Вищий навчальний заклад

Університет економіки та права "КРОК"

Коледж економіки, права та інформаційних технологій

Циклова комісія з інформаційних технологій

ЖОВТЯК І.В., ДОБРИШИН Ю.Є., ГАРКУША В.В.

МЕТОДИЧНИЙ ПОСІБНИК ДЛЯ ПРОВЕДЕННЯ ТЕХНОЛОГІЧНОЇ ПРАКТИКИ ЧАСТИНА 1

«РОЗРОБКА WEB-ЗАСТОСУВАНЬ ASP.NET MVC НА ПЛАТФОРМІ .NET CORE»

(для студентів спеціальності 122 «Комп’ютерні науки» та спеціальності 121 «Інженерія програмного забезпечення»)

Київ 2019 р.

І.В. Жовтяк, Ю.Є. Добришин, В.В. Гаркуша

Методичні посібник для проведення навчальної практики частина 1 «РОЗРОБКА WEB-ЗАСТОСУВАНЬ ASP.NET MVC на платформі .NET Core» для студентів спеціальності 122 «Комп'ютерні науки» та для спеціальності 121 «Інженерія програмного забезпечення» - К.: Університет економіки та права "КРОК". 2019. – 37 с.

Методичні посібник "РОЗРОБКА WEB-ЗАСТОСУВАНЬ ASP.NET MVC на платформі .NET Core" містить опис основних компонент технології ASP.NET MVC Core, структури проекту, засобів валідації даних та аутентифікації користувачів.

Матеріали методичного посібника використовуються під час проведення технологічної практики частина 1, інших практичних та лабораторних занять із студентами спеціальностей 122 «Комп'ютерні науки» та для спеціальності 121 «Інженерія програмного забезпечення».

РОЗГЛЯНУТО І СХВАЛЕНО
Педагогічною радою Коледжу економіки, права та інформаційних технологій
Протокол 1 від 30.08.2019 р.

ЗМІСТ

ВСТУП

Розглядається розробка веб-застосувань *ASP.NET MVC* на платформі *ASP.NET Core* . Платформа *ASP.NET Core* побудована на основі кросплатформенної версії *.NET - .NET Core.* Застосування побудовані на основі *.NET Core* можна розгортати на серверах з ОС Linux і OSX/macOS.

1. РОЗРОБКА ВЕБ-ЗАСТОСУВАНЬ ЗА ТЕХНОЛОГІЄЮ ASP.NET MVC

1.1. Основні компоненти ASP.NET MVC

Абревіатура MVC розшифровується як *Model-View-Controller* (*Модель - Представлення-Контролер*). Технологія ASP.NET MVC є реалізацією на платформі .Net фірми Microsoft популярного в галузі веб-розробки однойменного патерна проектування[1].

Відповідно до концепції MVC, застосування розподіляється на три частини:

• *Модель* – класи, які визначають даних, що передаються від контролера до представленню, і методи для роботи з цими даними.

• *Представлення (відображення)* – зазвичай, інтерфейс, який використовується для обробки і відображенні даних, описаних в моделі.

• *Контролери* - обробляють вхідні запити, виконують операції з даними моделі і вибирають предсталення для їх відображення користувачеві.

Технологія *ASP.NET MVC*, на відміну від попередньої технології *ASP.NET Web Forms*, робить акцент на чистому коді, розподілі компонет та можливості тестування коду. Компанія Microsoft вбудувала в інфраструктуру *MVC Framework* інструменти з відкритим кодом, такі як бібліотеки *jQuery*, *nuGet packages,* врахувала проектні рішення і передовий досвід конкуруючих платформ. Висхідний код *MVC Framework* відкритий, що дозволяє вивчати його іншим розробниками.

Модель даних

Поняття предметної області та взаємозв’язки її об’єктів в MVC реалізуються в компоненті *Модель,* яка є сукупністю взаємопов’язаних класів, структур, інших типів мови C#. Операції моделі – це методи, визначені для цих типів, а правила відображіють логіку методів. На основі типів моделі створюються екземпляри (об'єкти) моделі, які віповідають даним предметної області, обробляються контролером і надсилаються в представлення.

Представлення

Представлення (*View*) використовується для формування зовнішнього вигляду сторінок і є файлами з розширенням *cshtml*, які містять опис інтерфейсу користувача мовою розмітки html та C#-подібною мовою Razor. Представлення містить код html, воно є не html-сторінкою, а програмним кодом з елементами html. В результаті компіляції файлу *cshtml* спочатку генерується клас мовою C#, а потім цей клас компілюється.

Представлення, зазвичай, відповідають методам (акціям) контролерів і записуються у відповідні їм папки в каталозі **Views**. Наприклад, файли представлень для методів контролера **Home**, будуть знаходитись в проекті в папці **Views/Home**.

