.

Создание макета

2. Для создания макета откройте контекстное меню папки **Views** в обозревателе решения, укажите команду **Add** (Добавить) и далее **New Item** (Создать новый).
3. В отрывшемся окне добавления нового элемента выберите страницу макета Razor, укажите имя `_BasicLayout.cshtml` и нажмите кнопку **Add** (Добавить).

Обратите внимание на первый символ имени – символ подчеркивания (`_`). Имена макетов, которые являются поддерживающими файлами должны начинаться с подчеркивания.

4. Изучите содержимое созданного макета, обратите внимание на вызов метода `@RenderBody`, который вставляет содержимое представления, указанного методом действия в разметку макета.
5. Добавьте следующий код для придания общей наглядности:

```
<title>@ViewBag.Title</title>
...
<body>
    <h1>ОАО "Добро пожаловать"</h1>
    <div style="padding: 20px; border: solid medium green;
font-size: 20pt">
        @RenderBody()
    </div>
</body>
```

6. Сохраните изменения.

Применение макета

1. Откройте файл представления `Index.cshtml`.

2. Чтобы применить макет к представлению установите значение макетной страницы свойству `Layout` и укажите значение для свойства `ViewBag.Title`, которое будет использоваться в качестве контента элемента `title` в HTML документе, отправленном обратно пользователю (это не обязательно, но желательно):

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
    ViewBag.Title = "Главная";
}
```

3. Удалите все элементы, которые обеспечивают структуру полной HTML страницы:

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
    ViewBag.Title = "Главная";
}

<div>
    @ViewBag.Greeting, спасибо, что зашли
    <p>
        @Html.ActionLink("Введите имя", "InputData")
    </p>
</div>
```

В итоге, то что осталось ориентировано на показе данных из объекта модели представления пользователю.

4. Запустите приложение и убедитесь, что теперь к первой странице применена стилизация шаблона.
5. Примените макет `_BasicLayout.cshtml` к остальным представлениям `Hello.cshtml`, `InputData.cshtml` и `ListPerson.cshtml`.
6. Запустите приложение и убедитесь, что теперь ко всем страницам применена стилизация шаблона.

Использование макета `_ViewStart`

Жесткая привязка к шаблону в разметке страниц бывает неудобной при практическом применении. Есть возможность для страницы пропустить указание шаблона и в таком случае будет применяться шаблон по умолчанию `_ViewStart`.

1. Добавьте в папку **Views** новый файл макета с именем `_ViewStart.cshtml`.
2. Замените его содержимое на присвоение свойству `Layout` готового макета:

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
}
```

3. Во всех представлениях удалите строку присвоения шаблона.

4. Запустите приложение и убедитесь, что ко всем страницам применена стилизация шаблона по умолчанию, а так как он ссылается на старый шаблон, то визуализация страниц не должна измениться.

Упражнение 3. Использование CSS

В этой части упражнения вы реализуете стилизацию приложения, настроив файл стилей для приложения.

1. Добавьте в проект сайта папку **Content**.
2. В папку **Content** добавьте файл **Таблица стилей** с именем Site.css.
3. Вставьте в файл код настройки стилей:

```
h1 {
    margin: 0px;
    padding: 2px 0;
    font-size: 30px;
    font-weight: bold;
}

body {
    font-family: "Trebuchet MS", Verdana, Helvetica, Sans-
    Serif;
    padding: 20px;
    border: solid medium green;
    font-size: 15pt
}

button[type=submit] {
    margin-top: 15px;
    padding: 5px;
    width: 336px;
}
```

4. В файле _BasicLayout.cshtml вставьте код подключения стилей с помощью метода `Url.Content` (путь к файлу) и удалите стиль, ранее указанный в элементе `<div>`:

```
...
<title>@ViewBag.Title</title>
<link href="@Url.Content("~/Content/Site.css")"
rel="stylesheet" type="text/css" />
...
<div>
    @RenderBody()
</div>
```

5. Запустите приложение и убедитесь, что ко всем страницам применена стилизация шаблона в соответствии с таблицей стилей.

Лабораторная работа 5. Разработка модели

Упражнение 1. Создание приложения с реализацией хранения данных

Создание проекта

В этом упражнении вы разработаете новое приложение, которое будет выводить на главной странице из базы данных информацию о кредитах, также будет реализована возможность пользователю подать заявку на получение кредита, и эта заявка сохранится в базе данных.

1. Создайте новый проект ASP.NET Web Application с именем **MvcCreditApp1**.
2. В окне выбора шаблона в разделе **Шаблоны ASP.NET** укажите **MVC**, в перечне флажков добавления папок и ссылок также выберите **MVC** – будет создан проект со структурой папок, необходимых для создания MVC приложений.
3. Изучите структуру проекта, раскройте папку **Content**, содержащую используемые стили, откройте контроллер **HomeController**, реализующий три метода действия и откройте соответствующие этим методам представления: **About**, **Contact** и **Index**, изучите их содержимое.
4. Запустите проект и протестируйте работу готового шаблона сайта.

Реализация модели

Данные, с которыми будет работать клиент, должны быть представлены моделями. Для данной задачи можно выделить две области данных: информация о кредите и информация о заявке на кредит, соответственно, необходимо создать две модели.

1. Добавьте в папку **Models** класс **Credit**, который будет реализовывать модель данных о кредите.
2. Добавьте в него код, описывающий модель кредита:

```
public class Credit
{
    // ID кредита
    public virtual int CreditId { get; set; }
    // Название
    public virtual string Head { get; set; }
    // Период, на который выдается кредит
    public virtual int Period { get; set; }
    // Максимальная сумма кредита
    public virtual int Sum { get; set; }
    // Процентная ставка
    public virtual int Procent { get; set; }
}
```

3. Добавьте в папку **Models** класс `Bid`, который будет реализовывать модель данных о заявке на кредит.
4. Добавьте в него код, описывающий модель заявки на кредит:

```
public class Bid
{
    // ID заявки
    public virtual int BidId { get; set; }
    // Имя заявителя
    public virtual string Name { get; set; }
    // Название кредита
    public virtual string CreditHead { get; set; }
    // Дата подачи заявки
    public virtual DateTime bidDate { get; set; }
}
```

Применение Entity Framework

Для доступа к данным будем использовать **Entity Framework**. Этот фреймворк позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

Entity Framework в сочетании с **LINQ** (Language-Integrated Query) представляет собой реализацию ORM (объектно-реляционное отображение (object-relational mapping — ORM) для платформы .NET Framework от компании Microsoft. **Entity Framework** содержит механизмы создания и работы с сущностями базы данных через объектно-ориентированный код на языке C#.

В этом упражнении рассматривается подход **Code-First**, при использовании которого сначала определяется модель в коде, а затем, на ее основе создается база данных. Вы создадите две таблицы, описывающие данные клиента и его кредит. Отношение между этими таблицами будет “один к одному” (one-to-one).

Entity Framework при работе с **Code First** требует определения ключа элемента для создания первичного ключа в таблице в БД. По умолчанию при генерации БД EF в качестве первичных ключей будет рассматривать свойства с именами `Id` или `[Имя_класса]Id` (т. е. `CreditId` и `BidId`).

Поскольку проект создан по шаблону **MVC**, то библиотека **Entity Framework** уже добавлена в проект.

Создание контекста данных

1. Создайте контекст данных (он нужен для облегчения доступа к БД на основе модели), для этого добавьте в папку **Models** класс `CreditContext`, унаследованный от класса `DbContext`.
2. Добавьте ссылку на пространство имен `System.Data.Entity` в файле `CreditContext.cs`:

```
using System.Data.Entity;
```

3. В классе `CreditContext` объявите соответствующие таблицы (по традиции во множественном числе) базы данных с помощью свойств с типом `DbSet`:

```
namespace MvcCreditApp1.Models
{
    public class CreditContext: DbContext
    {
        public DbSet<Credit> Credits { get; set; }
        public DbSet<Bid> Bids { get; set; }
    }
}
```

Этот класс контекста представляет полный слой данных, который можно использовать в приложениях. Благодаря **DbContext**, можно запросить, изменить, удалить или вставить значения в базу данных.

4. Постройте приложение.

Создание базы данных на основе модели

В этой части упражнения вы создадите базу данных и заполните ее значениями.

1. В папку **Models** добавьте класс `CreditsDbInitializer` унаследованный от класса `DropCreateDatabaseIfModelChanges` (при инициализации в случае изменения модели старая база будет удаляться и создаваться новая с начальными значениями), подключите требуемое пространство имен:

```
using System.Data.Entity;
...

public class CreditsDbInitializer :
    DropCreateDatabaseIfModelChanges<CreditContext>
{
}
```

2. Переопределите метод `Seed()` в котором создайте, например, три кредита и добавьте их в таблицу **Credits** с помощью метода `Add()` свойства `Credits`:

```
public class CreditsDbInitializer :
    DropCreateDatabaseIfModelChanges<CreditContext>
{
    protected override void Seed(CreditContext context)
    {
        context.Credits.Add(new Credit { Head = "Ипотечный
кредит", Period = 10, Sum = 1000000, Procent = 15 });
        context.Credits.Add(new Credit { Head = "Образовательный
кредит", Period = 7, Sum = 300000, Procent = 10 });
        context.Credits.Add(new Credit { Head = "Потребительский
кредит", Period = 5, Sum = 500000, Procent = 19 });

        base.Seed(context);
    }
}
```

3. Для генерации базы данных необходимо чтобы при запуске приложения создавался экземпляр класса **CreditsDbInitializer**. Для этого откройте файл **Global.asax** и добавьте в метод `Application_Start`, который выполняется при старте приложения, создание объекта:

```
Database.SetInitializer(new CreditsDbInitializer());
```

4. Импортируйте в файл **Global.asax** пространства имен `MvcCreditApp1.Models` и `System.Data.Entity`:

```
using MvcCreditApp1.Models;
using System.Data.Entity;
```

Упражнение 2. Настройка работы с данными

В этом упражнении вы внесете изменения в контроллеры и представления для работы с данными.

Настройка контроллера

1. Откройте находящийся в папке **Controllers** контроллер `HomeController`.
2. Импортируйте пространство имен `MvcCreditApp1.Models`.
3. В поле класса контроллера создайте экземпляр контекста данных:

```
private CreditContext db = new CreditContext();
```

4. В методе `Index()` обратитесь к контексту и получите все записи о кредитах:

```
var allCredits = db.Credits.ToList<Credit>();
```

5. Далее создайте свойство `Credits` в объекте `ViewBag` и присвойте ему извлеченный список. Объект `ViewBag` является таким объектом, который передается в представление:

```
ViewBag.Credits = allCredits;
```

6. В итоге класс контроллера с измененным методом выглядит следующим образом:

```
using MvcCreditApp1.Models;
...
public class HomeController : Controller
{
    private CreditContext db = new CreditContext();

    public ActionResult Index()
    {
        var allCredits = db.Credits.ToList<Credit>();
        ViewBag.Credits = allCredits;

        return View();
    }
}
```

Обратите внимание, что так как модели находятся в другом пространстве имен, хотя и в одном проекте, то его необходимо импортировать.

7. Постройте проект.

Изменение представления для отображения данных

1. Откройте находящиеся в папке **Home** представление Index.
2. Изучите разметку для позиционирования элементов, для отображения данных будет использован тот же принцип – в виде сетки с указанием строк и столбцов.
3. Замените сгенерированный код разметки на следующий:

```
@{
    ViewBag.Title = "Главная";
}

<div class="row">
    <div class="col-md-4">
        <h3>Программы кредитования</h3>
        <table>
            <tr><td><p>Тип кредита</p></td><td><p>Период
кредитования</p></td><td><p>Максимальная сумма</p></td><td><p>Ставка
%</p></td></tr>
            @foreach (var c in ViewBag.Credits)
            {
                <tr><td><p>@c.Head</p></td><td><p>@c.Period</p></td><td><p>@c.Sum</p><
/td><td><p>@c.Procent %</p></td></tr>
            }
        </table>
        <p><a class="btn btn-default"
href="/Home/CreateBid">Подать заявку на получение кредита
&raquo;</a></p>
    </div>
</div>
```

В этом коде создается таблица, в которой будут располагаться данные о программах кредитования. Вся информация помещается в ячейку сетки.

Конструкция `@foreach (var c in ViewBag.Credits)` использует синтаксис движка представления Razor (после символа `@` согласно синтаксису, можно использовать выражения кода на языке C#).

В цикле реализован проход по элементам в объекте `ViewBag.Credits`, который был создан в методе контроллера, получены свойства каждого элемента и помещены в таблицу.

Ссылка `href="/Home/CreateBid">Подать заявку на получение кредита` означает адрес, по которому будет размещаться форма заявки на кредит.

4. Постройте и запустите приложение. После загрузки и генерации базы данных должна отобразиться таблица с ранее определенными данными (см. рис.5.1).

Имя приложения Домашняя страница О программе Контакт			
Программы кредитования			
Тип кредита	Период кредитования	Максимальная сумма	Ставка %
Ипотечный кредит	10	1000000	15 %
Образовательный кредит	7	300000	10 %
Потребительский кредит	5	500000	19 %
Подать заявку на получение кредита »			
© 2017 – приложение ASP.NET			

Рис. 5.1 Отображение таблицы данных

Реализация подачи заявки

В этой части упражнения будет создан метод `CreateBid`, который будет отвечать за обработку ввода пользователя при подаче заявки.

1. Добавьте в контроллер **HomeController** обычный метод для получения информации о существующих кредитах и скопируйте в него код из метода `Index()`:

```
private void GiveCredits()
{
    var allCredits = db.Credits.ToList<Credit>();
    ViewBag.Credits = allCredits;
}
```

2. В методе замените `Index()` замените соответствующий код на вызов метода.
3. Добавьте в контроллер **HomeController** два метода, определяющие одно действие **CreateBid**, в первом случае оно выполняется при получении запроса GET, а во втором случае – при получении запроса POST:

- a. Первый метод `ActionResult CreateBid()` возвращает соответствующее представление с получением всех записей о кредитах и заявках:

```
[HttpGet]
public ActionResult CreateBid()
{
    GiveCredits();
    var allBids = db.Bids.ToList<Bid>();
    ViewBag.Bids = allBids;
    return View();
}
```

- b. Второй метод принимает переданную ему в запросе POST модель `newBid` и добавляет ее в базу данных. В конце возвращается строка уведомительного сообщения:

```
[HttpPost]
public string CreateBid(Bid newBid)
{
    newBid.bidDate = DateTime.Now;
    // Добавляем новую заявку в БД
    db.Bids.Add(newBid);
    // Сохраняем в БД все изменения
    db.SaveChanges();
    return "Спасибо, <b>" + newBid.Name + "</b>, за выбор  
нашего банка. Ваша заявка будет рассмотрена в течении 10 дней.";
}
```

4. Добавьте представление **CreateBid**. Для этого нажмите на метод `public ActionResult CreateBid()` правой кнопкой и добавьте в проект новое представление, имя (`CreateBid`) и остальные параметры оставьте по умолчанию, проверьте, что флажок *Использовать страницу макета* включен, но само поле пустое.
5. Добавьте в конце файла следующий код отображения существующих заявок и создания формы ввода данных:

```
<h3>Существующие заявки</h3>
<table>
    <tr><td><p>Имя</p></td><td><p>Тип кредита
</p></td></tr>
        @foreach (var c in ViewBag.Bids)
        {
            <tr><td><p>@c.Name</p></td><td><p>@c.CreditHead
</p></td></tr>
        }
    </table>
    <h3>Форма подачи заявки по программам кредитования</h3>
    <form method="post" action="">
        <table>
            <tr><td><p>Введите свое имя </p></td><td><input type="text"
name="Name" /> </td></tr>
```

```
|  |  |  |  | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Выберите из списка тип кредита :</p></td>  <select name="CreditHead"> @foreach (var cr in ViewBag.Credits) { <option>@cr.Head</option> } </select> </td> </tr> |  |  | | --- | --- | | <input type="submit" value="Отправить" /> </td><td></td></tr> </table> </form> | | | |

```

Обратите внимание на реализацию списка кредита с помощью цикла `foreach`, в дальнейшем будет реализована возможность добавления нового типа кредита и в этом случае в списке новый кредит отобразится.

6. Постройте и запустите приложение. Введите данные о заявке и нажмите кнопку "Отправить". После этого заявка попадет в базу данных, а в браузере отобразится соответствующее уведомление.
7. Вернитесь на страницу **GreateBid** и обновите ее. Проверьте, что информация о существующих заявках отобразилась на странице.

Стилизация приложения

В этом приложении по умолчанию применяется фреймворк Bootstrap. Для того, чтобы созданная вами таблица была оформлена в стиле Bootstrap, ей нужно назначить класс `table`: `<table class = "table">`.

1. Назначьте класс `table` таблицам на странице **GreateBid**.
2. Откройте находящийся в папке **Content** файл `Bootstrap.css` и добавьте в раздел `table` новый стиль (выделен жирным шрифтом):

```

table {
border-collapse: collapse;
border-spacing: 0;

border: 6px solid silver;
vertical-align: middle;
text-align: center;
}

```

3. Для таблицы на странице **Index** добавьте дополнительный класс Bootstrap `table-bordered` к базовому классу `table`: `<table class="table-bordered">`, в этом случае добавятся границы для всех ячеек таблицы.
4. В файле `_Layout.cshtml` добавьте ссылку на страницу с заявкой:

```

<ul class="nav navbar-nav">
...
<li>@Html.ActionLink("Послать заявку", "CreateBid", "Home")</li>

```


5. Запустите приложение и убедитесь, что теперь к сайту применена стилизация (см. рис. 5.2, 5.3).

[Имя приложения](#) [Домашняя страница](#) [О программе](#) [Контакт](#) [Послать заявку](#)

Программы кредитования

Тип кредита	Период кредитования	Максимальная сумма	Ставка %
Ипотечный кредит	10	1000000	15 %
Образовательный кредит	7	300000	10 %
Потребительский кредит	5	500000	19 %

Подать заявку на получение кредита »

Рис. 5.2 Применение стилизации к стартовой странице

[Имя приложения](#) [Домашняя страница](#) [О программе](#) [Контакт](#) [Послать заявку](#) [Регистрация](#) [Выполнить вход](#)

CreateBid

Существующие заявки

Имя	Тип кредита
Алексей	Ипотечный кредит
Алексей	Ипотечный кредит

Форма подачи заявки по программам кредитования

Введите свое имя

Выберите из списка тип кредита :

Ипотечный кредит ▾

Отправить

Рис. 5.3 Применение стилизации к странице подачи заявки

Лабораторная работа 6. Применение контролеров для формирования шаблонов данных

В этой работе вы реализуете работу с данными с помощью шаблонных контроллеров и представлений, использующих Entity Framework.

Упражнение 1. Добавление шаблонного контроллера

1. В папку контролеров проекта **MvcCreditApp1** добавьте новый контроллер BidsController по шаблону «MVC-контроллер с представлениями, использующий Entity Framework» (см. рис. 6.1).
2. В окне настройки контроллера (см. рис. 6.2) укажите:
 - a. Класс модели: Bid(MvcCreditApp1.Models),
 - b. Класс контекста модели: CreditContext(MvcCreditApp1.Models),
 - c. Флажок *Создать представление* включен.
 - d. Включите флажок *Использовать страницу макета* и с помощью значка троеточия в окне выбора укажите страницу макета _Layout.

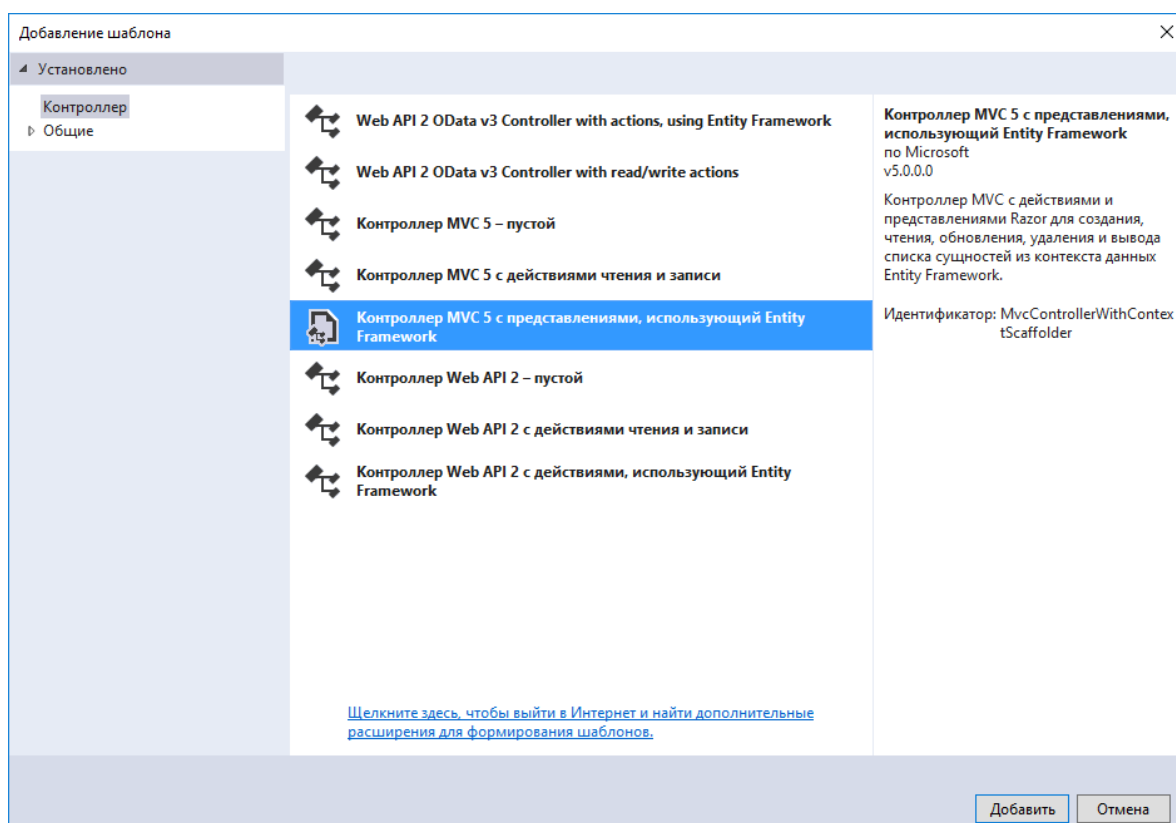


Рис. 6.1 Добавление контроллера по шаблону

Рис. 6.2 Настройка контроллера

3. Изучите содержимое сгенерированного файла BidsController.cs. Обратите внимание на реализацию методов действия.
4. Откройте в папке представлений **Views** папку **Bids**, там должны находиться сгенерированные шаблонные представления для каждого метода действия контроллера.
5. Откройте представление Index.cshtml и замените в указанных местах английский язык на русский:

```
...
    ViewBag.Title = "Заявки";

<h2>Заявки</h2>
<p>
    @Html.ActionLink("Создать новую заявку", "Create")
</p>
...
    @Html.ActionLink("Изменить", "Edit", new { id=item.BidId }) |
    @Html.ActionLink("Подробнее", "Details", new { id=item.BidId }) |
    @Html.ActionLink("Удалить", "Delete", new { id=item.BidId })
```

6. Откройте представление Create.cshtml и замените в указанных местах английский язык на русский:

```
...
    ViewBag.Title = "Создание заявки";

<h2>Создание заявки</h2>
...
```

```

<legend>Заявка</legend>
...
    <p>
        <input type="submit" value="Создать" />
    </p>

```

```
@Html.ActionLink("Вернуться в список", "Index")
```

7. Откройте представление Delete.cshtml и подкорректируйте заголовок ViewBag.Title, содержимое заголовков h2, h3, легенды legend, кнопки <input type="submit"... и ссылки @Html.ActionLink.
8. Откройте представление Details.cshtml и подкорректируйте заголовок ViewBag.Title, содержимое заголовка h2, легенды legend, ссылок @Html.ActionLink("Изменить"... и @Html.ActionLink("Вернуться в список"....
9. Откройте представление Edit.cshtml и подкорректируйте заголовок ViewBag.Title, содержимое заголовка h2, легенды legend, кнопки <input type="submit"... и ссылки @Html.ActionLink("Вернуться в список"....
10. Откройте страницу макета _Layout и добавьте ссылку на новую страницу:

```

<ul class="nav navbar-nav">
    <li>@Html.ActionLink("Главная", "Index", "Home")</li>
    <li>@Html.ActionLink("Послать заявку", "CreateBid", "Home")</li>
    <li>@Html.ActionLink("Просмотреть заявки", "Index", "Bids")</li>
</ul>

```

11. Удалите ссылку на страницу Contact.
12. Постройте и запустите приложение. Перейдите по ссылке «Просмотреть заявки». Должен открыться список заявок (названия столбцов будут изменены в следующем упражнении).
13. Протестируйте работу ссылок по созданию новой заявки, изменению существующей, просмотра подробностей и удалению заявки.
14. В папку контролеров проекта **MvcCreditApp1** добавьте новый контроллер CreditsController и аналогичным образом настройте соответствующие шаблонные представления, сгенерированные в папке **Credits**.

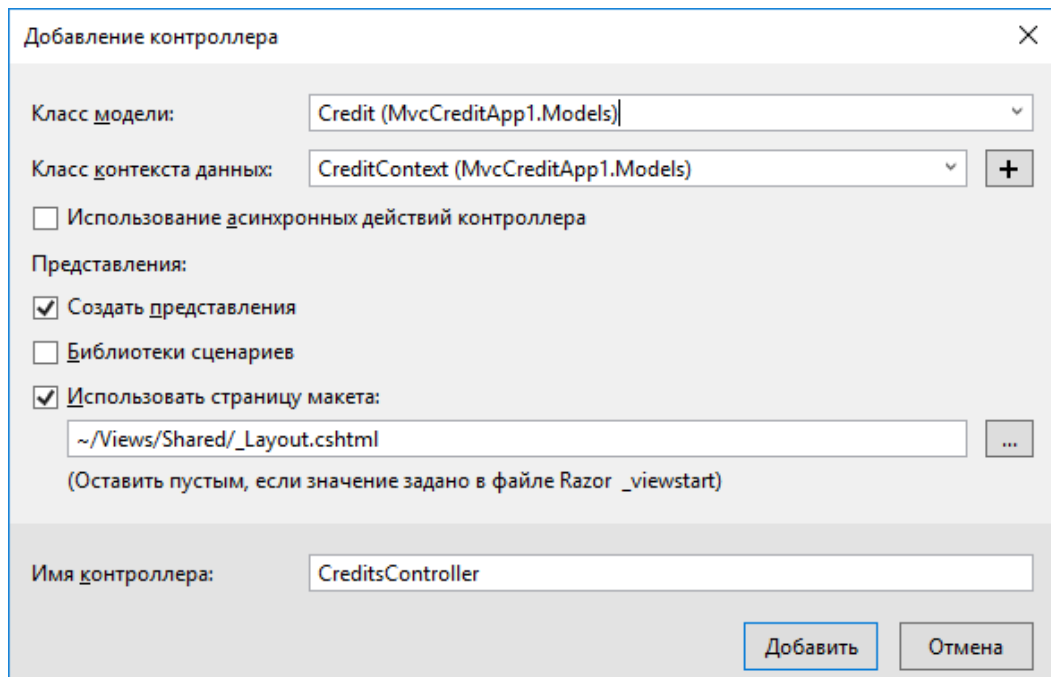


Рис. 6.3. Настройка контроллера CreditsController

15. Откройте страницу макета _Layout и добавьте еще одну ссылку на новую страницу:

```
<ul class="nav navbar-nav">
  <li>@Html.ActionLink("Главная", "Index", "Home")</li>
  <li>@Html.ActionLink("Послать заявку", "CreateBid", "Home")</li>
  <li>@Html.ActionLink("Просмотреть заявки", "Index", "Bids")</li>
  <li>@Html.ActionLink("Кредиты", "Index", "Credits")</li>
</ul>
```

16. Постройте и запустите приложение. Перейдите по ссылке «Кредиты». Должен открыться список кредитов (названия столбцов будут изменены в следующем упражнении).
17. Протестируйте работу ссылок по созданию нового кредита, изменению существующего, просмотра подробностей и удалению кредита.

Упражнение 2. Применение аннотации данных в модели

В этом упражнении вы добавите в модель MVC метаданные, которые позволяют указать некоторую дополнительную информацию об объекте, например, о том, как отображать свойства в представлении, или о том, как выполнять валидацию вводимых данных.

1. Добавьте в класс **Bid** для автоматически реализуемых свойств атрибут **DisplayName**, который содержит строку, отображаемую в представлении вместо имени свойства и атрибуты **DataType** и **DisplayFormat**:

```
public class Bid
{
    // ID заявки
    public virtual int BidId { get; set; }
```



```

        // Имя заявителя
        [DisplayName("Имя заявителя")]
        public virtual string Name { get; set; }

        // Название кредита
        [DisplayName("Название кредита")]
        public virtual string CreditHead { get; set; }

