**ASP.Net** – это очень сложная и глубокая технология, рассмотреть за несколько лекций её просто невозможно, поэтому мы будем останавливаться только на значимых моментах без особенных подробностей.

**Слайд№1**

После выпуска ASP.NET с .NET Framework в 2002 году, ASP.NET Core (первая версия, выпущена в 2016 году) была полностью переписана, что не только предполагает работу в Linux, но и использует современные шаблоны (например, DI) а также предлагает новые способы создания веб-приложений.

Razor Pages предоставляет легкий способ создания HTML страниц с интегрированным c# кодом и поддержкой DI.

Blazor предлагает вариант full stack .NET. Вместо написания кода JavaScript вы можете написать код C#, который выполняется либо на сервере (Blazor Server), либо на клиенте в WebAssembly (Blazor WASM). Blazor основан на компонентах Razor, которые расширяют функциональные возможности Razor Pages.

Для начала разберем базовые составляющие веб-технологий, которые в дальнейшем понадобятся для работы.

**Слайд№2**

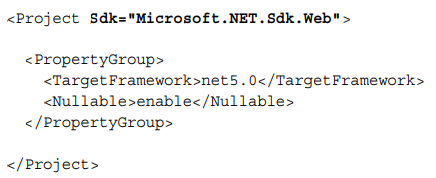
*По слайду…*

**Слайд№3**

*По слайду…*

**SimpleConsoleHost**

Чтобы превратить консольное приложение в простейший хост, отвечающий на HTTP запросы, для начала нужно изменить Sdk в свойствах проекта:



Мы реализуем простейший хост – используем top-level statements, вызовем метод Start класса WebHost. Метод принимает RequestDelegate. RequestDelegate — это делегат, который получает HttpContext в качестве параметра и возвращает Task. HttpContext можно использовать для чтения запроса от клиента и отправки ответа. В нашем примере Httpcontext считывает запрос и отправляет ответ со строкой. Метод WaitForShutdownAsync запускает задачу и ожидает либо комбинации Ctrl+C либо SIGTERM для остановки приложения.

Когда вы запускаете веб-приложение, то открывается консоль, куда выводится лог запросов.

*appsettings.json*

Чтобы увидеть вывод журнала при запуске приложения (который также показывает номера портов, которые прослушивает сервер Kestrel), вы можете добавить файл параметров приложения appsettings.json

Вы можете изменить конфигурацию класса WebHost, используя StartWith, где IApplicationBuilder можно использовать для дальнейшей настройки.

*Properties\launchsetting.json…*

Для веб-проекта Visual Studio создает файл *launchsetting.json.* С помощью этого файла вы можете указать переменные среды, которые используются при запуске приложения, а также URL-адреса, используемые сервером Kestrel.

**SampleWebApp**

Создадим web-приложение посложнее, рассмотрим некоторые возможности ASP.NET Core и разберем структуру проекта.

*Program.cs…*

Жизненный цикл сервиса определяют следующие три метода:

1. AddTransient  
   Transient подразумевает, что сервис создается каждый раз, когда его запрашивают. Этот жизненный цикл лучше всего подходит для легковесных, не фиксирующих состояние, сервисов.
2. AddScoped  
   Scoped - сервис создается единожды для каждого запроса.
3. AddSingleton  
   Singleton - сервис создается при первом запросе (или при запуске ConfigureServices, если вы указываете инстанс там), а затем каждый последующий запрос будет использовать этот же инстанс.

*Use…*

Интерфейс IApplicationBuilder используется для добавления ПО промежуточного слоя(middleware) в конвейер HTTP-запросов. Когда вы вызываете метод Use этого интерфейса, вы можете построить конвейер запросов HTTP, чтобы определить, что должно быть сделано в ответ на запрос.

*endpoints.MapGet("/", async context =>…*

Пример приложения содержит страницу входа, где все функции, отображаемые приложением, могут быть легко доступны с помощью ссылок HTML.

*HtmlExtensions.cs…*

Класс HTMLExtensions определен для создания определенного HTML-кода и уменьшения объема необходимого HTML-кода. Этот класс определяет методы расширения для создания элементов div, span и li.

*Program.cs - app.UseStaticFiles();*

Обычно вы отправляете клиенту простые строки. По умолчанию простые HTML-файлы и другой статический контент отправлять нельзя. ASP.NET Core максимально снижает накладные расходы. Даже статические файлы не возвращаются с сервера, если вы их не включите. Чтобы включить статические файлы, обслуживаемые веб-сервером, вы можете добавить метод расширения UseStaticFiles для добавления необходимого промежуточного программного обеспечения. Это промежуточное ПО проверяет, соответствует ли запрос существующему файлу.

*wwwroot - hello.html*

Папка, в которую следует добавлять статические файлы, — это папка wwwroot в проекте. Если не указать UseStaticFiles, то по запросу html страницы возвращаться не будут.

*libman.json*

Если вы хотите использовать js библиотеки, нужно установить менеджер библиотек Libman:

> dotnet tool install microsoft.web.librarymanager.cli –g

После инициализировать его для проекта, создастся файл libman.json

Установка библиотек происходит по команде

> libman install jquery

По-умолчанию в проекте сразу установлен jQuery и bootstrap.

**Создание пользовательских middleware**

Когда вы вызываете метод расширения UseStaticFiles с помощью IApplicationBuilder, как показано выше, реализуется промежуточное ПО(middleware).

Это промежуточное ПО проверяет запрос на доступность физического файла. Если это так, возвращается этот файл. В противном случае вызывается следующее промежуточное ПО. Промежуточное ПО реализовано как конвейер – одно за другим. На middleware реализованы аутентификация и авторизация, обработка сеансов, кэширование и другие функции.

Вы можете реализовать пользовательские функциональные возможности промежуточного программного обеспечения, вызвав метод Use. Параметр является делегатом с RequestDelegate в качестве параметра и типом возвращаемого значения. RequestDelegate — это делегат, определяющий HttpContext как параметр, возвращающий задачу.

