

Разработка Web приложений с использованием ASP.NET MVC

СОДЕРЖАНИЕ

Лабораторная работа 5. Разработка модели	2
Упражнение 1. Создание приложения с реализацией хранения данных	2
Упражнение 2. Настройка работы с данными.....	9

Лабораторная работа 5. Разработка модели

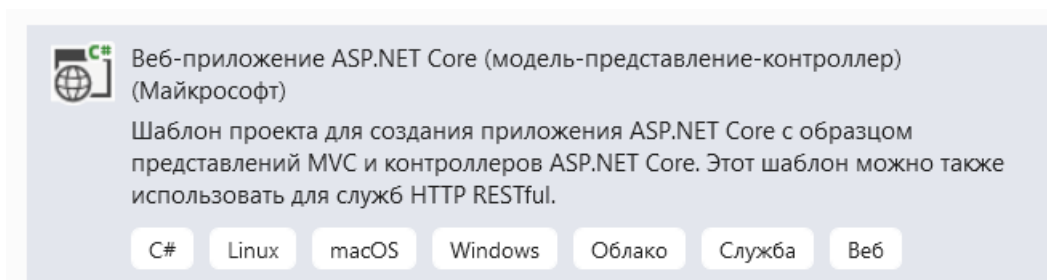
В этой работе вы создадите приложение, работающее с базой данных с применением Entity Framework, но без создания шаблонных элементов (их вы будете реализовывать в следующей работе).

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 **MvcCreditApp**.
2. В окне выбора шаблона в разделе **Шаблоны** укажите **Веб-приложение ASP.NET Core (MVC)**



3. Укажите имя проекта **MvcCreditApp** и расположение.
4. В окне **Дополнительные сведения** в списке *Тип проверки подлинности* выберите *Индивидуальные учетные записи* (они потребуются в работе 8):

Дополнительные сведения

Веб-приложение ASP.NET Core (модель-представление-контроллер) (Майкрософт)

Платформа ⓘ
[.NET 7.0 (Поддержка со стандартным сроком)]

Тип проверки подлинности ⓘ
[Индивидуальные учетные записи] ✓

☒ Настроить для HTTPS ⓘ
☐ Включить Docker ⓘ

Операционная система Docker ⓘ
[Linux]

☐ Не использовать операторы верхнего уровня ⓘ

5. Поскольку при создании проекта была выбрана поддержка проверки подлинности, проверьте, что в папке **Models** есть класс **ApplicationDbContext**, а в файле **Program.cs** с помощью инверсии зависимостей добавлен **AddDbContext<ApplicationDbContext>**. Наличие данного контекста нужно будет учесть при создании своего.
6. 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; }
}
```

- ✓ Обратите внимание, что компилятор создает предупреждения о том, как обрабатываются значения **null**. В настоящее время подсистема формирования шаблонов не поддерживает ссылочные типы, допускающие

значения NULL, поэтому модели, используемые в формировании шаблонов, также не поддерживают их, чтобы исключить предупреждения из ссылочных типов, допускающих значения NULL, удалите следующую строку из файла проекта MvcCreditApp.csproj:

```
<Nullable>enable</Nullable>
```

Применение Entity Framework

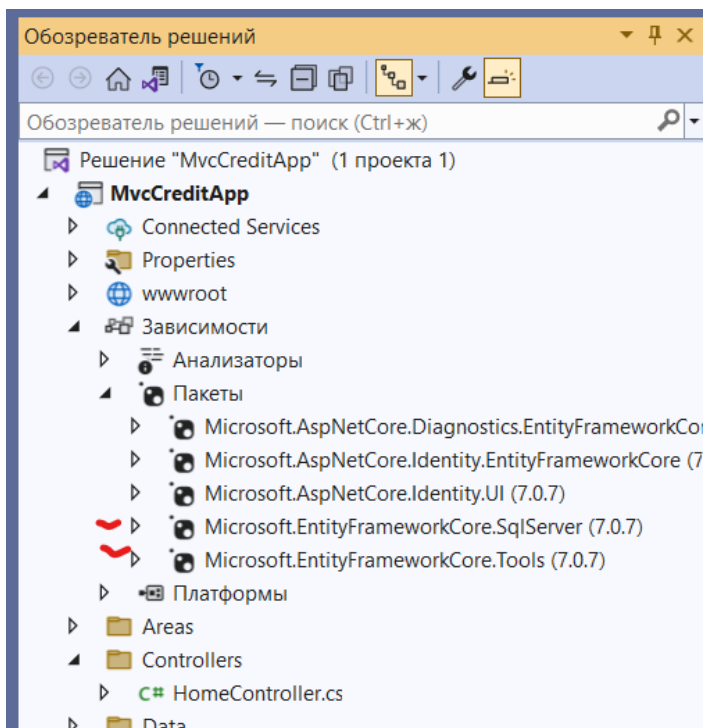
Для доступа к данным будем использовать **Entity Framework**. Этот фреймворк позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

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 требуемые ссылки на пространство имен и в классе `CreditContext` объявите соответствующие таблицы (по традиции во множественном числе) базы данных с помощью свойств с типом `DbSet`:

```
using Microsoft.EntityFrameworkCore;
using MvcCreditApp.Models;

namespace MvcCreditApp.Data
{
    public class CreditContext : DbContext
    {
        public CreditContext(DbContextOptions<CreditContext> options)
        : base(options)
        { }
        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.

3. Постройте приложение.
4. Зарегистрируйте контекст базы данных с помощью контейнера внедрения зависимостей в файле `Program.cs` (по аналогии с `ApplicationDbContext`):

```
builder.Services.AddDbContext<CreditContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("CreditContext")) ?? throw new InvalidOperationException("Connection string 'CreditContext' not found."));
```

```
var app = builder.Build();
```

5. Система конфигурации ASP.NET Core считывает ключ **ConnectionString**, в данном случае (для разработки на локальном уровне) конфигурация получает строку подключения из файла `appsettings.json`, добавьте ее по аналогии со строкой для **DefaultConnection**:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=aspnet-MvcCreditApp-019d1794-c4ae-44a4-a7ee-6416b3b7f65d;Trusted_Connection=True;MultipleActiveResultSets=true",
    "CreditContext":
"Server=(localdb)\\mssqllocaldb;Database=CreditContext;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

Первоначальная миграция

EF Core использует функцию миграции для создания базы данных, миграции — это набор средств, с помощью которых можно создавать и обновлять базы данных в соответствии с моделью данных.

- В меню **Средства** последовательно выберите пункты **Диспетчер пакетов NuGet → Консоль диспетчера пакетов**.
- Введите в консоли диспетчера пакетов (PMC) команду для добавления миграции (обратите внимание на необходимость явно указать имя класса контекста, так в данном проекте их два):

```
PM> Add-Migration InitialCreate -Context 'CreditContext'
```

В случае успешного выполнения команды должно быть соответствующее сообщение:

```
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

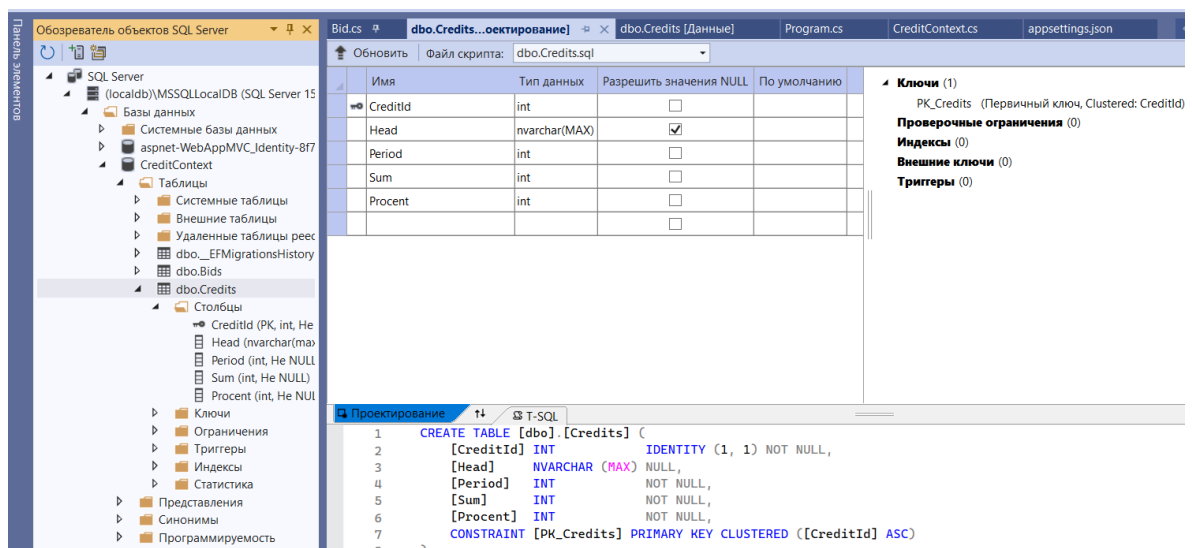
- Выполните обновление базы данных:

```
PM> Update-Database -Context 'CreditContext'
```

В случае успешного выполнения команды должно быть соответствующее сообщение и скрипты для создания базы данных и таблиц:

```
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (274ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
  CREATE DATABASE [CreditContext];
...
```

- Для просмотра созданной базы данных откройте с помощью меню **Вид** обозреватель объектов SQL Server (SSOX).
- Для локального сервера в контекстном меню выберите команду **Обновить**.
- Раскройте базу данных **CreditContext** и для таблицы **dbo.Credits** выберите в контекстном меню **Показать конструктор**:



Инициализация базы данных с тестовыми данными

EF создает пустую базу данных. В этой части вы добавите метод, который вызывается после создания базы данных и заполняет ее тестовыми данными.

12. В папке **Models** создайте класс **SeedData** со следующим кодом:

```
using Microsoft.EntityFrameworkCore;
using MvcCreditApp.Data;

namespace MvcCreditApp.Models
{
    public class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new CreditContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<CreditContext>>()))
            {
                if (context == null || context.Credits == null)
                {
                    throw new ArgumentNullException("Null CreditContext");
                }

                // Если в базе данных уже есть кредиты,
                // то возвращается инициализатор заполнения и кредиты не добавляются
                if (context.Credits.Any())
                {
                    return;
                }

                context.Credits.Add(new Credit { Head = "Ипотечный", Period =
10, Sum = 1000000, Procent = 15 });
                context.Credits.Add(new Credit { Head = "Образовательный",
Period = 7, Sum = 300000, Procent = 10 });
                context.Credits.Add(new Credit { Head = "Потребительский",
Period = 5, Sum = 500000, Procent = 19 });

                // можно использовать метод AddRange
                context.Credits.AddRange(
                    new Credit
                    {
                        Head = "Льготный",
                        Period = 12,
                        Sum = 55555,
                        Procent = 7
                    },

                    new Credit
                    {
                        Head = "Срочный",
                        Period = 3,
                        Sum = 3333,
                        Procent = 19
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

- ✓ Обратите внимание на то, что если в базе данных уже есть кредиты, то возвращается инициализатор заполнения и кредиты не добавляются.

13. Данными БД заполняется при запуске приложения, поэтому в файл Program.cs добавьте следующий код:

```
using MvcCreditApp.Models;
...
var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
```

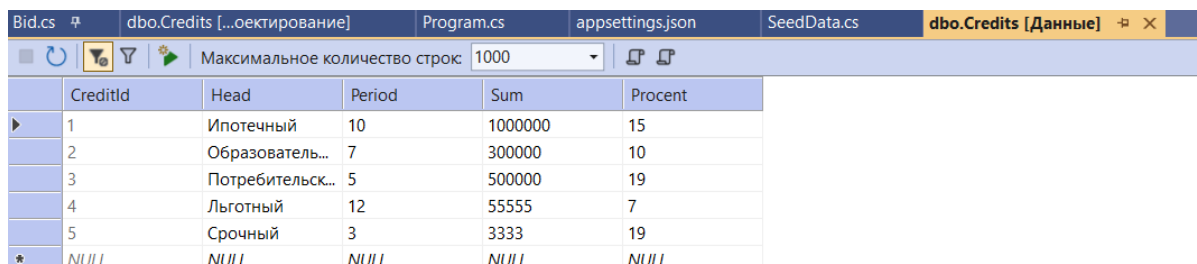
При запуске приложения в этой части Program.cs выполняются следующие действия:

- получение экземпляра контекста базы данных из контейнера внедрения зависимостей;
- вызывается метод инициализации;
- по завершению работы метода Initialize контекст удаляется.

