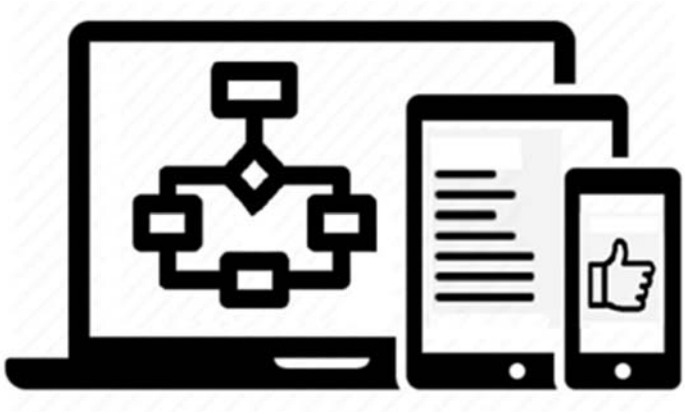
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ СЕРВЕРНОЙ РАЗРАБОТКИ

Методические рекомендации к лабораторным работам для студентов специальности 6-05-0611-01 «Информационные системы и технологии» дневной и заочной форм обучения



Могилев 2026

УДК 004.42
ББК 32.973-018
С56

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Программное обеспечение информационных технологий» «4» февраля 2026 г., протокол № 7

Составитель канд. техн. наук, доц. Н. Н. Горбатенко

Рецензент канд. техн. наук, доц. В. М. Ковальчук

Методические рекомендации разработаны на основе рабочей программы по дисциплине «Современные технологии серверной разработки» для студентов специальности 6-05-0611-01 «Информационные системы и технологии» дневной и заочной форм обучения. Предназначены для использования при выполнении лабораторных работ.

Учебное издание

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ СЕРВЕРНОЙ РАЗРАБОТКИ

Ответственный за выпуск	В. В. Кутузов
Корректор	А. А. Подошевко
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс. Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 21 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение 4
1 Лабораторная работа № 1. Изучение структуры приложения ASP.NET Core 5
2 Лабораторная работа № 2. Создание первого приложения ASP.NET Core 6
3 Лабораторная работа № 3. Промежуточное программное обеспечение (middleware) ASP.NET Core. Обработка запросов и подготовка ответов. Конвейер обработки запросов 7
4 Лабораторная работа № 4. Пользовательский ввод и визуализация в программе ASP.NET Core при использовании синтаксиса Razor. Маршрутизация 12
5 Лабораторная работа № 5. Разработка программы ASP.NET Core с использованием паттерна MVC 19
6 Лабораторная работа № 6. Удаленное выполнение запросов к базе данных на основе использования Entity Framework Core 22
Список литературы 25

Введение

Целью преподавания дисциплины «Современные технологии серверной разработки» является получение обучающимися базовых знаний в области разработки клиент-серверных приложений с использованием фреймворка ASP.NET Core.

Лабораторные работы проводятся с целью закрепления теоретических знаний, полученных на лекциях, путем их практического применения.

Последовательность выполнения лабораторных работ.

1 Ознакомиться с целью предстоящей работы.

2 Изучить вопросы пункта «Теоретические сведения», используя указанные литературные источники и конспект лекций.

3 Ответить на контрольные вопросы к лабораторной работе.

4 Составить план решения задачи индивидуального задания.

5 Набрать текст программы решения задачи в среде Visual Studio.NET.

6 Отладить программу и убедиться в ее правильной работе.

7 Подготовить отчёт по лабораторной работе.

8 Защитить лабораторную работу. Защита включает в себя демонстрацию работы программы преподавателю и ответы на его вопросы по теме лабораторной работы в объёме методических рекомендаций. После защиты лабораторной работы преподаватель ставит на титульном листе свою подпись и дату. Только после этого лабораторная работа считается полностью выполненной.

Отчёт по лабораторной работе оформляется на листах формата А4 и содержит следующее:

– титульный лист;

– тема и цель работы;

– постановка задачи: текст задания согласно варианту;

– краткие теоретические сведения: основные понятия и методы, использованные при выполнении работы;

– листинг программы: полный код программы с подробными комментариями;

– результаты тестирования: скриншоты работы программы с различными входными данными, демонстрирующие ее корректность;

– выводы: краткое описание полученных навыков и возникших при выполнении работы трудностей.

1 Лабораторная работа № 1. Изучение структуры приложения ASP.NET Core

***Цель работы*:** изучить назначение и область применения фреймворка ASP.NET Core.

Последовательность выполнения лабораторной работы

1 Прочитать подразд. 1.1 «Введение в ASP.NET Core» [1, с. 34–40] и п. 1.2.1 «Какие типы приложений можно создавать?» [1, с. 40–43].

На основе изученного материала в письменной форме ответить на следующие вопросы.

Что такое веб-фреймворк и для чего он используется?

Какие преимущества даёт использование веб-фреймворков при разработке приложений?

Дайте определение ASP.NET Core. Чем он отличается от классического ASP.NET?