За потреби можна створити в каталозі **Views** папку з довільним іменем, де будуть зберігатися файли представленнь, не обов'язково пов'язані з методами контролера.

Для рендерингу (відправки) представлення з контролера у вихідний потік використовується метод **View()**. Якщо в цей метод не передається імені представлення (не вказано параметр-представлення), то, за замовчуванням, генерується сторінка представлення з тим же іменем, що і назва методу.

Контролер

У MVC контролери є *C#* класами. Кожен відкритий (*public*) метод в класі, похідному від **Controller**, називається *методом дії (Action)*, який пов'язаний з налаштованим URL через систему маршрутизації (роутингом) ASP.NET. Іншими словами маршрутизація пов’язує метод дії з сторінкою, яка є результатом компіляції представлення.

Метод дії контролера обробляє запит користувача, виконує свої оператори, передає дані до моделі, а потім вибрає вид представлення для клієнта і генерує відповідну сторінку.

На мал. 1 показано взаємодію між контролером, моделлю і представленням. У MVC контролери знаходяться в папці під назвою **Controllers**. Коли браузер запитує стартову сторінку сайту, то отримує об’єкт представлення, згенерований

методом **Index** контролера **HomeController.** За потреби, початкову сторінку можна змінити (зміни записів у Web.config та маршрутизацію).

Мал. 1 Взаємодія між контролером, моделлю і представленням

Метод **Index** повертає об'єкт типу **ViewResult**, який створюється методом **View()** і містить контест (тобто зміст) запитаної веб-сторінки. Зовнішній вигляд сторінки формується html-розміткою файлу **Index.cshtml** з папки **Views**.

Файли представлень мають розширення *.cshtml*, вони обробляються транслятором мови представлень *Razor*. Розмітка сторінки *.cshtml* містить HTML код та елементи мови *Razor*, які розпочинаються з символу @. У них знаходиться код, який обробляється транслятором представлень *Razor* на серверній стороні.

Методу дії контролера співставляється представлення, яке відповідає угоді про імена, тобто представлення має ім'я методу дії і міститься в папці, названу іменем контролера без закінчення **Controller**.

Призначення контролера MVC в проекті – отримати, згенерувати, обробити дані і передати їх представленю, яке представляє їх у вигляді HTML розмітки.

Одним із способів передачі даних від контролера до подання є використання об'єкта **ViewBag**. **ViewBag** – це динамічний об'єкт, якому можна присвоїти довільні значення. Вони доступні для будь-якого представлення, яке може їх використовувати. Наприклад, **@ViewBag.Message** в стандартному файлі

Index.cshtml використовується для вставки в макет сторінки тексту з властивості **Message**, визначеного у файлі **HomeController.cs** .

1.2. Структура проекту MVC .NET Core

Стандартний проект містить наступні розділи і компоненти:

- **Connected Services**: підключення хмарні служби.
- **Properties**: містить файл налагоджень для запуску проекту з Visual Studio.
- **wwwroot**: вузол (на диску йому відповідає однойменна папка) призначений для зберігання статичних файлів - зображень, скриптів *javascript*, файлів *css* і т.д., які використовуються застосуванням. Мета додавання цієї папки в проект - розмежування доступу до статичних файлів, до яких дозволений доступ з боку клієнта і до яких доступ заборонений.
- **Залежності**: всі додані в проект пакети і бібліотеки.
- **Controllers**: папка для зберігання контролерів додатку.
- **Models**: папка для класів моделей.
- **Views**: каталог для зберігання представлень.
- **appsettings.json**: зберігає конфігурацію програми.
- **bower.json**: файл, який керує клієнтськими залежностями (бібліотеки *javascript* і *css*), які підключаються через менеджер пакетів *Bower.*
- **bundleconfig.json**: файл, який містить завдання по мініфікації скриптів і стилів, які виконуються при побудові проекту.
- **Program.cs**: файл, який визначає клас *Program*, що ініціалізує і запускає хост з застосуванням.
- **Startup.cs**: файл, який визначає клас *Startup*, з якого починається робота програми, тобто це точка входу в застосування.

У підкаталозі **Views\Shared** розташований файл **_Layout.cshtml**, який використовується для редагування макета сторінки застосування.

1.3. Запуск програми. Класи Program і Startup

Створення і запуск веб-хоста

Застосування *ASP.NET Core* є консольним, яке ініціалізує веб-сервер за допомогою якого потім обробляє вхідні запити.

У проекті програми міститься файл **Program.cs**, в якому визначено однойменний клас **Program**, з якого починається виконання програми:

```
……
public class Program
{
   public static void Main(string[] args)
   {
    CreateWebHostBuilder(args).Build().Run();
   }

   public static IWebHostBuilder
          CreateWebHostBuilder(string[] args) =>
           WebHost.CreateDefaultBuilder(args).UseStartup<Startup>(); // (1)
 }
}
```

У методі **Main** викликається метод **CreateWebHostBuilder**, в якому для створення хоста веб-застосування використовується клас **WebHost** з простору імен **Microsoft.AspNetCore**.