Тестирование приложения

14. Постройте и запустите приложение.

15. Проверьте с помощью обозревателя серверов, что данные действительно заполнились в требуемую таблицу:



CreditId	Head	Period	Sum	Procent
1	Ипотечный	10	1000000	15
2	Образователь...	7	300000	10
3	Потребительск...	5	500000	19
4	Льготный	12	55555	7
5	Срочный	3	3333	19
NULL	NULL	NULL	NULL	NULL

16. Установите точку останова в `if (context.Credits.Any())` и снова запустите приложение, так как данные есть, то проверьте, что на следующем шаге заполнение данными игнорируется.

17. С помощью обозревателя серверов удалите все записи в базе данных и снова запустите приложение с той же точкой останова. Выполняя пошагово код проверьте заполнение таблицы новыми данными.

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 `MvcCreditApp.Models`.
3. В поле класса контроллера создайте экземпляр контекста данных:

```
private readonly CreditContext db;
```

4. Реализуйте внедрение зависимости с помощью конструктора контроллера (logger можно оставить):

```
public HomeController(ILogger<HomeController> logger, CreditContext context)
{
    _logger = logger;
    db = context;
}
```

5. В методе `Index()` обратитесь к контексту, получите все записи о кредитах, создайте свойство `Credits` в объекте `ViewBag` и присвойте ему извлеченный список (объект `ViewBag` является таким объектом, который с помощью свойства передает данные в представление):

```
public IActionResult Index()
{
    var allCredits = db.Credits.ToList<Credit>();
    ViewBag.Credits = allCredits;
    return View();
}
```

6. 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. Добавьте ниже следующий код разметки:

```
<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 asp-controller="Home" asp-action="CreateBid" asp->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, реализованная с помощью тег-хелпера `<a asp-controller="Home" asp-action="CreateBid" asp->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).

MvcCreditApp Home Privacy

Welcome

Learn about [building Web apps with ASP.NET Core](#).

Программы кредитования

Тип кредита	Период кредитования	Максимальная сумма	Ставка %
Ипотечный	10	1000000	15 %
Образовательный	7	300000	10 %
Потребительский	5	500000	19 %
Льготный	12	55555	7 %
Срочный	3	3333	19 %

[Подать заявку на получение кредита »](#)

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, " + newBid.Name + ", 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 (представление будет не типизированным), 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 (код разметки можете изменить на свое усмотрение – главное реализовать передачу 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>
      <tr><td><p>Выберите из списка тип кредита :</p></td>
      <td>
        <select name="CreditHead">
          @foreach (var cr in ViewBag.Credits)
          {
            <option>@cr.Head</option>
          }
        </select>
      </td>
    </tr>
    <tr><td><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**

```
<table class="table">
```

2. Откройте находящийся в папке **wwwroot\css** файл **site.css** и добавьте раздел **table** стиль для всех таблиц:

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

MvcCreditApp Home Privacy

Welcome

Learn about [building Web apps with ASP.NET Core](#).

Программы кредитования

Тип кредита	Период кредитования	Максимальная сумма	Ставка %
Ипотечный	10	1000000	15 %
Образовательный	7	300000	10 %
Потребительский	5	500000	19 %
Льготный	12	55555	7 %
Срочный	3	3333	19 %

[Подать заявку на получение кредита »](#)

Рис. 5.2 Применение стилизации к стартовой странице

CreateBid

Существующие заявки

Имя	Тип кредита
Polo	Образовательный
Polo	Потребительский

Форма подачи заявки по программам кредитования

Введите свое имя	<input type="text"/>
Выберите из списка тип кредита :	<input type="text" value="Ипотечный"/>
<input type="button" value="Отправить"/>	

Рис. 5.3 Применение стилизации к странице подачи заявки