Какие основные задачи решает ASP.NET Core?

На каких операционных системах может работать ASP.NET Core?

Какие типы приложений можно создавать с помощью ASP.NET Core?

Что понимается под кроссплатформенностью ASP.NET Core?

Для чего используется модель MVC в ASP.NET Core?

Какие основные компоненты входят в архитектуру ASP.NET Core?

Почему ASP.NET Core считается современным инструментом для веб-разработки?

2 Прочитать подразд. 1.3 «Как работает ASP.NET Core?» [1, с. 51–57].

В письменной форме ответить на следующие на вопросы.

Привести схему и описать процесс запроса веб-страницы с сервера.

Что такое HTTP-запрос и какие основные части он содержит?

Что такое HTTP-ответ и какие основные компоненты он включает?

Как приложение ASP.NET Core обрабатывает запрос?

Какую роль выполняет Kestrel в ASP.NET Core?

2 Лабораторная работа № 2. Создание первого приложения ASP.NET Core

Цель работы: ознакомиться со структурой, элементным составом, работой веб-приложения ASP.NET Core.

Последовательность выполнения лабораторной работы

1 Прочесть из [1, гл. 2, с. 59–94].

2 Создать веб-приложение, рассматриваемое в [1, гл. 2].

3 Описать состав и назначение компонентов веб-приложения.

4 Составить подробные комментарии к каждой строке кода созданного веб-приложения.

Контрольные вопросы

1 Как создать новое приложение ASP.NET Core при помощи шаблона? Какие шаги включаются в стартовый шаблон?

2 Что хранится в файле .csproj у вновь созданного ASP.NET Core приложения?

3 Как устроен класс Program.cs? Что в нём настраивается?

4 Какова роль класса Startup? Что делают методы ConfigureServices и Configure?

5 Как добавлять и настраивать сервисы в методе ConfigureServices? Приведите примеры.

6 Как определяется, каким образом приложение обрабатывает HTTP-запросы (какой middleware задействуется)?

7 Что такое Razor Pages и как в базовом шаблоне они участвуют в ответах приложения?

8 В каких случаях стоит использовать Razor Pages вместо контроллеров и MVC?

3 Лабораторная работа № 3. Промежуточное программное обеспечение (middleware) ASP.NET Core. Обработка запросов и подготовка ответов. Конвейер обработки запросов

Цель работы: освоить принципы построения конвейера обработки HTTP-запросов с использованием промежуточного программного обеспечения (middleware) в ASP.NET Core; научиться настраивать встроенные middleware-компоненты: реализовывать обработку ошибок и ознакомиться с основами работы Razor Pages.

Теоретические сведения

Конвейер middleware – это последовательность обработчиков запросов, через которые проходит каждый HTTP-запрос. Порядок регистрации middleware критически важен. Middleware позволяет реализовать логирование, маршрутизацию, статические файлы, безопасность и обработку ошибок. Также в ASP.NET Core поддерживается модель Razor Pages – упрощённый подход к созданию страниц без контроллеров, но с использованием того же конвейера. Более подробные сведения см. в [1, с. 95–129].

Последовательность выполнения лабораторной работы

1 Анализ встроенного конвейера.

1.1 Создайте новое веб-приложение ASP.NET Core (шаблон Web Application с Razor Page).

1.2 Изучите файл Program.cs (или Startup.cs для .NET 5 и ниже) – найдите методы app.Use..., отвечающие за регистрацию middleware.

1.3 Зарегистрируйте и протестируйте работу следующих встроенных middleware компонентов:

– UseDeveloperExceptionPage() – для отладки ошибок;
– UseWelcomePage() – для отображения стартовой страницы приложения ASP.NET Core;
– UseStaticFiles() – для обслуживания статических файлов (css, js, файлов изображений);
– UseRouting() и UseEndpoints() – для маршрутизации и обработки запросов контроллерами;
– UseHttpsRedirection() – для перенаправления HTTP → HTTPS;
– UseAuthorization() – проверка прав доступа.

1.4 Поэкспериментируйте с порядком middleware.

Что произойдёт, если UseStaticFiles() поставить после UseRouting()?

Что будет, если UseRouting() поставить после UseEndpoints()?

Зафиксируйте наблюдения в отчёте.

2 Знакомство с Razor Pages.

2.1 Добавьте поддержку Razor Pages в приложение. Убедитесь, что в Program.cs вызваны:

```
app.UseRouting();
app.MapRazorPages(); // в блоке UseEndpoints или отдельно
```

2.2 Создайте файл Pages/Index.cshtml с содержимым:

```
@page
<h1>Добро пожаловать на главную страницу!</h1>
<p>Текущее время: @DateTime.Now</p>
```

2.3 Запустите приложение и перейдите по адресу /Index или /.