Викликається статичний метод цього класу **CreateDefaultBuilder**, що створює екземпляр класу **WebHostBuilder**.

За допомогою послідовного виклику методів **WebHostBuilder** ініціалізує веб-сервер для розгортання веб-застосування.

Спочатку викликається метод **UseStartup<Startup>()**.

Цим викликом:

встановлюється стартовий клас застосування - клас **Startup**, з якого і буде розпочинатися обробка вхідних запитів

створюється об'єкт типу **IWebHostBuilder**, який використовується для створення та конфігурування веб-хоста.

Зауваження

1. Під терміном **хост** розуміємо **web-сервер**, тобто програму, яка обробляє запити користувачів та надає результати віддаленій програмі чи застосуванню.

2. Лямда-вираз (1) є скороченим записом методу

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
 return  WebHost.CreateDefaultBuilder(args).UseStartup<Startup>();
}
```

Після повернення в метод **Main** викликається метод **Build()**, який і створює веб-хост метод **Build(),** і метод **Run(),** який запускає застосування. Після запуску веб-сервер починає обробляти всі вхідні HTTP-запити.

Конфігурування веб-хоста

Клас *Startup* повинен визначати метод ***Configure( )***. Опціонально в *Startup* можна визначити конструктор класу і метод *ConfigureServices()*.

Після запуску програми середовищем виконання спочатку виконується конструктор, потім метод *ConfigureServices()* і в наприкінці - метод ***Configure( ),*** які викликаються.

У проекті *ASP.NET Core* по шаблону *MVC* клас *Startup* виглядає так:

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

      public IConfiguration Configuration { get; }

      // This method gets called by the runtime.
      // Use this method to add services to the container.
      public void ConfigureServices(IServiceCollection services)
      {
         services.Configure<CookiePolicyOptions>(options =>
         {
            // This lambda determines whether user consent for
            //  non-essential cookies is needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
         });


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
      }

      // This method gets called by the runtime.
      // Use this method to configure the HTTP request pipeline.
      public void Configure(IApplicationBuilder app, IHostingEnvironment env)
      {
         if (env.IsDevelopment())
         {
            app.UseDeveloperExceptionPage();
```

```
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }

        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
```

1.4. Розробка представлення

Макет на основі майстер-сторінки

Майстер-сторінки є шаблонами веб-сторінок і використовуються для створення єдиного стилю оформлення всіх веб-сторінок сайту. Наприклад, можна визначити на майстер-сторінці загальні для всіх сторінок меню, а також підключити загальні стилі і скрипти. За допомогою спеціальних тегів в потрібне місце на майстер-сторінці можна вставити представлення іншої сторінки сайту, тому не потрібно буде дублювати на всіх сторінках шляхи до тих самих файлів стилів чи меню.

Під час створення проекту до нього додається майстер-сторінка під назвою **_Layout.chtml**, яка знаходиться в каталозі **Views/Shared**. Вона містить (як і всі інші майстер-сторінки) метод мови Razor **@RenderBody()**, на місце якого будуть підставлятися представлення сторінок, що її використовують. В ASP.NET Web.Forms йому відповідав тег PlaceHolder.

В представленні кожної сторінки є властивість *Layout*, яка зберігає посилання на майстер-сторінку. За замовчуванням, для задання трансляторові назви майстер-сторінки в **@RenderBody()** якої має розміщуватись представлення даної, використовують файл **_ViewStart.cshtml,** який знаходиться в папці **Views** . Код цього файлу автоматично додається в початок коду представлень конкретної сторінки після її запуску.

Файл **_ViewStart.cshtml** містить наступний код:

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

В прикладі вказано, що майстер-сторінкою, за замовчуванням буде файл **_Layout.cshtml,** розширення можна не використовувати.

Під час рендеринга сторінки (тобто її формування для відправки клієнту), система буде шукати майстер сторінку *_Layout* за наступними шляхами:

/Views/[Назва_контролера]/_Layout.cshtml

/Views/Shared/_Layout.cshtml

За потреби, можна розробити й інші майстер-сторінки і підключити їх аналогічно до попереднього в самому представлені командою:

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

Структура файлу _Layout.chtml

Файл *_Layout.chtml* має структуру звичайного *html* документу з відповідними тегами - *html, head, body*. Також файл містить ряд серверних розширень *html*, які обробляються хелперами (*helpers*) на стороні сервера. Хелпери підключаються при використанні спеціальних тегів і атрибутів тегів. У секції *head* міститься хелперний тег *environment*, який використовується для генерації розмітки в залежності від того, чи знаходиться застосування в процесі розробки, тестування або вже опубліковано на сервері. Як правило, даний тег-хелперів використовується спільно з тегами *link* і *script*.