*app.Use((context, next) =>…*

В реализованном вызове метода Use переменная next является параметром RequestDelegate. Этот параметр ссылается на лямбду с параметром HttpContext и возвращает задачу. При реализации лямбда в ответ HTTP записывается настраиваемый заголовок с именем CustomHeader1, а затем вызывается следующее промежуточное ПО, определенное с помощью next.

*Program.cs - app.UseHeaderMiddleware();*

Вместо реализации промежуточного программного обеспечения в качестве параметра метода Use можно создать класс, например, класс HeaderMiddleware. С классом промежуточного программного обеспечения конструктор получает следующее промежуточное программное обеспечение с параметром RequestDelegate.

*HeaderMiddleware.cs*

С помощью метода Invoke вы реализуете функциональность промежуточного программного обеспечения и вызываете следующее middleware, пересылающее HttpContext. Пример кода записывает собственный HTTP-заголовок в HTTP-ответ, аналогичный предыдущему middleware.

*HeaderMiddleware.cs - HeaderMiddlewareExtensions*

Чтобы упростить регистрацию промежуточного программного обеспечения, можно определить метод расширения для IApplicationBuilder. Метод UseHeaderMiddleware вызывает метод UseMiddleware, передавая HeaderMiddleware в качестве универсального параметра.

Запустим проект и убедимся, что срабатывает middleware – прописывает в контекст доп. Заголовок



**Это middleware добавляется аналогично тому, как добавляются все middleware. Поскольку метод расширения middleware UseHeaderMiddleware добавляется после UseStaticFiles, со статическими файлами эта информация заголовка не возвращается клиенту.**

**Маршрутизация конечных точек (Endpoint routing)**

Маршрутизация эндпоинтов определяет, какой адрес запроса соответствует коду обработчика. Маршрутизация реализуется как middleware, подключается методом UseEndpoints.

UseRouting использует EndpointRoutingMiddleware, где принимаются решения о маршрутизации; он определяет, какой маршрут соответствует какой реализации. Однако middleware, указанное перед маршрутизацией, может изменить конечную точку для запроса. Например, если запрос отклонен из-за того, что пользователь не аутентифицирован, маршрут для этого запроса изменяется, например, на страницу с неавторизованной информацией.

Для определения маршрутов различные технологии предлагают различные методы расширения интерфейса IEndpointRouteBuilder. В Razor Pages метод MapRazorPages определяет маршрутизацию для ссылки на Razor Pages в папке Pages, где ссылка сопоставляется с именем Razor Page. В SignalR метод MapHub сопоставляет указанную ссылку с классом, производным от класса концентратора.

В предыдущих примерах кода вы видели, как определить маршрут с помощью метода Map (или MapGet). Использование строки с «/» в качестве аргумента определяет строку запроса от корневого элемента. Map — это метод расширения для интерфейса IEndpointRouteBuilder, который является типом параметра метода UseEndpoints.

*Program.cs - endpoints.Map("/session", async context*

Следующий вызов метода Map определяет маршрут к URL-адресу /session. Если URL-адрес соответствует этой ссылке, служба SessionSample извлекается из контейнера внедрения зависимостей и вызывается метод SessionAsync.

*Program.cs - endpoints.Map("/randr/{action?}", async context =>*

В следующем фрагменте кода показано использование шаблона строки с параметром маршрута. Сегмент URI, который следует за /randr/, сопоставляется c маршрутом с ключом action.

Например для строки запроса /randr/header методом GetRouteValue можно получить часть запроса, отвечающую за ключ

*Program.cs - endpoints.Map("/add/{x:int}/{y:int}", async context =>*

**Ограничения маршрутизации**

С помощью параметра шаблона вы можете указать ограничения, как определено в следующем фрагменте кода. Здесь два обязательных сегмента URI, которые следуют за /add, должны сопоставляться со значениями int; в противном случае этот метод Map не имеет соответствия для маршрута, и следующие определения маршрута будут проверяются на соответствия. GetRouteValue возвращает объект, допускающий значение NULL, содержащий строковое значение, и вам необходимо преобразовать его в нужный вам тип.

*Program.cs - (null, "GET") => service.GetRequestInformation(context.Request) Program.cs - ("header", "GET") => service.GetHeaderInformation(context.Request)*

**Запрос и ответ**

В HTTP протоколе клиент отправляет HTTP запрос на сервер и получает ответ.

Запрос содержит заголовок с множеством полей и чаще всего тело запроса. Сервер использует информацию из заголовков, чтобы понять что клиент хочет и в зависимости от этого сформировать соответствующий ответ.

Метод *GetRequestInformation – RequestAndResponseSamples* использует HttpRequest для доступа к Scheme, Host, Path, QueryString, Method и Protocol

*GetRequestInformation – GetHeaderInformation*

Чтобы получить доступ к заголовкам HTTP, объект HttpRequest определяет свойство Headers типа IHeaderDictionary которое содержит имя заголовка и строковый массив значений.

**Что тут полезного?**

В HTTP/2 заголовки прав доступа, методов, путей и схем теперь имеют префикс :. Некоторые заголовки, используемые в HTTP/1.1, например заголовок Connection, больше не нужны в HTTP/2.

Заголовок Accept определяет форматы многоцелевых расширений почты Интернета (MIME), которые браузер принимает. MIME первоначально использовался с вложениями электронной почты, но теперь имеет более широкое применение. Список упорядочен по предпочтительным форматам. В зависимости от этой информации вы можете принять решение о возврате данных в различных форматах в зависимости от потребностей клиента. Edge предпочитает HTML, затем XHTML и XML, затем WEBP и APNG. С некоторой частью этой информации также определяется квантификатор. Браузеры, используемые для вывода, имеют \*.\* в конце этого списка, чтобы принять все возвращаемые данные.