2.4 Создайте файл Pages/About.cshtml с содержимым:

```
@page
@model laba3.Pages.AboutModel
<h2>@Model.Message</h2>
```

2.5 Создайте файл кода Pages/About.cshtml.cs:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace laba3.Pages
{
   public class AboutModel : PageModel
   {
      public string Message { get; private set; } = "Страница 'О нас'";

      public void OnGet()
      {
      }
   }
}
```

2.6 Проверьте маршрутизацию.

Как ASP.NET Core определяет маршрут по имени файла?

Что будет, если переименовать About.cshtml в О-нас.cshtml?

3 Обработка ошибок с помощью промежуточного ПО.

3.1 Найдите в файле Startup.cs метод Configure() и настройте middleware для обработки исключений:

– в режиме разработки (if (env.IsDevelopment())) используйте UseDeveloperExceptionPage();

– в режиме продакшена добавьте UseExceptionHandler("/Error") – для перехвата исключений и перенаправления на страницу ошибки.

3.2 Зарегистрируйте в методе Configure() следующие встроенные middleware компоненты:

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

// Обработка HTTP-статусов (404, 500 и др.)
app.UseStatusCodePagesWithReExecute("/Error/{0}"); //перехватывает HTTP-статусы, например 404, и перенаправляет на /Error/404

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
};
```

3.3 Используя Razor Page; создайте простую страницу /Error, которая отображает:

– код ошибки;

– сообщение (в режиме разработки – с деталями; в продакшене – общее сообщение).

Для этого необходимо:

1) открыть файл: Pages/Error.cshtml и вставить следующий код:

```
@page "{code?}"
@model ErrorHandlingApp.Pages.ErrorModel
@{
   ViewData["Title"] = "Ошибка";
}

<div class="container mt-5">
   <h1 class="text-danger">Ошибка @Model.StatusCode</h1>
   <p>@Model.Message</p>

   @if (Model.StatusCode == 404)
   {
      <p>Проверьте адрес и попробуйте снова.</p>
   }
</div>
```

2) открыть файл Pages/Error.cshtml.cs и вставить следующий код:

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Hosting;
```

```
namespace ErrorHandlingApp.Pages
{
    public class ErrorModel : PageModel
    {
        private readonly IWebHostEnvironment _environment;

        public ErrorModel(IWebHostEnvironment environment)
        {
            _environment = environment;
        }

        public int StatusCode { get; set; } = 500;
        public string Message { get; set; } = "Произошла внутренняя ошибка сервера.";

        public void OnGet(int? code = null)
        {
            // Если вызван через UseStatusCodePagesWithReExecute (например, 404)
            if (code.HasValue)
            {
                StatusCode = code.Value;
            }
            else
            {
                // Если вызван через UseExceptionHandler – читаем исключение из контекста
                var exceptionHandlerPathFeature = HttpContext.Features.Get<IExceptionHandler-
PathFeature>();
                if (exceptionHandlerPathFeature?.Error != null)
                {
                    // Исключение есть -> статус 500 по умолчанию
                    StatusCode = 500;
                }
            }

            // Формируем сообщение в зависимости от окружения
            if (_environment.IsDevelopment())
            {
                // В разработке показываем код ошибки и типичное сообщение
                Message = $"Ошибка {StatusCode}. Детали отображаются только в режиме раз-
работки.";
            }
            else
            {
                // В продакшене – общее сообщение
                Message = StatusCode switch
                {
                    404 => "Страница не найдена.",
                    403 => "Доступ запрещён.",
                    _ => "Произошла внутренняя ошибка сервера."
                };
            }
        }
    }
}
```

3.4 Протестируйте работу обработчика:

– добавьте в проект пустую страницу Razor Page, выполнив команду «Добавить – страница Razor пустая» и в поле «Имя» введите TestException.cshtml;

– в файл TestException.cshtml.cs вставьте код:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ErrorHandlingApp.Pages
{
   public class TestExceptionModel : PageModel
   {
      public void OnGet()
      {
         throw new System.Exception("Test exception for error handling demo.");
      }
   }
}
```

– в файл TestException.cshtml вставьте код:

```
@page
@model ErrorHandlingApp.Pages.TestExceptionModel
```

– теперь попытка перейти на страницу /TestException в адресной строке браузера вызовет исключение и перенаправит на страницу /Error с кодом 500;

– переход на /non-existent-page вызовет 404 и перенаправит на /Error с кодом 404.

3.5 Проверьте, что в продакшен-режиме подробная информация об ошибках не отображается пользователю.

Для этого:

– откройте файл launchSettings.json;

– убедитесь, что свойству ASPNETCORE_ENVIRONMENT присвоено значение «Development»;

– запустите приложение на выполнение. В адресной строке браузера перейдите на страницу /TestException и после появления исключения нажмите на клавишу «F5». В результате в браузере отобразится страница с подробным описанием ошибки;

– перейдите на несуществующую страницу /non-existent. Появится ошибка 404;

– присвойте ASPNETCORE_ENVIRONMENT значение “Production” и запустите приложение;

– теперь /TestException покажет «Произошла внутренняя ошибка сервера», а /non-existent покажет: «Страница не найдена».

Контрольные вопросы

1 Что такое middleware в ASP.NET Core? Опишите назначение и принцип работы промежуточного ПО в конвейере обработки HTTP-запросов.