За допомогою тегів *environment* до макету майстер-сторінки підключаються файли бібліотек *bootstra* і *jQuery*, які використовуються для адаптивної розмітки сторінок та сценаріїв обробки подій на сторінці. Ці бібліотеки будуть доступні на всіх сторінках, представлення яких використовують майстер-сторінку.

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
 </environment>

<environment exclude="Development">
  <link rel="stylesheet"
     href=https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css
```

asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
asp-fallback-test-class="sr-only" **asp-fallback-test-property**="position"
asp-fallback-test-value="absolute" />

<**link** rel="stylesheet" **href**="~/css/site.min.css" **asp-append-version**="true" />
</**environment**>

В цьому ж розділі підключаються файли стилю сторінок сайту site.css або site.min.css.

В тілі сторінки для створення меню навігації по сайту використовується тег **nav**. Основні елементи сторінки поміщені в контейнери **div**, яким призначаються класи стилів і які задають компонування сторінки.

Тег **button** задає кнопку, яка відображається при малих розмірах вікна браузера і розкриває згорнуте меню навігації.

Посилання на сторінки сайту в меню навігації реалізовані на основі хелпер-тега гіперпосилання **a** зі спеціальними атрибутами **asp-area**, **aspcontroller**, **asp-action**. Атрибут **asp-area** задає область веб-сайту. У простих проектах області зазвичай не використовуються. Атрибути **asp-controller** і **aspaction** задають контролер і його метод дії, який буде викликатися при активізації посилання.

Метод **@RenderBody()** задає місце вставки представлень, які використовують цю майстер-сторінку.

В кінці контейнера **body** знаходяться теги **environment**, які підключають скрипти, які будуть доступні в кодах майстер-сторінки і представлень.

Метод **@RenderSection ("Scripts", required: false)** задає місце вставки скриптів з цією назвою зі сторінок тих представлень , що використовують цю майстер-сторінку. Під час компонування сторінки транслятором Razor, виклик допоміжного методу **@RenderSection** замінюється вмістом розділу з представлення з іменем **Scripts**.

У наступній таблиці перераховані відмінності між методами **RenderBody()** та **RenderSection()**.

RenderBody ()	**RenderSection ()**
Метод RenderBody() є обов’язковим елементом макету майстер-сторінки.	Метод RenderSection () необов’язковий.

RenderBody ()	RenderSection ()
RenderBody () відображає той вміст дочірної сторінки, який не розміщений в іменованій секції.	RenderSection() відображає лише частину дочірнього представлення, яка знаходить в іменованій секції.
Використання декількох методів RenderBody () **НЕ** дозволяється в одному представлені макету майстер-сторінки.	Декілька методів RenderSection () дозволено в одному макеті майстер-сторінки.
Метод RenderBody () не містить параметрів.	Метод RenderSection () містить булевий параметр " required ", що робить розділ необов’язковим або обов'язковим. Якщо його значення true, то дочірне представлення повинно містити розділ з відповідним іменем.

Файли представлень стандартного макета веб-сайту

В папці **Views/Home** знаходяться файли *Index.cshtml*, *Contact.cshtml*, *About.cshtml*, що містять макети інформаційних сторінок сайту.

У файлі *Index.cshtml* для демонстрації слайд-шоу банерів використовується компонент *Carousel* бібліотеки *bootstrap*[2]. Компонент реалізується за допомогою тегу *div* та класів стилів, які описують параметри елементів каруселі.

Наступний тег *div* з класом *row* задає *bootstrap* блок, який за замовчуванням містить 12 блоків-колонок. Чотири внутрішні теги *div* з класом *col-md-3* вказують, що блок-“рядок” *row* розіб’ється на чотири блоки-колонки, кожна з яких по ширині займає місце трьох стандартних блоків-колонок. В колонках стандартного макету розміщені списки посилань на ресурси по ASP.NET.