        // Дата подачи заявки
        [DisplayName("Дата подачи заявки")]
        [DataType(DataType.DateTime)]
        [DisplayFormat(DataFormatString = "{0:dd/MM/yy}")]

        public virtual DateTime bidDate { get; set; }
    }

```

2. Подключите требуемые пространства имен:

```

using System.ComponentModel.DataAnnotations;
using System.ComponentModel;

```

3. Постройте и запустите приложение. Перейдите по ссылке «Просмотреть заявки». Должен открыться список заявок и названия столбцов отображаются в нужном виде, а дата указана в требуемом формате.

4. Результат может быть примерно следующим (рис. 6.4):

Имя приложения Домашняя страница О программе Контакт Послать заявку Просмотреть заявки Кредиты			
Регистрация Выполнить вход			
Создать новую заявку			
Имя заявителя	Название кредита	Дата подачи заявки	
Алексей	Ипотечный кредит	07.05.17	Изменить Подробнее Удалить
Алексей	Ипотечный кредит	07.05.17	Изменить Подробнее Удалить
Иван	Особый	07.05.17	Изменить Подробнее Удалить

© 2017 – приложение ASP.NET

Рис. 6.4 Список заявок

- Аналогичным образом добавьте в класс **Credit** для автоматически реализуемых свойств атрибут **DisplayName**.
- Добавьте в оба класса атрибут валидации **[Required]** для свойств, которые обязательны для заполнения.
- Запустите приложение и проверьте работу валидации полей.

Упражнение 3. Применение кеширования

В этом упражнении вы реализуете механизм кеширования. С помощью кеширования можно уменьшить количество обращений к базе данных, оптимизировать обработку запросов и тем самым уменьшить нагрузку на сервер и

повысить производительность. Наиболее простой способ кэширования данных представляет использование атрибута **OutputCache**.

Данный атрибут может применяться как к всему контроллеру, так и к отдельным его методам. Применение атрибута к контроллеру позволяет задать единую политику кэширования ко всем методам данного контроллера.

1. Перейдите в Bid-контроллер и добавьте перед методом действия `Index()` атрибут **OutputCache** с параметрами `Duration` (продолжительность кэширования контента в секундах) и `Location` (место, где размещается кэшированный контент):

```
[OutputCache(Duration = 60, Location =  
OutputCacheLocation.ServerAndClient)]  
public ActionResult Index()  
{  
    ...  
}
```

2. Запустите и протестируйте работу механизма кэширования – создайте заявку, вы увидите, что при переходе на страницу списка данные не отобразились. В итоге несмотря на новые обращения к ресурсу в пределах 60 секунд вы будете получать одно и то же значение, потому что оно будет браться из кэша. И только через 60 секунд отобразятся изменения.

Лабораторная работа 7. Создание интерактивных страниц в ASP.NET MVC

В этой работе вы добавите в проект предыдущей работы возможность использования AJAX и частичного обновления страниц.

AJAX (Асинхронный JavaScript и XML) представляет собой технологию гибкого взаимодействия между клиентом и сервером. Благодаря ее использованию можно осуществлять асинхронные запросы к серверу без перезагрузки всей страницы.

Применительно к ASP.NET MVC использование AJAX выражается в концепции под названием "ненавязчивого AJAX" и ненавязчивого JavaScript (unobtrusive Ajax/JavaScript). Смысл этой концепции заключается в том, что весь необходимый код JavaScript используется не на самой веб-странице, а помещается в отдельные файлы с расширением *.js. А затем с помощью тега `<script>` реализуется в веб-странице ссылка на данный файл кода.

Упражнение 1. Использование AJAX и частичных страниц

В этом упражнении вы внедрите в приложение поддержку технологии AJAX.

Включение поддержки AJAX в проект

В проект ASP.NET Web Application не включена поддержка технологии AJAX. Поэтому перед тем как использовать **AJAX** в проекте необходимо загрузить специальный плагин **Microsoft.jQuery.Unobtrusive.Ajax**.

1. Для добавления поддержки **AJAX** с помощью менеджера пакетов **NuGet** (в контекстном меню проекта **MvcCreditApp1**) выполните команду **Управление пакетами Nuget (Manage Nuget Packages)**.
2. В появившемся диалоговом окне (см. рис. 7.1) перейдите на вкладку **Обзор** и в строку поиска введите **ajax**, а в поле **Источник пакета** выберите **nuget.org**.
3. В списке результате поиска выберите **Microsoft.jQuery.Unobtrusive.Ajax**
4. В выпадающем списке версий выберите последнюю и нажмите кнопку **Установить (Install)**.

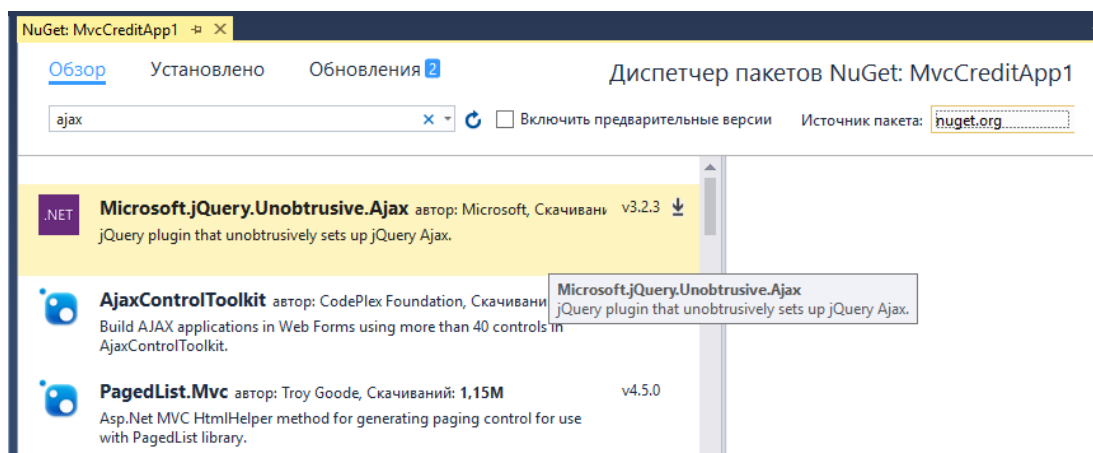


Рис. 7.1 Выбор модуля *Microsoft.jQuery.Unobtrusive.Ajax*

5. После этого появится окно с описанием лицензии на использование **jQuery.Unobtrusive.Ajax**. Согласитесь с условиями, после чего **NuGet** установит в проект **jQuery.Unobtrusive.Ajax**.
6. В окне вывода вы должны получить информацию о выполнении операции.
7. Раскройте папку **Scripts** проекта и проверьте, что требуемые скрипты добавлены.
8. Зарегистрируйте скрипт Unobtrusive AJAX в конфигурационном файле **App_Start\BundleConfig.cs**, для этого после регистрации jQuery добавьте требуемый код (выделен жирным):

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
        "~/Scripts/jquery-{version}.js"));

    bundles.Add(new ScriptBundle("~/bundles/ajax").Include(
        "~/Scripts/jquery.unobtrusive-ajax.min.js"));
}
```