2 Почему порядок регистрации middleware в методе Configure критически важен? Приведите пример, когда неправильный порядок приводит к ошибке или неожиданному поведению приложения.

3 Какие встроенные middleware-компоненты используются для обслуживания статических файлов, маршрутизации, перехвата ошибок? Приведите примеры их регистрации в Program.cs.

4 Как настроить разные страницы обработки ошибок для среды разработки и продакшена? Опишите, какие middleware используются и как определяется текущая среда выполнения.

5 Что делает метод app.UseRouting() и зачем он нужен перед app.UseEndpoints()? Объясните, как они взаимодействуют друг с другом.

6 Что такое Razor Pages и чем эта модель отличается от MVC? Назовите преимущества Razor Pages для создания простых страниц и форм.

7 Как происходит маршрутизация в Razor Pages? По какому принципу URL-адрес страницы соотносится с её физическим расположением в проекте?

8 Для чего используется атрибут [BindProperty] в PageModel? Как он влияет на привязку данных из формы?

9 Как реализовать обработку POST-запроса на странице Razor? Приведите пример метода в PageModel, обрабатывающего отправку формы.

10 Как создать и использовать именованный обработчик (например, OnGetDetails) в Razor Pages? Как передать параметр (например, ID) в такой обработчик через URL?

4 Лабораторная работа № 4. Пользовательский ввод и визуализация в программе ASP.NET Core при использовании синтаксиса Razor. Маршрутизация

Цель работы: научиться создавать, настраивать и обрабатывать веб-страницы с использованием связки .cshtml + PageModel, реализовывать формы, привязку данных, валидацию и маршрутизацию.

Теоретические сведения

Razor Pages – это подход к разработке веб-приложений, где каждая страница представляет собой пару файлов:

– PageName.cshtml – представление (HTML + Razor-синтаксис);

– PageName.cshtml.cs – модель страницы (PageModel), содержащая логику обработки запросов на C#.

Основные преимущества.

1 Простота и локальность кода (всё, что нужно для страницы – рядом).

2 Автоматическая маршрутизация по пути файла.

3 Поддержка обработчиков: OnGet(), OnPost(), OnPostCustomName().

4 Встроенная привязка модели и валидация через Data Annotations.

5 Идеально подходит для форм, целевых страниц landing pages.

Более подробные сведения см. в [1, с. 133–163].

Последовательность выполнения лабораторной работы

1 Создание проекта и изучение его структуры.

1.1 Создайте новое веб-приложение ASP.NET Core (шаблон Web Application с Razor Page).

1.2 Запустите приложение.

1.3 Изучите структуру:

- папка Pages/ – все страницы (Index, Privacy, Shared/_Layout.cshtml);
- файл Program.cs – конфигурация middleware и маршрутов.

1.4 Откройте в браузере: https://localhost:5001 → изучите работу страниц /Index, /Privacy. Обратите внимание: маршруты совпадают с именами файлов – /Index → Pages/Index.cshtml.

2 Создание страницы «Оставьте отзыв».

Задача. Создать страницу /Feedback с формой, показанной на рисунке 1. Эта страница должна открываться в браузере при обращении к ней по ссылке URL: https://localhost:44342/Feedback.

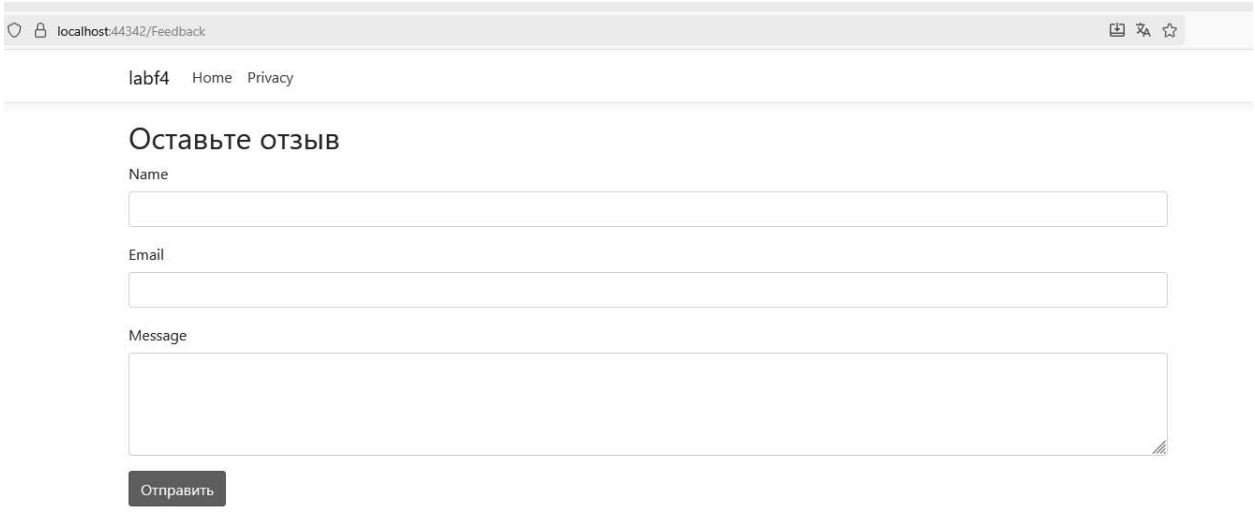


Рисунок 1 – Вид страницы «Оставьте отзыв»

Для решения задачи выполните следующие действия.