Файли *Contact.cshtml* і *About.cshtml* містять мінімальні заготовки відповідних сторінок сайту майбутнього реального сайту.

Каскадні таблиці стилів

Для оформлення і компонування веб-сторінок використовується технологія каскадних таблиць стилів (*Cascading Style Sheets, CSS*). Таблиця стилів містить набір правил (стилів), що описують оформлення веб-сторінки та її окремих фрагментів. Ці правила визначають колір тексту, вирівнювання абзацу, відступи між графічним зображенням і оточуючий його текстом, наявність і параметри рамки у таблиці, колір фону веб-сторінки та інше [3].

Стиль підключається до елементу веб-сторінки або всієї веб-сторінки. Параметри, описані стилем, застосовуються до елементу під час відображення її браузером. Прив'язка може бути *явна* - вказується елемент сторінки, до якого підключається стиль, або *неявна* - стиль автоматично підключається до всіх елементів з певним тегом.

Таблиця стилів розміщуватись безпосередньо в HTML-коді сторінки або в окремому файлі з розширенням .**css**. Останній підхід відповідає сучасній концепції розділення вмісту і представлення сторінки. Окремі стилі також можна вказати в самому в тегові; але такий підхід використовується зараз досить рідко і є ознакою недостатньо продуманого дизайну сайту.

Сучасні стандарти CSS описані в версіях *CSS2* і *CSS 3*.

Формат визначення стилю CSS:

<селектор>
{
<Атрибут стилю 1>: <значення 1>;
 <Атрибут стилю 2>: <значення 2>;
. . .
<Атрибут стилю n-1>: <значення n-1>;
<Атрибут стилю n>: <значення n>
}

Визначення стилю включає **селектор** і **список атрибутів стилю з їх значеннями**. Селектор використовується для підключення стилю до елементу веб-сторінки, вигляд якого від задає. Атрибути стилю задають параметри елемента веб-сторінки: колір шрифта, вирівнювання тексту, величину відступу, товщину рамки та ін. Пари *<атрибут стилю>: <значення>* відокремлюють один від одного крапкою з комою. Між останньою парою *<атрибут стилю>: <значення>* і фігурною дужки символ «**;**» не ставлять. Визначення різних стилів розділяють пропусками або переходом на новий рядок.

У стандартному проекті *APN.NET MVC* основна таблиця стилів розміщується в файлі **Content\Site.css.** Таблиці стилі підключаються тегом **link** в секції **head** HTML документа:

<link rel = "stylesheet" href = "<адреса файлу таблиці стилів>" type = "text/css">

У *MVC* для оптимізації завантаження стилів і скриптів використовується два механізми: **бандлінг** і **мініфікація** .

Бандлінг (Bandling) здійснює з'єднання скриптів або стилів в один загальний файл або бандл. **Мініфікація (Minification)** зменшує розмір файлу скриптів або стилів. Для виконання операцій бандлінга і мініфікації в ASP.NET Core використовується розширення Visual Studio – **BundlerMinifier**[4].

Файл **bundleconfig.json** проекту містить декілька параметрів**.**

Параметр **outputFileName** задає шлях до вихідного файлу-результату об'єднання файлів, вказаних в параметрі **inputFiles.** Файли, які будуть об'єднуватися, розділяються комою. З файлу "**wwwroot/js/site.js**", наприклад, буде формуватися файл "**wwwroot/js/site.min.js** ". Додатковий параметр **minify** вказує, чи будуть мінізуватись файли, які включаються в бандл. Значення **enabled:true** включає мініфікацію. А значення "**renameLocals**":**true** дозволяє скоротити імена локальних змінних. Останній параметр **sourceMap** вказує, чи треба генерувати файл-карту, яка співставляє висхідний і вихідний файли.

2. ВЕБ-ФОРМИ І ВАЛІДАЦІЯ ДАНИХ В ASP.NET MVC

2.1. Веб-форми

Форми в HTML призначені для передачі даних на сервер. Форма містить елемент **<form>**, всередині якого розміщуються керуючі елементи: *кнопки, однорядкові і багаторядкові текстові поля, списки, що випадають* і т.д.