Заголовок User-Agent использовался в древние времена HTML, чтобы отличать код, возвращаемый клиенту. Существовали файлы конфигурации, в которых перечислены возможности конкретных браузеров. Поскольку в более новых версиях браузеров это часто не срабатывало, а некоторые браузеры позволяют настраивать эту строку, она больше не используется. Просто проверьте строку User-Agent в браузере Edge, который помечает себя как Mozilla, AppleWebKit, Gecko, Chrome, Safari и Edge. Вместо использования заголовка пользовательского агента просто динамически проверяйте возможности браузера при работе с кодом JavaScript.

Информация заголовка Accept-Language показывает языки, настроенные пользователем. Вы можете использовать эту информацию для возврата локализованной информации.

Информация заголовка sec-fetch-xx относится к заголовкам запросов на выборку метаданных HTTP/2. sec-fetch-site используется с общим доступом к ресурсам между источниками (CORS). sec-fetch-mode определяет как был инициирован запрос. sec-fetch-user предоставляет информацию о том, был ли запрос инициирован пользователем. ?1 верно, ?0 ложно

sec-fetch-dest определяет назначение запроса на сервер. В HTML-коде, в котором была инициирована навигация, значение равно document. Другими значениями являются script, serviceworker, audio, image и другие.

Информация заголовка, которую вы видели до сих пор, — это то, что браузер отправляет для очень простых сайтов. Обычно будет больше деталей, таких как файлы cookie, информация для аутентификации и пользовательская информация. Чтобы просмотреть всю информацию, которая отправляется на сервер и с сервера, включая информацию заголовка, вы можете использовать инструменты разработчика браузера.

*RequestAndResponseSamples – QueryParameters*

Следующий метод QueryParameters используется для извлечения параметров с именами x и y из строки запроса. Если синтаксический анализ параметров в значения int завершается успешно, выполняется вычисление. В зависимости от ввода возвращается разный HTML-код.

Коллекция IQueryCollection, возвращаемая из строки запроса, также позволяет получить доступ ко всем ключам с помощью свойства Keys и предлагает метод ContainsKey для проверки доступности указанного ключа.

**Данные форм**

*Program.cs - ("form", "GET" or "POST") => service.Form(context.Request)*

Вместо того, чтобы передавать данные от пользователя на сервер с помощью строки запроса, вы можете использовать HTML-элемент формы.

В этом примере вместо GET используется HTTP-запрос POST. При запросе POST пользовательские данные передаются вместе с телом запроса, а не в строке запроса.

Использование данных формы определяется двумя запросами. Сначала форма отправляется клиенту с помощью GET-запроса, а затем пользователь заполняет форму и отправляет данные с помощью POST-запроса. Как показано в следующем фрагменте кода, метод Form вызывает метод GetForm или ShowForm в зависимости от типа метода HTTP.

Форма создается с помощью элемента ввода с именем text1 и кнопки «Отправить». Нажатие кнопки «Отправить» вызывает метод действия формы с методом HTTP, как определено с помощью аргумента метода

Для чтения данных формы класс HttpRequest определяет свойство Form. Это свойство возвращает объект IFormCollection, содержащий все данные из формы, отправляемой на сервер.

По ссылке /randr/form форма получена GET-запросом. Когда вы нажимаете кнопку «Отправить», форма отправляется с запросом POST, и вы можете увидеть ключ text1 для данных формы.

**Куки**

*RequestAndResponseSamples – WriteCookie*

Чтобы запомнить пользовательские данные между несколькими запросами, вы можете использовать файлы cookie. Добавление файла cookie к объекту HttpResponse отправляет файл cookie в заголовке HTTP с сервера клиенту.

По умолчанию файл cookie является временным (не сохраняется на клиенте), и браузер отправляет его обратно на сервер, если URL-адрес совпадает с доменом, из которого был получен файл cookie.

Вы можете установить путь для ограничения, когда браузер возвращает файл cookie. В этом случае cookie возвращается только в том случае, если он исходит из того же домена и используется путь /randr.

Когда вы устанавливаете свойство Expires, файл cookie является постоянным файлом cookie, что означает, что он хранится на клиенте. По истечении срока действия файл cookie будет удален. Однако нет гарантии, что файл cookie не будет удален ранее.

*RequestAndResponseSamples – ReadCookie*

Файл cookie можно снова прочитать, прочитав объект HttpRequest. Свойство Cookies содержит все файлы cookie, возвращаемые браузером.

**Отправка JSON**

Сервер возвращает больше, чем код HTML; он также возвращает множество форматов данных, таких как файлы CSS, изображения и видео. Клиент знает, какие данные он получает с помощью типа MIME в заголовке ответа.

Метод GetJson создает строку JSON из анонимного объекта со свойствами Title, Publisher и Author. Тип MIME для формата JSON — application/json. Это устанавливается через свойство ContentType HttpResponse.

**Состояние сессии**

Middleware поддерживает состояние сессии, сессия хранит временные данные от клиента. Сессия сама по себе это middleware.

Сессия(или сеанс) инициируется, когда пользователь впервые запрашивает страницу с сервера. Пока пользователь продолжает открывать страницы на сервере, сеанс продолжается до тех пор, пока не истечет время ожидания (обычно 10 минут). Чтобы сохранить состояние на сервере, пока пользователь переходит на новую страницу, состояние можно записать в сеанс. По истечении времени ожидания данные сеанса удаляются.

Для идентификации сеанса при первом запросе создается временный файл cookie с идентификатором сеанса. Этот файл cookie возвращается клиентом при каждом запросе к серверу до закрытия браузера, а затем файл cookie удаляется. Идентификаторы сеанса также могут быть отправлены в строке URL в качестве альтернативы использованию файлов cookie.