2.1 Добавьте в проект пустую страницу Razor Page, выполнив команду «Добавить – страница Razor пустая». В поле «Имя» введите Feedback.cshtml.

2.2 В файл Feedback.cshtml вставьте код:

```
@page
@model laba4.Pages.FeedbackModel
@{
    ViewData["Title"] = "Оставьте отзыв";
}

<h2>@ViewData["Title"]</h2>

<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>

    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Message"></label>
        <textarea asp-for="Message" class="form-control" rows="4"></textarea>
        <span asp-validation-for="Message" class="text-danger"></span>
    </div>

    <button type="submit" class="btn btn-primary">Отправить</button>
</form>

@if (Model.IsSubmitted)
{
    <div class="alert alert-success mt-3">
        Спасибо, @Model.Name! Ваш отзыв принят.
    </div>
}
```

2.3 В файл модели Feedback.cshtml.cs вставьте код:

```
namespace laba4.Pages
{
    public class FeedbackModel : PageModel
    {
        [BindProperty]
        [Required(ErrorMessage = "Имя обязательно")]
```

```
        [StringLength(50, MinimumLength = 2, ErrorMessage = "Имя должно быть от 2 до 50 символов")]
        public string Name { get; set; } = "";

        [BindProperty]
        [Required(ErrorMessage = "Email обязателен")]
        [EmailAddress(ErrorMessage = "Некорректный адрес email")]
        public string Email { get; set; } = "";

        [BindProperty]
        [Required(ErrorMessage = "Сообщение обязательно")]
        [StringLength(500, MinimumLength = 10, ErrorMessage = "Сообщение должно быть от 10 до 500 символов")]
        public string Message { get; set; } = "";

        public bool IsSubmitted { get; private set; } = false;

        public void OnGet()
        {
            // Инициализация при первом открытии страницы
        }

        public IActionResult OnPost()
        {
            if (!ModelState.IsValid)
            {
                return Page(); // Вернуться на страницу с ошибками
            }

            // Имитация обработки отзыва
            IsSubmitted = true;

            // Очистка полей после отправки
            Name = "";
            Email = "";
            Message = "";

            return Page();
        }
    }
}
```

2.4 Проверьте:

– доступность страницы по адресу /Feedback;

– работу валидации (пустые поля, неверный email, короткое сообщение);

– отображение сообщения об успешной отправке.

3 Использование именованных обработчиков – «Страница товаров».

Задача. Создайте страницу /Products с формой, показанной на рисунке 2. Эта страница должна открываться в браузере при обращении к ней по ссылке вида: https://localhost:44342/Products. При нажатии на текстовое поле «Подробнее» должна выводится информация о наименовании товара (Товар 1) и его идентификационный номер (ID: 1).

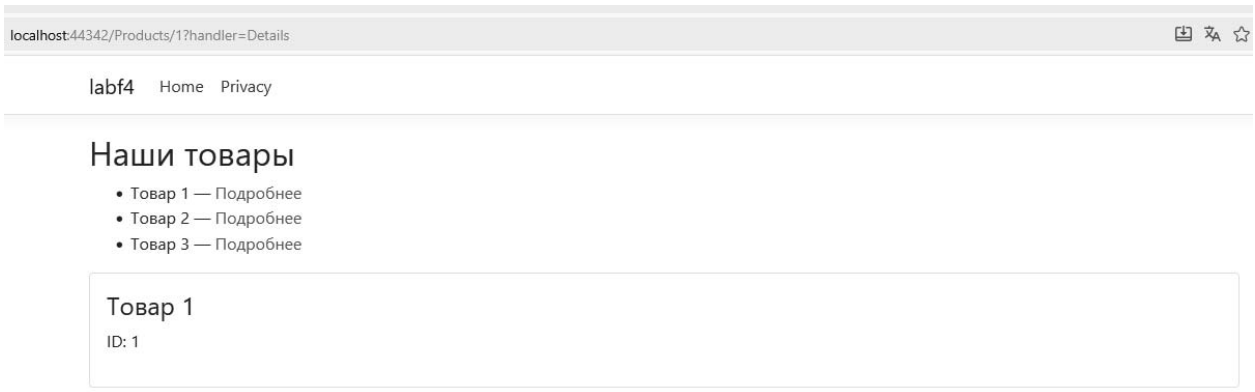


Рисунок 2 – Вид страницы /Products (Продукты)

Для этого выполните следующие действия.