Функціональність форми визначається атрибутами елемента *<form>*. В атрибуті **name** можна вказати унікальне ім'я форми. У документі може бути декілька форм, але їх імена не повинні збігатися. За допомогою імені форми клієнтські скрипти можуть отримувати динамічний доступ до її полів ще до відправки даних на сервер. Атрибут **enctype** вказує *MIME*-типу, до якого перетворюються дані з полів форми перед її відправкою на сервер. Атрибут **action** задає адресу сервера на яку надсилаються дані. Атрибут **method** вказує метод (наприклад, GET або POST), яким дані надсилаються.

Якщо атрибути форми заповнені, то програма на сервері зможе отримати, обробити інформацію, сформувати сторінку і надіслати її назад браузеру.

2.2. Елементи форми для введення даних

За допомогою елементу форми <*input*> створюються поля для введення тексту, паролів і вибору файлів, а також кнопки, прапорці та перемикачі. У HTML 5 добавлені елементи <*input*> для введення дат, числових значень, телефонів, адрес, вибору кольору і т.д.

Призначення елемента визначається атрибутом *type* (див. табл.2). Ім'я поля задається атрибутом *name*, а його значення, за замовчуванням, вказується в *value* .

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

Таблиця 2

Значення type	Опис
Text	Значення за замовчуванням. Елемент призначений для введення текстового рядка.
Password	Елемент призначений для введення паролів. Символи, вводяться замінюються великими крапками.
Reset	Кнопка очищення форми.
Submit	Кнопка відправки даних на сервер.
Image	Альтернативний варіант кнопки відправки даних в вигляді графічного зображення, адреса якого вказується в атрибуті **src**, а альтернативний текст - в **alt** .
Hidden	Приховане поле. У браузері не відображається, але також може містити значення **name** і **value**, які відправляються на сервер.
Checkbox	Прапорець. Відображається у вигляді невеликої області з встановленої або знятої «галочкою». Якщо елемент містить атрибут **checked = "checked"**, то галочка за замовчуванням буде встановлено.
Radio	Перемикач, що відображається у вигляді кружечка з жирною крапкою (значення вибрано) або без неї (не вибрано). Вибір за замовчуванням визначається атрибутом **checked = "checked"**. На відміну від інших типів полів, у формі може бути декілька елементів **<input type = "radio">** з однаковим name, однак вибрати з них можна лише один.

File	Вибір файлу. Відображається аналогічно текстовому полю, але з доданою справа кнопкою «**Огляд**». При натиску на ній з'являється діалогове вікно вибору файлу.

У таблиці 3 вказані значення атрибута *type*, додані в стандарті HTML 5.

Таблиця 3

Search	Текстове поле, призначене для введення пошукового запиту. Деякі браузери відображають на ньому додаткову кнопку очищення поля
email	Текстове поле для введення адрес електронної пошти.
url	Текстове поле для введення абсолютного *URL*
tel	Поле для введення телефонних номерів
number	Поле числового введення. Містить кнопки-стрілки, що дозволяють збільшувати і зменшувати значення.
range	Елемент для вибору значення із заданого діапазону. Являє собою повзунок, мінімальне і максимальне значення якого задаються в атрибутах **min** і **max** відповідно, а крок - в атрибуті **step.**
time	Елемент для введення часу. Аналогічний полю для введення чисел, але з поділом на годинник і хвилини.
date	Елемент для вибору дати, що представляє собою список, що випадає григоріанський календар.
datetime -local	Комбінація двох попередніх елементів для введення дати і часу без обліку часового поясу.
datetime	Те ж, що і попередній елемент, але з встановленої тимчасової зоною UTC.
Week	Елемент для вибору тижні. Візуально аналогічний елементу з **type = "date"**, відрізняється лише формат значення.
Month	Елемент для вибору місяця. Візуально аналогічний елементу з **type = "date"**, відрізняється лише формат значення.
Color	Елемент для вибору кольору в форматі RGB.

Для введення багаторядкового тексту використовуєтьсятег <*textarea*>. Його атрибути *rows* і *cols* задають, відповідно, кількість рядків і символів в рядку. Текст задається не в атрибуті *value*, а вказується між тегами <*textarea*> та </*textarea*>. Як і в <*input*>, максимальна кількість символів в тексті задається атрибутом *maxlengthю*.