9. После этого в заголовке шаблона подключите необходимые библиотеки:

```
@Scripts.Render("~/scripts/jquery-1.10.2.js")
@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.js")
```

10. При этом необходимо строго соблюдать последовательность подключения. Вначале подключается jQuery и только затем плагин (как это показано в примере выше), а не наоборот.

Реализация асинхронного метода

1. Откройте контроллер HomeController и добавьте метод действия контроллера, выполняющий асинхронную операцию «Извлечение из БД нужной информации в частичное представление»:

```
public ActionResult BidSearch(string name)
{
    var allBids = db.Bids.Where(a =>
a.CreditHead.Contains(name)).ToList();
    if (allBids.Count == 0)
    {
        return Content("Указанный кредит " + name + " не найден");
        //return HttpNotFound();
    }
    return PartialView(allBids);
}
```

2. Добавьте частичное представление BidSearch (см. рис. 7.2).

Рис. 7.2 Добавление частичного представления

3. В открывшуюся страницу частичного представления добавьте реализацию результата поиска:

```
@model IEnumerable<MvcCreditApp1.Models.Bid>
<div id="results">
    <h4> Все заявители кредита "@Model.First().CreditHead"</h4>
    <ul>
        @foreach (var item in Model)
        {
            <li>@item.Name</li>
        }
    </ul>
</div>
```

```

    }
  </ul>
</div>

```

Настройка представления, отображающее результат

Для отправки запроса на сервер применяется хелпер **Ajax.BeginForm**.

1. Откройте представление `Index` контроллера `HomeController` и в конце кода разметки добавьте следующий код:

```

<div>
    <h5>Просмотр заявителей по кредиту</h5>
</div>
<div>
    @using (Ajax.BeginForm("BidSearch", new AjaxOptions {
        UpdateTargetId = "results" }))
    {
        <input type="text" name="Name" />
        <input type="submit" value="Поиск" />
    }
    <div id="results"> </div>
</div>

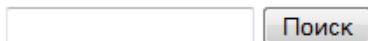
```

Результаты запроса по поиску

2. Постройте и запустите приложение. Протестируйте его работу.

Пользователь обращается к ресурсу – ожидает получить определенный ответ, например, в виде частичной веб-страницы с некоторыми данными, например:

Просмотр заявителей по кредиту



На стороне сервера метод контроллера, получая параметры, обрабатывает их и формирует некоторый ответ в виде результата действия, например:

Просмотр заявителей по кредиту



Все заявители кредита "Особый"

- Петр
- Сергей

Лабораторная работа 8. Реализация авторизации и аутентификации в приложении ASP.NET MVC 5

В этой работе вы рассмотрите применение новой системы авторизации и аутентификации в .NET приложениях – ASP.NET Identity. Эта система пришла на смену провайдерам Simple Membership, которые были введены в ASP.NET MVC 4.

Упражнение 1. Использование типа аутентификации Individual User Accounts

В этом упражнении вы используете в проекте тип аутентификации предлагаемый по умолчанию - **Individual User Accounts**. Созданный проект уже по умолчанию имеет всю необходимую для авторизации инфраструктуру: модели, контроллеры, представления.

1. Откройте проект прошлой лабораторной работы.
2. Раскройте узел **References** (Ссылки) и проверьте, что там есть ссылки на основные библиотеки, которые содержат необходимые для авторизации и аутентификации классы библиотеки OWIN (The Open Web Interface for .NET):
 - **Microsoft.AspNet.Identity.EntityFramework**: содержит классы Entity Framework, применяющие ASP.NET Identity и осуществляющие связь с SQL Serverом
 - **Microsoft.AspNet.Identity.Core**: содержит ряд ключевых интерфейсов ASP.NET Identity.
 - **Microsoft.AspNet.Identity.OWIN**: добавляет в приложение ASP.NET MVC аутентификацию OWIN с помощью ASP.NET Identity.
3. Запустите проект.
4. Нажмите на ссылку **Регистрация** (Register). На открывшейся странице создания учетной записи введите свои данные (произвольные) и нажмите кнопку регистрации.
5. Проверьте, что после регистрации логин веб-приложения.
6. Нажмите ссылку **Выйти**, отображающуюся в правом верхнем углу веб-страницы.

Упражнение 2. Ограничение входа для не зарегистрированных пользователей

В этом упражнении вы ограничите доступ к определенным страницам приложения.

Чтобы ограничить доступ к представлениям ASP.NET MVC, следует ограничить доступ к методу действия, который используется для отображения представления. Для этого платформа MVC Framework предоставляет класс **AuthorizeAttribute**. Если метод действия помечается атрибутом AuthorizeAttribute, доступ к этому методу действия ограничивается пользователями, прошедшими проверку подлинности и авторизацию.