На стороне сервера информация о сеансе может храниться в локальной памяти. В веб-ферме состояние сеанса, хранящееся в локальной памяти, не распространяется между разными системами. В конфигурации с фиксированным сеансом пользователь всегда возвращается на один и тот же физический сервер, поэтому у пользователя есть состояние, доступное на сервере, если только сервер не выйдет из строя. Если сервер не настроен на использование закрепленных сеансов, клиентский запрос может быть отправлен на любой экземпляр сервера. С такой конфигурацией вы можете хранить состояние сеанса в распределенной памяти или базе данных. Хранение состояния сеанса в распределенной памяти также помогает перезапускать серверный процесс; перезапуск убивает состояние сеанса, если вы используете только один серверный процесс.

Чтобы включить сеансы, необходимо настроить промежуточное ПО и контейнер DI. В контейнере DI вам необходимо зарегистрировать интерфейс ISessionStore, который используется промежуточным программным обеспечением.

*Program.cs - builder.Services.AddDistributedMemoryCache();*

В следующем фрагменте кода показана регистрация контейнера DI. Метод AddSession — это метод расширения, который регистрирует ISessionStore в классе реализации DistributedSessionStore. Классу DistributedSessionStore требуется объект, реализующий IDistributedCache с помощью конструктора. Это регистрируется методом расширения AddDistributedMemoryCache. С помощью параметров метода AddSession вы можете настроить время простоя и параметры файлов cookie. Файл cookie используется для идентификации сеанса

*Program.cs - app.UseSession();*

Вторая часть — настроить ПО промежуточного слоя в конвейере, вызвав метод расширения UseSession.

Вам нужно вызвать этот метод до того, как будет записан какой-либо ответ, где может потребоваться сеанс — например с UseHeaderMiddleware — таким образом, UseSession вызывается перед другими методами.

*SessionSample – SessionAsync*

Вы можете записать состояние сеанса, используя методы Setxxx, такие как SetString и SetInt32. Эти методы определяются с помощью интерфейса ISession, который возвращается из свойства Session объекта HttpContext. Данные сеанса извлекаются с использованием методов Getxxx.

Задеплоить веб приложение можно в докере

[*https://docs.docker.com/samples/dotnetcore/*](https://docs.docker.com/samples/dotnetcore/)

## Для этого нужно в проект добавить файл **Dockerfile, затем выполнить**

docker build . -t WebSampleApp/v1.0 > docker tag WebSampleApp/v1.0 {linktoyourregistry}/WebSampleApp/v1.0 > docker push {linktoyourregistry}/WebSampleApp/v1.0

понадобится создать регистр

$ docker run -d -p 5000:5000 --restart=always --name registry registry:2

**RazorPagesSampleWebApp**

MVC расшифровывается как Model-View-Controller и представляет собой шаблон проектирования, описывающий структуру приложения.

Шаблон MVC подчеркивает разделение задач, когда функциональность определяется независимо от данных и интерфейса. Это особенно важно в web-разработке, когда изначально можно поместить все на одной странице и в принципе все будет работать.

Ранние версии MVC Framework были построены на основе ASP.NET, изначально предназначенной для веб-страниц, что приводило к некоторым неудобным функциям и обходным путям. С переходом на .NET Core ASP.NET стал ASP.NET Core, а MVC Framework была перестроена на открытой, расширяемой и кросс-платформенной основе.

Одним из недостатков MVC Framework является то, что может потребоваться много подготовительной работы, прежде чем приложение сможет начать выдавать некий контент. Несмотря на структурные проблемы, одним из преимуществ веб-страниц было то, что простые приложения можно было создать за пару часов.

Razor Pages использует принципы разработки Web Pages и реализует их с использованием функций платформы, изначально разработанных для MVC Framework. Код и контент смешиваются, образуя автономные страницы; это воссоздает скорость разработки веб-страниц.

Используем шаблон проекта ASP.Net Core.

*Program.cs…*

В первую очередь нам понадобится настроить WebApplication для работы с RazorPages

builder.Services.AddRazorPages();

app.UseStaticFiles();

app.MapRazorPages();

**Маршрутизация RP**

В конфигурации по умолчанию маршрутизация Razor Pages использует папку **Pages** и вложенные папки и сопоставляет URL-адрес с файлами .cshtml в этой папке.

Вы можете переопределить параметры с помощью методов AddRazorOptions и AddRazorPageOptions. Если вы обращаетесь к URL-адресу /Hello, выполняется поиск страницы Pages/Hello.cshtml.

С URL-адресом /Admin/User ожидается страница Pages/Admin/User.cshtml. Если эти страницы не найдены, поиск продолжается в папке Views/Shared. Когда вы устанавливаете свойство PageViewLocationFormats с помощью метода AddRazorOptions, это поведение можно изменить. Просто изменив папку Pages на другую папку, метод AddRazorPagesOptions можно использовать для установки свойства RootDirectory RazorPagesOptions.

**Cлайд №4**

Страницы Razor используют синтаксис Razor. Синтаксис Razor позволяет смешивать код HTML и C#. Razor использует символ @ в качестве символа перехода для переключения с HTML на C#.

Возвращаемое значение можно использовать непосредственно с неявным выражением Razor. Например, ViewData["Title"] возвращает строку. Возвращаемая строка помещается непосредственно между тегами заголовка HTML, как показано на слайде. После этого выражения Razor переключается на HTML, а следующая строка перед закрывающим тегом title представляет собой простой HTML

В некоторых ситуация конец C# кода нельзя точно установить, тогда нужно использовать круглые скобки – это **явное Razor выражение.**

Если в HTML коды вы вызываете выражение, которое возвращает void или не возвращает ничего, то необходимо использовать фигурные скобки.