3.1 Создайте страницу Pages/Products.cshtml:

```
@page "{handler?}/{id?}"
@* Поддержка параметров в маршруте *@
@model Lab4_RazorSite.ProductsModel

<h2>Наши товары</h2>

<ul>
    @foreach (var product in Model.Products)
    {
        <li>
            @product.Name -
            <a asp-page-handler="Details" asp-route-id="@product.Id">Подробнее</a>
        </li>
    }
</ul>

@if (Model.SelectedProduct != null)
{
    <div class="card p-3 mt-3">
        <h4>@Model.SelectedProduct.Name</h4>
        <p>ID: @Model.SelectedProduct.Id</p>
    </div>
}
```

3.2 Создайте модель Pages/Products.cshtml.cs:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace Lab4_RazorSite
{
    public class ProductsModel : PageModel
    {
        public List<Product> Products { get; set; }
        public Product? SelectedProduct { get; set; }

        public void OnGet()
        {
            // Инициализация списка товаров
            Products = new List<Product>
            {
                new Product { Id = 1, Name = "Товар 1" },
                new Product { Id = 2, Name = "Товар 2" },
                new Product { Id = 3, Name = "Товар 3" }
            };
        }

        // Этот метод ОБЯЗАТЕЛЬНО нужен!
        public void OnGetDetails(int id)
        {
            // Повторно загружаем список (или получаем из БД)
            Products = new List<Product>
            {
                new Product { Id = 1, Name = "Товар 1" },
                new Product { Id = 2, Name = "Товар 2" },
                new Product { Id = 3, Name = "Товар 3" }
            };

            SelectedProduct = Products.FirstOrDefault(p => p.Id == id);
        }
    }

    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
    }
}
```

3.3 Проверьте:

- переход по ссылке URL вида /Products/Details;
- отображение информации о выбранном товаре.

4 Расширенное задание.

4.1 Добавьте страницу /Contact с формой обратной связи и отправкой данных в файл или лог (через ILogger).

4.2 Реализуйте кастомную валидацию (пользовательскую логику проверки данных, которая выходит за рамки стандартных встроенных правил валидации), например, запрет домена @mail.ru в поле Email.

4.3 Используйте TempData для отображения flash-сообщений (временных уведомлений, которые показываются пользователю один раз после перенаправления или следующего HTTP-запроса, а затем автоматически удалятся), например, после POST-обработки, т. е. после основной обработки HTTP-запроса.

Контрольные вопросы

1 Что такое Razor Pages и в чём её основное отличие от модели MVC? Объясните, почему Razor Pages удобна для создания небольших форм и страниц.

2 Из каких двух основных файлов состоит страница Razor Pages? Опишите назначение каждого файла и как они связаны между собой.

3 Как происходит маршрутизация в Razor Pages? По какому принципу URL-адрес страницы определяется на основе её расположения в проекте?

4 Для чего используется класс, наследуемый от PageModel? Какие методы обработки запросов можно в нём определить и как они вызываются?

5 Как передать данные из формы на странице Razor в модель страницы? Опишите назначение атрибута [BindProperty] и условия его работы.

6 Как реализовать валидацию данных на странице Razor? Приведите пример использования Data Annotations (например, [Required], [EmailAddress]) и отображения ошибок в представлении.

7 Как обработать POST-запрос на странице Razor? Напишите сигнатуру метода, который будет вызван при отправке формы методом POST.

8 Что такое именованные обработчики (handler methods)? Как создать метод OnPostSubscribe() и вызвать его из формы или ссылки?

9 Как передать параметр (например, ID) в обработчик страницы через URL? Приведите пример директивы @page с параметром и метода OnGetDetails(int id).

10 Как отобразить сообщение об успешной обработке формы без перезагрузки страницы или с очисткой полей? Опишите подход с использованием свойства модели и условного отображения в .cshtml.

5 Лабораторная работа № 5. Разработка программы ASP.NET Core с использованием паттерна MVC

Цель работы: изучение архитектурного паттерна Model-View-Controller (MVC), механизмов маршрутизации, передачи данных через модели и реализации валидации данных на платформе ASP.NET Core.

Теоретические сведения

ASP.NET Core MVC – это современный фреймворк для создания веб-приложений, разделяющий логику на три компонента:

1) Model (Модель): содержит бизнес-логику и описывает структуру данных. С помощью атрибутов (Data Annotations) здесь задаются правила валидации;

2) View (Представление): отвечает за интерфейс. Использует файлы .cshtml (движок Razor), где HTML перемешивается с кодом C#;

3) Controller (Контроллер): обрабатывает запросы пользователя, взаимодействует с моделью и выбирает нужное представление.

Более подробные сведения см. в [2, с. 31–71].

Последовательность выполнения лабораторной работы

Описание прикладной задачи. В рамках работы необходимо разработать модуль управления каталогом домашней библиотеки.

Функциональные требования:

– просмотр списка: система должна отображать таблицу с книгами (ID, автор, название);

– добавление данных: пользователь заполняет форму и данные сохраняются в список в оперативной памяти;

– валидация: система должна блокировать отправку пустых полей или некорректных имен авторов.

Шаг 1: создание модели (Models/Book.cs).

Добавьте атрибуты валидации, чтобы контролировать ввод данных.