1. Откройте файл контроллера HomeController и укажите перед методом CreateBid() атрибут [Authorize], теперь создать (отправить) заявку смогут только пользователи, прошедшие проверку подлинности и авторизацию:

```
[Authorize]
```

```
[HttpGet]
public ActionResult CreateBid()
{ ...
```

2. Запустите приложение и попробуйте без авторизации послать заявку. Вы должны быть перенаправлены на страницу авторизации.
3. Откройте файл контроллера `CreditsController` и укажите перед классом контроллера атрибут `[Authorize]`, теперь все методы действий в этом контроллере будут ограничены:

```
[Authorize]
public class CreditsController : Controller
{ ...
```

4. Запустите приложение и попробуйте без авторизации перейти на страницу *Кредиты*. Вы должны быть перенаправлены на страницу авторизации.

Упражнение 3. Создание ролей и их использование для разграничения доступа

В этом упражнении вы используете средства для управления ролями системы ASP.NET Identity. Роли позволяют создать группы пользователей с определенными правами и в зависимости от принадлежности к той или иной группе, разграничить доступ к ресурсам приложения. Вы создадите пользователя, выполняющего роль администратора и позволите только ему получить доступ к странице с кредитами.

В проекте MVC 5 каждая роль представлена объектом класса **IdentityRole**, который реализует интерфейс **IRole**. Для управления ролями предназначен класс **RoleManager**, который использует в качестве хранилища ролей объект **RoleStore**.

Задействуем систему ролей в проекте. Например, нам надо, чтобы у нас в базе данных уже был один пользователь, выполняющий роль админа. Итак, проинициализируем базу данных начальными значениями для ролей и пользователей.

1. Добавьте в папку **Models** новый класс `AppDbInitializer`:

```
public class AppDbInitializer :
DropCreateDatabaseAlways<ApplicationDbContext>
{
```

2. Переопределите метод `Seed()`, в котором создайте объекты `UserManager` и `RoleManager` с помощью контекста данных для управления пользователями и ролями:

```
protected override void Seed(ApplicationDbContext context)
{
    var userManager = new ApplicationUserManager(new
UserStore<ApplicationUser>(context));

    var roleManager = new RoleManager<IdentityRole>(new
RoleStore<IdentityRole>(context));
```

3. Далее создайте две роли с именами `admin` и `user`:

```
var role1 = new IdentityRole { Name = "admin" };
var role2 = new IdentityRole { Name = "user" };
```

4. Добавьте роли в базу данных:

```
roleManager.Create(role1);
roleManager.Create(role2);
```

5. Создайте пользователя-администратора с указанием соответствующих параметров:

```
var admin = new ApplicationUser { Email =
"admin@mail.ru", UserName = "admin@mail.ru" };
string password = "qwerty_311";
var result = userManager.Create(admin, password);
```

6. В случае успешного создания пользователя-администратора добавьте для него роли:

```
if (result.Succeeded)
{
    userManager.AddToRole(admin.Id, role1.Name);
    userManager.AddToRole(admin.Id, role2.Name);
}
```

```
base.Seed(context);
```

```
}
```

```
}
```

7. Для инициализации базы данных добавьте ее вызов в файл Global.asax в метод App_Start (при запуске приложения в базе данных окажется один пользователь и две роли):

```
protected void Application_Start()
{
    Database.SetInitializer<ApplicationDbContext>(new
AppDbInitializer());
```

8. Установите разграничение доступа по ролям на уровне контроллеров или методов, например, откройте файл контроллера CreditsController и измените стоящий перед классом контроллера атрибут [Authorize] так, чтобы все методы действий в этом контроллере были разрешены только для администратора:

```
[Authorize (Roles="admin")]
public class CreditsController : Controller
{ ...
```

9. Запустите приложение. Войдите под именем администратора, логин: admin@mail.ru, пароль: qwerty_311. Теперь вы можете перейти на страницу *Кредиты* и выполнять на ней все действия с кредитами.

Использование роли user

В прошлом примере была создана роль "user", теперь требуется, чтобы при регистрации всем пользователям присваивалась эта роль.

1. Откройте файл контроллера **AccountController**.
2. Найдите метод `Register()` и добавьте в условие проверки создания пользователя код добавления роли пользователя (выделено жирным шрифтом):

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email,
Email = model.Email };
        var result = await UserManager.CreateAsync(user,
model.Password);
        if (result.Succeeded)
        {
            await UserManager.AddToRoleAsync(user.Id, "user");
            await SignInManager.SignInAsync(user,
isPersistent: false, rememberBrowser: false);
            ...
        }
    }
}
```

3. С помощью метода `GetRoles()` класса **UserManager** можно получить набор ролей определенного пользователя. Добавьте следующий код в метод действия `About()` контроллера **HomeController**:

```
public ActionResult About()
{
    ViewBag.Message = "Your application description page.";
    IList<string> roles = new List<string> { "Роль не
определена" };
    ApplicationUserManager userManager =
HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>();
    ApplicationUser user =
userManager.FindByEmail(User.Identity.Name);
    if (user != null)
        roles = userManager.GetRoles(user.Id);
    ViewBag.rol = roles;

    return View();
}
```

4. Подключите в файл контроллера требуемые пространства имен:

```
using Microsoft.AspNet.Identity.Owin;
using Microsoft.AspNet.Identity;
```

5. Откройте представление и добавьте разметку для вывода информации о всех ролях текущего пользователя:

```
<p>Все роли этого пользователя</p>
@foreach (var c in @ViewBag.rol)
```

```
{  
    <li>@c</li>  
}
```

6. Запустите приложение и перейдите на страницу «О программе». Проверьте, что список ролей текущего пользователя отобразился.

Литература

1. Мэтью Макдональд, Адам Фримен, Марио Шпушта Microsoft ASP.NET 4 с примерами на C# 2010 для профессионалов, М.: Вильямс, - 2011. – 1424 с.
2. Фримен А., Сандерсон С. ASP.NET MVC 4 Framework с примерами на C# 5.0 для профессионалов. 4-е изд.
3. Палермо Д., Богард Д., Хекстер Э., Хинзе М., Скиннер Д. ASP.NET MVC 4 в Действии