Также можно использовать конструкции @if, else if, else, @switch и операторы циклов @for, @foreach, @while, @do while

**Cлайд №5**

Вместо импорта пространств имен с каждой страницей Razor вы можете импортировать пространства имен с помощью файла \_ViewImports.cshtml, определенного в папке Pages с помощью @using. Чтобы объявить пространство имен для типов, определенных в папке, вы можете использовать @namespace

**Компоненты Razor страницы**

*Pages - Index.cshtml…*

Razor Pages имеют расширение файла .cshtml и начинаются с директивы @page. Это простая страница Razor с HTML-кодом. Также Razor-страница может быть создана со встроенным кодом, для этого нужно использовать блок @functions

В Razor Page создается класс, производный от базового класса Microsoft.AspNetCore.Mvc.RazorPages .Page. Директива @model использует универсальную версию этого базового класса и указывает тип модели в качестве универсального параметра. При этом свойство Model сгенерированного класса можно использовать для доступа к данным базового кода.

**Layout(Макеты)**

**Разберем подробнее взаимодействие макета Layout и страницы Index.chtml**

*Pages - ViewStart.cshtml…*

Обычно многие страницы веб-приложений имеют один и тот же контент, например, информацию об авторских правах, логотип и основную структуру навигации. Используя страницы макета, вы можете совместно использовать код HTML с различными страницами Razor.

Чтобы использовать макет, в базовом классе Razor Page PageModel определяется свойство Layout. Чтобы указать параметры по умолчанию для Razor Pages, используйте файл \_ViewStart.cshml.

В нашем примере шаблон установлен в *\_Layout.cshtml*

Страница макета содержит объявление HTML, элементы html, head и body, а также элементы заголовка и нижнего колонтитула body. Конечно, вы также можете использовать синтаксис Razor в этом файле. С расширением файла .cshtml (и без директивы @page) класс, используемый за кулисами для страницы макета, — это класс RazorPage.

Вызов метода RenderBody важен для страниц макета. Этот метод реализован с помощью базового класса и отображает содержимое страницы Razor.

<div class="container">

<main role="main" class="pb-3">

@RenderBody()

</main>

</div>

**Передача данных между представлениями**

Простой способ передавать данные между представлениями – использовать ViewData, это словарь параметров текущего представления. В нашем случае ViewData["Title"] = "Home page"; используется в \_Layout <title>@ViewData["Title"] - RazorPagesSampleWebApp</title>.

Тип словаря dynamic, поэтому Интелисенс не распознает данные в словаре на этапе разработки.

**Разделы**

*\_Layout.cshtml*

Если вы хотите разместить содержимое своей страницы в разных частях макета, вы можете отображать разделы. Для этого вы вызываете метод RenderSectionAsync на странице макета, как показано в следующем фрагменте кода.

@await RenderSectionAsync("Keywords", required: false)

@await RenderSectionAsync("Scripts", required: false)

Если вы не хотите, чтобы на каждой странице был раздел «Scripts», для аргумента required установлено значение false.

В элементе head страницы макета имеется ссылка на раздел с именем Keywords . В элементе head каждая страница может добавлять мета-значения ключевых слов HTML.

Страница Razor теперь может предоставлять разделы, как показано в следующем фрагменте кода, используя директиву Razor @section и используя имя раздела.

@section Keywords {

<meta name="keywords" content="C#, .NET, Azure">

}

**Передача параметров по маршруту**

Маршрутизация с помощью Razor Pages определяется простым соглашением: имя файла Razor Page в папке Pages используется с URI.

В дополнение к этому вы можете передавать параметры на страницу, как показано на следующей встроенной странице Razor.

В Razor Pages в зависимости от отправляемых HTTP-команд вы указываете методы OnGet и OnPost. Параметры, которые вы указываете с помощью этих методов, сопоставляются с запросом.

*Pages-ShowBook…*

Здесь параметры названия и издателя назначаются свойствам Title и Publisher, которые, в свою очередь, используются с содержимым HTML с неявными выражениями Razor, обращающимися к свойствам.

Блок @functions тут заменяет файл модели, можно перенести его в связанный файл ShowBook.cshtml.cs

Теперь мы можем перейти по ссылке <https://localhost:7246/ShowBook?title=CSharp&publisher=Book>

Вместо использования параметров с методом OnGet вы также можете получить доступ к свойству RouteData базового класса, чтобы получить доступ ко всем значениям маршрута.

*Pages-Calc.cshtml…*

C @page можно создать пользовательский маршрут для связки параметров, шаблон можно использовать в связке с регексом.

@page "{op:regex(^[add|sub|mul|div])}/{x:int}/{y:int}"

**Areas**

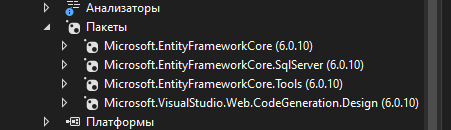
При использовании областей под папкой «Areas» вы добавляете вложенную папку с названием категории (например, «Администратор» или «Книги»), а затем вложенную папку Pages, в которой хранятся ваши Razor Pages.