```
using System.ComponentModel.DataAnnotations;

namespace MvcSimpleApp.Models
{
   public class Book
   {
      public int Id { get; set; }

      [Required(ErrorMessage = "Название книги обязательно")]
      [StringLength(100, MinimumLength = 2, ErrorMessage = "От 2 до 100 символов")]
      [Display(Name = "Название книги")]
      public string Title { get; set; }
```

```
        [Required(ErrorMessage = "Автор обязателен")]
        [RegularExpression(@"^[A-ZA-Я][a-za-я ]+$", ErrorMessage = "Имя должно начинаться с большой буквы")]
        [Display(Name = "Автор")]
        public string Author { get; set; }
    }
}
```

Шаг 2: создание контроллера (Controllers/LibraryController.cs).

Реализуйте методы для отображения списка и обработки формы добавления.

```
using Microsoft.AspNetCore.Mvc;
using MvcSimpleApp.Models;

namespace MvcSimpleApp.Controllers
{
    public class LibraryController : Controller
    {
        // Статический список имитирует базу данных
        private static List<Book> _books = new List<Book>
        {
            new Book { Id = 1, Title = "Преступление и наказание", Author = "Ф. Достоевский" }
        };

        // Отображение списка
        public IActionResult Index() => View(_books);

        // Форма добавления (GET)
        [HttpGet]
        public IActionResult Create() => View();

        // Обработка данных формы (POST)
        [HttpPost]
        public IActionResult Create(Book newBook)
        {
            if (ModelState.IsValid)
            {
                newBook.Id = _books.Count + 1;
                _books.Add(newBook);
                return RedirectToAction("Index");
            }
            return View(newBook);
        }
    }
}
```

Шаг 3: создание представлений (Views).

1 Список книг (Views/Library/Index.cshtml):

```
HTML
@model IEnumerable<MvcSimpleApp.Models.Book>
```

```
<h2>Библиотека</h2>
<table class="table table-striped">
   <thead>
      <tr><th>ID</th><th>Автор</th><th>Название</th></tr>
   </thead>
   <tbody>
      @foreach (var item in Model) {
         <tr><td>@item.Id</td><td>@item.Author</td><td>@item.Title</td></tr>
      }
   </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Добавить книгу</a>
```

2 Форма создания (Views/Library/Create.cshtml):

HTML

```
@model MvcSimpleApp.Models.Book

<h3>Добавление новой книги</h3>
<form asp-action="Create" method="post">
   <div class="form-group">
      <label asp-for="Author"></label>
      <input asp-for="Author" class="form-control" />
      <span asp-validation-for="Author" class="text-danger"></span>
   </div>
   <div class="form-group">
      <label asp-for="Title"></label>
      <input asp-for="Title" class="form-control" />
      <span asp-validation-for="Title" class="text-danger"></span>
   </div>
   <br />
   <button type="submit" class="btn btn-success">Сохранить</button>
</form>

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

Контрольные вопросы

1 Какую роль играет компонент Model в решении задачи контроля корректности данных?

2 Почему для отображения списка и сохранения книги используются разные HTTP-методы в контроллере?

3 Каким образом ModelState.IsValid защищает приложение от некорректного ввода?

4 Какую роль играет компонент Model в решении задачи контроля корректности данных?

5 В чем преимущество разделения методов на HttpGet и HttpPost?

6 Как подключить клиентскую валидацию в представлении?

6 Лабораторная работа № 6. Удаленное выполнение запросов к базе данных на основе использования Entity Framework Core

Цель работы: изучить архитектуру ORM (Object-Relational Mapping) на примере технологии Entity Framework Core (EF Core); научиться создавать модель данных, контекст базы данных, выполнять запросы к базе данных с помощью синтаксиса LINQ, выполнять миграции.

Теоретические сведения

Entity Framework Core – это кроссплатформенная ORM-система для .NET. Она позволяет разработчикам работать с базой данных, используя объекты .NET, избавляя от необходимости писать большую часть кода доступа к данным вручную. Ключевые компоненты технологии EF Core:

– DbContext: класс, представляющий сессию с базой данных. Через него осуществляются запросы и сохранение изменений;

– DbSet<T>: представляет коллекцию сущностей определенного типа в базе данных (таблицу);

– модель данных (Entities): классы C#, которые маппятся на таблицы БД;

– LINQ (Language Integrated Query): синтаксис для написания запросов в коде C#, который EF Core транслирует в SQL.

Удаленное выполнение запросов. В контексте данной работы под «удаленным» подразумевается архитектура, при которой приложение и СУБД разделены сетевым взаимодействием. EF Core формирует SQL-запрос на стороне клиента (приложения) и отправляет его на сервер БД по протоколу TCP/IP.

Более подробные сведения см. в [1, с. 432–468].

Задания

1 Развернуть экземпляр СУБД (PostgreSQL или MS SQL Server) в Docker-контейнере или использовать предоставленный преподавателем удаленный сервер.