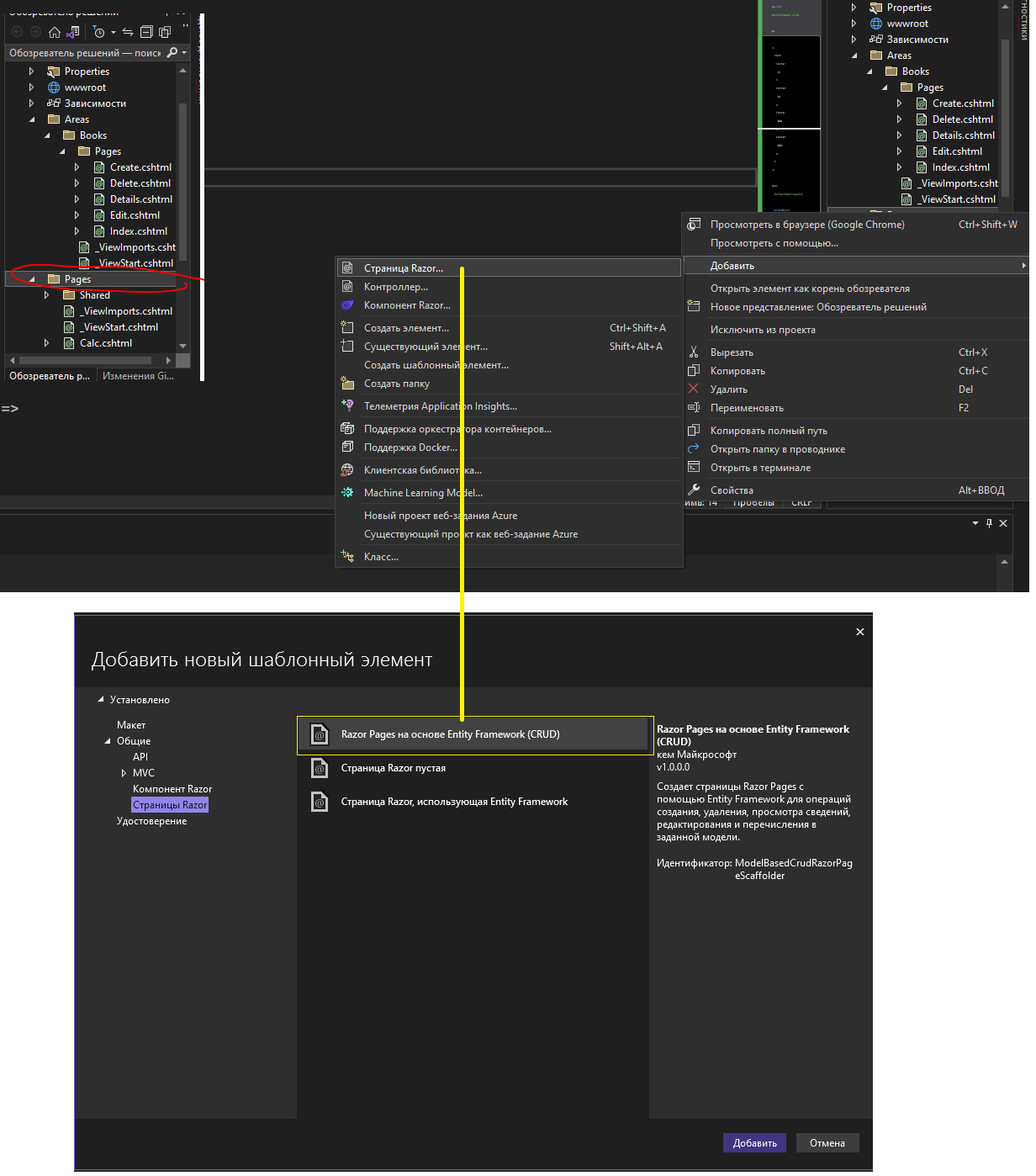
Пример приложения определяет страницы Index, Details, Create, Edit и Delete Razor для чтения и записи объектов Book.

Само слово Areas не добавляется в URL при запросе и позволяет вам вести логическое разделение архитектуры вашего проекта.

**Создание простого CRUD интерфейса**

1. Создадим структуру Areas-Books-Pages
2. Подключим пакеты



1. Используем EF Core Power Tools - Reverse Engineer, создадим модель и контекст доступа к БД с книгами
2. После создадим страницы Razor для CRUD операций
3. Добавим подключение к БД через DI builder.Services.AddSqlServer<CSharpBooksContext>("Data Source=localhost;Initial Catalog=CSharpBooks;User ID=sa;Password=HelloWorld10");

**Рассмотрим некоторые аспекты CRUD операций**

**Create**

*Books-Pages-Create.cshtml.cs*

При создании новых книг сначала делается запрос HTTP GET, чтобы получить форму для заполнения данных.

Когда данные заполняются пользователем, запрос HTTP POST отправляет данные книги с телом HTTP на сервер.

Если от клиента отправляется запрос GET, вызывается метод OnGet на странице Create. Здесь возвращается только пустая страница.

При отправке запроса GET пользователь получает форму, может заполнить данные и отправить форму, нажав кнопку отправки, выполнится пост запрос.

*[BindProperty]*

*public Book Book { get; set; }*

Для доступа к данным, полученным с помощью POST-запроса, можно использовать атрибут BindProperty.

BindProperty использует связывание модели (интерфейс IModelBinder) для присвоения значений из данных формы типу, которому назначен атрибут; в следующем фрагменте кода свойство Book типа Book снабжено аннотацией атрибута BindProperty. При использовании метода OnPostAsync, который вызывается с запросом POST, полученное свойство Book используется для добавления новой записи в контекст EF Core и записи новой записи в базу данных. В случае успеха в браузер отправляется HTTP-запрос перенаправления, поэтому браузер продолжает GET-запрос на страницу индекса.

**Read**

*Books-Pages-Index.cshtml*

Вместо того, чтобы просто использовать код HTML с синтаксисом Razor с выражениями, обращающимися к модели, вы можете использовать HTML Helpers.

На странице Razor сгенерированный класс содержит свойство Html типа IHtmlHelper или, если используется директива @model, свойство Html универсального типа IHtmlHelper<Model>. В этом интерфейсе доступно несколько HTML хелперов, которые возвращают HTML-код. Вы также можете создать собственный HTML хелпер, определив метод расширения, который расширяет интерфейс IHtmlHelper и возвращает строку.

DisplayNameFor использует лямбда-выражение для определения свойства, которое следует использовать для доступа к имени свойства. Затем **имя** свойства возвращается с кодом HTML. Метод DisplayFor использует то же выражение, но возвращает **значение** свойства. Вспомогательная функция HTML DisplayNameFor используется в заголовке таблицы HTML, метод DisplayFor в итерации @foreach для отображения каждого значения коллекции.

**Tag-Helpers**

HTML хелперы были доступны с ранних версий ASP.NET MVC. Вспомогательные функции тегов — это новая конструкция, доступная в ASP.NET Core. Вместо использования синтаксиса Razor для активации вспомогательных функций HTML с помощью вспомогательных функций тегов вы записываете синтаксис HTML на своих страницах Razor. Вспомогательные функции тегов резолвятся на сервере с помощью серверного кода, а HTML и JavaScript возвращается клиенту.

<td>

<**a** **asp-page**="./Edit" **asp-route-id**="@item.BookId">Edit</**a**> |

<**a** **asp-page**="./Details" **asp-route-id**="@item.BookId">Details</**a**> |

<**a** **asp-page**="./Delete" **asp-route-id**="@item.BookId">Delete</**a**>

</td>

рассмотрим пример с использованием Tag-helper. В следующем фрагменте кода HTML-элемент <a> используется для создания ссылки на странице Razor. С этим элементом указываются атрибуты asp-page, asp-route-id и asp-area. За кулисами находится класс AnchorTagHelper со свойствами Page, Area и RouteValues. Эти свойства снабжены аннотациями атрибута HtmlAttributeName со значениями "asp-page", "asp-area" и "asp-route-{value}". С помощью префикса asp- можно легко отличить эти имена атрибутов на стороне сервера от имен атрибутов HTML. Когда вы используете эти атрибуты с AnchorTagHelper, атрибут href возвращается к ссылке соответствующей страницы Razor, включая параметр id.

**Валидация ввода**

Для проверки пользовательского ввода на клиенте вы можете использовать Tag-хелперы. ValidationMessageTagHelper прикрепляет сообщения об ошибках к полям ввода (используя атрибут asp-validation-for). Этот помощник создает атрибут data-valmsg-for HTML 5.

Элементы управления проверки используют аннотации модели, такие как атрибуты Required и StringLength. С атрибутами CreditCard, EmailAddress, Phone и Url вы можете использовать больше атрибутов для проверки ввода с обычно используемыми данными.

**Слайд №6**

**BLAZOR**

Blazor — это новая технология ASP.NET Core для создания интерактивных веб-приложений. С Blazor вы получаете полнофункциональную разработку .NET без необходимости писать код JavaScript. Приложение может быть создано с использованием HTML, C# и CSS — как для клиента, так и для сервера.

Вам нужно понимать разницу между двумя вариантами: Blazor Server и Blazor WebAssembly. Оба варианта предлагают полнофункциональную разработку .NET, и с обоими этими вариантами вы создаете компоненты Razor.

Компоненты Razor являются расширением Razor Pages, С Blazor Server компоненты Razor запускаются на сервере. С Blazor WebAssembly компоненты Razor запускаются на клиенте. Чтобы понять различия, преимущества и недостатки этих вариантов, рассмотрим их более подробно.

**BLAZOR SERVER**

В Blazor Server клиент всегда должен быть подключен к серверу. Вы можете написать серверный код C#, используя компоненты Razor.

Вы напрямую работаете с HTML и C# на сервере и вносите обновления в HTML через привязку данных. Незаметно обновления HTML отправляются клиенту, а клиентская библиотека JavaScript Blazor, с которой вам не нужно работать напрямую, обновляет пользовательский интерфейс.

С Blazor Server соединение (цепь) между клиентом и сервером остается открытым, и клиенту нужно просто запустить HTML и JavaScript. Чтобы поддерживать соединение открытым и обмениваться данными между клиентом и сервером, используется SignalR. SignalR реализует уровень абстракции над WebSockets.

При использовании Blazor Server размер загружаемого клиентом кода значительно меньше, чем при использовании Blazor WebAssembly. С Blazor Server вы можете использовать гораздо больше возможностей сервера.

Недостатком Blazor Server является то, что клиенты всегда подключены. Офлайн-работы нет. Таким образом создается более постоянная нагрузка на сервер и сеть.

**WebAssembly**

Когда вы используете Blazor WebAssembly, вы пишете компоненты Razor, аналогичные Blazor Server, но на этом сходство заканчивается. Вне среды программирования Blazor WebAssembly сильно отличается. Здесь код .NET выполняется на клиенте , вам вообще не нужен сервер, потому что файлы нужно просто передать клиенту. Blazor WebAssembly также можно создать как прогрессивное веб-приложение (PWA), которое может работать без подключения к серверу (после установки приложения на клиенте).

Blazor WebAssembly не использует SignalR. Вместо этого среда выполнения .NET (двоичный файл Wasm) и среда выполнения Blazor отправляются клиенту вместе с библиотеками DLL .NET (код IL) вашего приложения.

Преимущества использования Blazor WebAssembly по сравнению с Blazor Server заключаются в том, что вы можете использовать возможности клиента и вычислительную мощность, а также в том, что приложение может быть установлено как PWA и по-прежнему может работать без доступного сервера. Конечно, есть и недостатки, потому что на клиент нужно загружать больше кода, а среда разработки не такая зрелая, как на стороне сервера. Сегодня отладка кода .NET на стороне клиента работает в большинстве сценариев, но не во всех.

**BlazorSampleServer**

Создадим проект и рассмотрим некоторые аспекты реализации.

После добавления Razor Pages с помощью метода расширения AddRazorPages службы, необходимые для Blazor на стороне сервера, добавляются с помощью метода расширения AddServerSideBlazor.

В middleware статические файлы необходимо настроить для отправки клиенту (UseStaticFiles). Маршрутизация конечной точки указывает, что Blazor берет на себя маршрутизацию по умолчанию (MapBlazorHub); это настраивает маршрут концентратора SignalR, используемый для связи WebSocket от клиента к серверу. В случае, если маршрут не найден, запасной вариант устанавливается на \_Host с помощью MapFallbackToPage.

**Запуск приложения**

После запуска ASP.NET Core с конфигурацией контейнера DI и конфигурацией middleware, следующим шагом для запуска Blazor Server является страница Razor \_Host.cshtml.

*Pages -\_Host.cshtml…*

Этот файл содержит tag-helper как показано в следующем фрагменте кода, который отображает компонент Razor App. С помощью параметров режима рендеринга вы можете указать вывод компонента Razor.