2 Создать ASP.NET Core Web API проект (или Console Application).

3 Реализовать модель данных предметной области (вариант выбирается по номеру в журнале).

4 Настроить DbContext и строку подключения для работы с удаленной БД.

5 Реализовать миграции для создания схемы БД.

6 Написать минимум пять различных LINQ-запросов (выборка, фильтрация, сортировка, агрегация, соединение таблиц).

7 Включить логирование SQL-запросов и проанализировать сгенерированный код для каждого пункта.

8 Продемонстрировать разницу между запросами, выполняемыми на сервере и на клиенте.

Варианты заданий (предметные области)

1 Библиотека: книги, авторы, читатели, выдачи.
2 Интернет-магазин: товары, категории, заказы, пользователи.
3 Университет: студенты, курсы, преподаватели, оценки.
4 Больница: пациенты, врачи, диагнозы, назначения.
5 Авиакомпания: рейсы, самолеты, пассажиры, билеты.
6 Банк: клиенты, счета, транзакции, валюты.
7 Гостиница: номера, гости, бронирования, услуги.
8 Склад: продукты, поставщики, приходные накладные, складские остатки.
9 Автосервис: клиенты, автомобили, услуги, мастера.
10 Кинотеатр: фильмы, сеансы, залы, билеты.

Для каждого варианта необходимо реализовать связи один-ко-многим и многие-ко-многим.

Порядок выполнения работы

Шаг 1. Подготовка окружения (СУБД).
Для имитации удаленного сервера рекомендуется использовать Docker. Пример для PostgreSQL:

```
1  docker run --name lab-pg -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 -d postgres
```

Запишите параметры подключения: Host (localhost), Port (5432), Database (labdb), Username (postgres), Password (mysecretpassword).

Шаг 2. Создание проекта и установка пакетов.
Создайте проект Web API:

```
1  dotnet new webapi -n EfCoreLab
2  cd EfCoreLab
```

Установите необходимые NuGet-пакеты (пример для PostgreSQL):

```
1  dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
2  dotnet add package Microsoft.EntityFrameworkCore.Design
3  dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Шаг 3. Настройка строки подключения
Откройте файл appsettings.json. Добавьте секцию ConnectionStrings:

"ConnectionStrings": {
"DefaultConnection": "Host=localhost;Port=5432;Database=labdb;Username=postgres;Password=mysecretpassword"
}

Шаг 4. Создание модели и контекста.

1 Создайте папку Models. Добавьте классы сущностей согласно вашему варианту.

2 Создайте папку Data. Добавьте класс ApplicationDbContext, наследующийся от DbContext.

3 Переопределите метод OnModelCreating (при необходимости) и добавьте свойства DbSet<T>.

4 Зарегистрируйте контекст в Program.cs:

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Шаг 5. Миграции и создание БД.

Выполните команды в терминале:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

Убедитесь, что таблицы появились в БД (можно использовать DBeaver или pgAdmin).

Шаг 6. Реализация запросов.

Создайте контроллер или сервис для выполнения запросов. Реализуйте методы для задания.

Пример запроса:

```
var result = await _context.Products
    .Where(p => p.Price > 100)
    .OrderBy(p => p.Name)
    .ToListAsync();
```

Шаг 7. Анализ SQL-запросов.

Для просмотра SQL-запросов в консоли включите чувствительное логирование в Program.cs:

```
options.UseNpgsql(...)
       .EnableSensitiveDataLogging() // Показывает параметры запроса
       .LogTo(Console.WriteLine, LogLevel.Information); // Вывод в консоль
```

Запустите проект, выполните запросы через Swagger/Postman и сохраните вывод SQL из консоли.

Шаг 8. Исследование Client-side evaluation.

Попробуйте использовать метод, который не поддерживается трансляцией в SQL (например, сложный метод .NET внутри Where).

Пример:

```
// Это может вызвать предупреждение или выполнение на клиенте
var data = _context.Products.Where(p => MyCustomFunction(p.Name)).ToList();
```

Зафиксируйте поведение EF Core. Исключение или предупреждение в работе программы отражается в лог-файле.

Контрольные вопросы

1 В чем разница между подходами Code-First и Database-First в EF Core?

2 Как EF Core обрабатывает отложенное выполнение (Deferred Execution) запросов LINQ?

3 Что такое N+1 проблема в EF Core и как её избежать (метод Include)?

4 Как настроить пул соединений для удаленной базы данных?

5 В каких случаях EF Core вынужден выполнять часть запроса на стороне клиента? Чем это опасно?

6 Как обеспечить безопасность строки подключения в продакшн-среде?

7 Что такое миграции и зачем они нужны?

Список литературы

1 **Лок, Э.** ASP.Net Core в действии : пер. с англ. / Э. Лок. – М. : ДМК Пресс, 2021. – 906 с. : ил.

2 **Фримен, А.** ASP.NET Core MVC 2 с примерами на C# для профессионалов : пер. с англ. / А. Фримен. – 7-е изд. – СПб. : Диалектика, 2019. – 1008 с. : ил.