Установка для этого параметра значения **Static** просто отображает HTML, а код Blazor становится не активен. **Server** создает маркеры, которые используются связью Blazor для динамической отправки вывода HTML и JavaScript через SignalR.

Параметр по умолчанию — **ServerPrerendered**, где HTML уже предварительно визуализируется на сервере, а в дополнение к HTML клиенту отправляются маркеры. С ServerPrerendered клиент быстрее увидит первый HTML. Затем маркеры используются для динамических обновлений.

*Pages -\_Layout.cshtml…*

В файле \_Layout.cshtml элемент base играет важную роль в настройке базового маршрута для Blazor. Этот хелпер используется среди других настроек заголовка HTML и включает таблицы стилей, используемые всеми компонентами Razor.

*App.razor…*

App — это первый компонент Razor, который включается при загрузке приложения. Этот компонент включает в себя некоторые другие компоненты Razor и небольшой HTML-код. Элемент p — это HTML; все остальные элементы являются компонентами Razor: Router, Found, NotFound, RouteView и LayoutView.

Компонент Router отвечает за маршрутизацию приложений Blazor. Ранее мы рассматривали создание маршрутов с помощью страниц Razor и директивы @page.

Директива @page также используется с компонентами Razor (но с компонентами Razor директива @page не является обязательной).

Если компонент Razor включает маршрут, он используется компонентом Router для добавления его к возможным совпадениям маршрута. Если совпадение найдено (с помощью компонента Found), для продолжения используется компонент RouteView.

Если совпадение не найдено (см. компонент NotFound), используется компонент LayoutView.

**Макет Blazor**

*Shared-MainLayout.razor…*

В соответствии с определением компонента App, компонент MainLayout, как показано в следующем фрагменте кода, используется для макета по умолчанию.

Этот компонент наследуется от базового класса LayoutComponentBase. LayoutComponentBase определяет свойство Body типа RenderFragment.

RenderFragment позволяет создавать шаблонные компоненты. В MainLayout свойство Body используется для визуализации компонента Razor, переданного из RouteView. В MainLayout вы можете увидеть другой используемый компонент Razor: NavMenu — это компонент, показывающий навигацию по приложению.

MainLayout хранится в папке Shared. Blazor использует те же механизмы, которые вы ранее. Компоненты сначала ищутся в папке Pages, а если их там нет, то поиск продолжается в папке Shared. Таким образом, вы можете переопределить приложение, чтобы использовать разные макеты MainLayout.razor в разных папках Pages и Areas.

**Навигация**

*Shared-NavMenu.razor…*

Как показано в следующем фрагменте кода, этот компонент использует кнопку HTML, в которой событие onclick привязано к методу C# ToggleNavMenu. С Blazor Server, когда пользователь нажимает кнопку HTML на клиенте, происходит связь с сервером, где на сервере запускается метод ToggleNavMenu.

Для сопоставления событий элемента HTML с методами .NET вам просто нужно использовать символ @ в качестве префикса к имени события, чтобы привязать событие к методу .NET.

С компонентами Razor код C# определяется в разделе @code (который отличается от раздела @functions со страницами Razor). В этом разделе кода вы можете увидеть ToggleNavMenu, который переключает значение CollarNavMenu между true и false. NavMenuCssClass возвращает разные значения, коллапс или нал, в зависимости от значения логического поля.

Привязка в другом направлении — от кода C# к HTML — выполняется с помощью атрибута class первого элемента div в следующем фрагменте кода. Класс атрибута HTML привязывается к строке NavMenuCssClass, определенной с помощью сегмента кода. Когда происходит событие (например, событие клика) и изменяется исходный код C#, пользовательский интерфейс обновляется. Элемент div также привязывает событие onclick к методу ToggleNavMenu, поэтому вы можете нажать кнопку или элемент div, чтобы вызвать этот метод.

Компонент NavMenu также содержит компоненты NavLink для создания HTML-ссылок на компоненты Razor, которые соответствуют маршруту по умолчанию, а также маршрутам счетчика и выборки данных, как указано в атрибутах href. Атрибуты href добавляются к элементам привязки (a), которые отображаются из NavMenu.

Компонент NavLink не только создает элемент HTML (тег a), но также переключает активный класс CSS. Установка для свойства Match значения NavLinkMatch.All активирует класс, только если ссылка полностью совпадает. При использовании NavLinkMatch.Prefix по умолчанию класс CSS активируется, если совпадает начало ссылки.

**Компонент Counter**

*Pages - Counter.razor…*

Теперь, когда мы рассмотрели начальную загрузку, макет и навигацию, давайте перейдем к компонентам.

Компонент счетчика — это один из компонентов, созданных с помощью шаблона по умолчанию. Большой разницы с компонентами MainLayout и NavMenu нет.

Компонент Counter начинается с директивы @page, которая определяет ссылку /counter. Компоненты Razor без директивы @page можно использовать только внутри других компонентов.

Поскольку у Counter есть директива @page со ссылкой /counter, доступ к этому компоненту можно получить, передав /counter с URI. Компонент Counter также использует привязку для подключения события onclick кнопки к методу .NET IncrementCount и для получения значения поля currentCount с помощью @currentCount.

Когда вы запускаете приложение, вы можете увидеть компонент счетчика и нажать кнопку, чтобы увеличить счетчик. Когда вы изменяете размер приложения, чтобы сделать его меньше, внешний вид меняется из-за используемой темы Bootstrap. При уменьшении размера приложения кнопка-переключатель, описанная ранее в разделе «Навигация», становится видимой, и вы можете отображать или скрывать меню.

**Компонент Fetch Data**

Еще одним интересным компонентом, созданным с помощью шаблона по умолчанию, является компонент FetchData. Этот компонент внедряет WeatherForecastService с помощью директивы @inject.

В разделе кода в методе OnInitializeAsync вызывается метод GetForecastAsync, который возвращает информацию о погоде на ближайшие несколько дней. С компонентами Razor при инициализации вызывается метод OnInitializeAsync.