```
<textarea rows="4" cols="20" name="myText">
    Тут можна розмістити 20 рядків тексту по 20 символів в кожному
```

```
</textarea>
```

Якщо текст не поміщається в рядку області <*textarea*>, то переноситься на наступний. Якщо рядків не вистачає для відображення всього тексту, то автоматично з'являється смуга прокрутки.

В HTML 5 до тегу <*textarea*> додано атрибут *wrap*, який визначає спосіб передачі вмісту тега на сервер. При значенні *hard*, в кінець кожного рядка додається код символу переносу. За замовчування параметр *wrap* має значення *soft*, при якому символи розриву рядків не додаються.

Обидва елементи <*input*> і <*textarea*> підтримують атрибут *readonly = "readonly"* (*тільки читання*), який забороняє редагування їх вмісту на сторінці.

Кнопки можна додавати також за допомогою елемента <*button*>. Атрибут *type* в нього може набувати значень *reset, submit* і *button*, що відповідають командам очистити форму, надіслати дані та виконати якусь дію. Тег <*button*> є контейнерним і надпис на кнопці задається не в атрибуті *value*, а є змістом елемента.

Для відображення на сторінці списків з елементами, які випадають, використовується тег <*select*> з дочірними тегами <*option*> для варіантів вибору.

```
<select name="food">
   <option value="pie"> Пиріг </option>
   <option value="bread" selected="selected"> Хліб </option>
   <option value="cookie" label="Печиво"> </option>
</select>
```

Передане на сервер ім'я поля вказується в Атрибуті *name* елемента <*select*> задає ім’я керуючого елемента сторінки за яким його можна ідентифікувати на сервері, допустимі значення поля *name* містять в атрибутах *value* елементів <*option*>. У випадаючому списку, як і в випадку з перемикачем <*input type = "radio"*>, із запропонованих варіантів може бути обраний лише один. Щоб вказати варіант за замовчуванням застосовується атрибут *selected = "selected"* .

Якщо в елементі <*select*> вказати атрибут *size* з деяким числовим значенням, то список стає не випадаючим, а звичайним з вказаною кількістю видимих на екрані варіантів. Якщо їх насправді більше ніж вказане число, то браузер додасть до елементу смугу прокрутки.

За допомогою атрибута *multiple = "multiple"* можна дозволити користувачеві вибрати декілька варіантів одночасно. Такий список також перестає бути випадаючим. Щоб вказати необхідну кількість видимих елементів, необхідно застосовувати атрибут *multiple* в парі з *size* .

HTML 5 надає можливість об'єднати випадаючий список зі звичайним елементом введення <*input*>. Такий список може містити, наприклад, потрібні пошукові запити або рекомендовані значення, якими заповнюється поля. Формується він елементом <*datalist*> з вкладеними тегами <*option*> із запропонованими в атрибутах *value* варіантами. Щоб зв'язати такий список з полем введення, необхідно присвоїти елементу <*datalist*> унікальний ідентифікатор *id* і вказати його в значенні атрибута *list* елемента <*input*>. За замовчуванням <*datalist*> не відображається на сторінці, а з'являється, тільки коли користувач вводить дані в поле, до якого він прив'язаний.

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

2.3. Форми в ASP.NET MVC

Методи дії

У формах в ASP.NET MVC зручно використати атрибути тег-хелпери *aspaction* і *asp-controller*, які задають ім'я методу дії, який отримує дані з форми і клас контролера, який містить цей метод дії.

Наприклад:

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

Методам дії дані з форми передаються через вхідні параметри, які для рядків мають тип *string*, для числових значень в полях можуть використовуватись відповідні типи, наприклад, *int*, а для *checkbox - bool*. Ім'я кожного поля форми (атрибут *name*) в такому випадку має збігатися з іменем відповідного йому параметра в оголошенні методу дії.

Також методу дії може передаватися об'єкт моделі, властивості класу якої будуть містити дані з полів форми, яка надіслала дані на сервер. В цьому випадку імена полів форми теж повинні збігатися з іменами властивостей класу моделі.

Передача об'єктів класів моделі з контролера в представлення

Метод дії передає об'єкт моделі в представлення, як параметр методу *View()*, викликом якого завершується робота методу.

Директива **@model** в коді представлення вказує клас моделі. Доступ до полів переданого з контролера об'єкта в коді представлення виконується за допомогою префікса **@Model**, наприклад:

 e-mail: @Model.Mail

2.4. Валідація в MVC ASP.NET

Валідацією називається перевірка вхідних дані на наявність недопустимих значень.

Валідація за допомогою атрибутів

У *ASP.NET MVC* валідація реалізується шляхом використання відповідних атрибутів у властивостях класу моделі.

Для їх використання необхідно підключити відповідний простір імен

using System.ComponentModel.DataAnnotations;

Обов'язкове поле позначається атрибутом **Required** з простору імен.

Параметр **ErrorMessage** задає текст повідомлення про помилку.

Наприклад:

```
[Required(ErrorMessage = "Не вказано ім'я")]
public string Name { get; set; }
```

Атрибут **RegularExpression** передбачає, що значення властивості має задовільняти зазначеному регулярному виразу.

Наприклад, якщо клас моделі містить властивість *Email,* то вираз може бути таким:

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

Існує компактніший варіант з використанням атрибуту **EmailAddress**:

```
[EmailAddress(ErrorMessage = "Некоректна адреса")]
```

```
public string Email { get; set; }
```

Для валідації також можна використовувати атрибути **CreditCard**, **Phone**, **Url**. Першим параметром атрибута **StringLength** є максимально допустима довжина рядка. Параметри **MinimumLength** і **ErrorMessage**, використовуються для встановлення додаткових обмежень на значення. Атрибут **Range** визначає мінімальні та максимальні обмеження для числових даних.

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

Атрибут **Compare** гарантує, що дві властивості об'єкта моделі мають одне і те ж значення. Якщо, наприклад, потрібно, щоб користувач ввів пароль двічі:

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

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

Атрибут **Remote** з простору імен **Microsoft.AspNetCore.Mvc** для валідації властивості виконує запит на сервер до певного методу контролера. Якщо необхідний метод контролера поверне значення *false*, то валідація не пройдена .

Наприклад:

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

В коді вказано, що для перевірки значення атрибут буде звертатися до методу **CheckEmail** контролера **Home**:

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

Метод валідації має повертати об'єкт **JsonResult**, який створюється методом **Json()**. Параметр *true* означає, що валідація успішна.

Валідація в класі моделі

В згальному випадку ***валідація на стороні сервера*** виконується методом дії контролера, який отримує об'єкт класу з моделі та перевіряє значення його властивостей за допомогою об’єкту **ModelState**. Перевірка властивостей здійснюється методами класу моделі, а результат перевірки записується у властивість в **ModelState.IsValid**:

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

Якщо в об'єкті **ModelState** є якісь помилки, то властивість **ModelState.IsValid** поверне *false*.

Валідація на стороні клієнта

Валідація на стороні клієнта дозволяє зменшити кількість звернень до сервера і виконати перевірку значень безпосередньо при введенні даних. Для активізації валідації на стороні клієнта необхідно підключити до представлення скрипти валідації. Код підключення скриптів валідації для дочірнього представлення сторінки (не мастер-сторінки) має такий вигляд:

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

Цей код необхідно додати на кінець файлу представлення.

Для валідації значень полів форми потрібно вказівки ім’я поля в його атрибуті **asp-for** і в атрибуті **aspvalidation-for** пов’язаного з ним тегу span. В тегові **span** буде виводитись повідомлення про помилку для некоректних даних:

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

На початок коду форми після тега **form** додається тег для відображення загальних помилок валідації, які одержуються з сервера:

<div class="validation" **asp-validation-summary**="ModelOnly">

3. ВИКОРИСТАННЯ БАЗ ДАНИХ В ASP.NET MVC

3.1. Модель даних та контекст БД

ASP.NET MVC підтримує створення баз даних на основі класів моделі даних за допомогою технології **Entity Framework,** яка реалізує об'єктно-реляційне відображення (*ORM*) структур даних, описаних в програмному коді, в таблиці реляційних баз даних, а також зворотне відображення.

Щоб використити ORM необхідно описати класи моделі даних, а потім на їх основі згенерувати таблиці бази даних засобами *Entity Framework*. Після цього можна отримувати дані з бази за допомогою запитів до класів. Взаємодію з базою, збереження змін, буде забезпечувати платформа *Entity Framework*.

У найпростішому випадку, модель даних утворюють класи, які містять автоматичні властивості (з порожніми аксессорами *get* і *set*). Наприклад, так може виглядати модель записів про книги для інтернет-магазинів:

```
public class Book
{
   public int Id { get; set; } // ID книги
   public string Name { get; set; } // назва книги
   public string Author { get; set; } // автор книги
   public int Price { get; set; }  // ціна
}
```

Для класів використовують певні домовленості щодо інтерпретації їх полів. Властивість з іменем *Id* використовується для зовнішніх (первинних) ключів, таблиць які створюються, регістр символів не враховується.

На основі класів моделі визначають клас контексту даних, який відповідає даним БД та її схемі. Клас контексту даних для бази, що містить одну таблицю, може виглядати так:

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

Для прив'язки класу контексту даних до бази даних використовується передача об'єкта конфігурації *DbContextOptions* в конструктор базового класу *DbContext*. В об'єкті конфігурації задається джерело даних і рядок ініціалізації з'єднання з базою даних. Після цього для доступу до бази створюється екземпляр класу контексту даних і викликаються його методи.

3.2. Запис, зчитування, вибірка даних з БД

Приклад додавання запису про книгу в базу даних:

```
ShopContext db = new ShopContext();
Book book = new Book();
book.Name = "Інтернат";
book.Author = "Сергій";
book.Price = "Жадан";
db.Books.Add(book);
db.SaveChanges();
```

Доступ до записів з бази даних:

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

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

Доступ за Id:

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

Отримання впорядкованої за назвами колекції книг за допомогою запиту мовою LINQ:

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

або за допомогою лямбда-виразу та функції EF:

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

Значення полів елементів колекції query, отриманої в результаті виконання запиту, можуть передаватись в представлення (View) і виводитися на сторінках сайту.

3.3. Моделі пов'язаних таблиць бази даних

Для створення моделей пов'язаних таблиць в *Entity Framework* в класи моделі додаються особливі поля.

Для організації зв'язку **один-до-багатьох** в клас, який відповідає таблиці із зовнішнім ключем, додаються два поля - *зовнішній ключ і навігаційна властивість*. *Навігаційна властивість* - це поле, значення якого є об’ктом, який відповідає рядку таблиці, на яку вказує зовнішній ключ.

В клас, що представляє таблицю, на яку вказують зовнішні ключі, також додається навігаційна властивість - колекція посилань на неї об'єктів. Ім'я поля формується як множина від імені типу, об'єктів, які посилаються.

Наведемо приклад додавання зв'язку один-до-багатьох для класів

Client (інформація про зареєстрованого користувача) і

Comment (відгук користувача).

Жирним шрифтом виділені навігаційні властивості, які реалізують зв'язок **один-до-багатьох**.

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

  // навігаційна властивість - колекція посилань
  public List<Comment> Comments { get; set; }
}
public class Comment
{
   public int Id { get; set; }
   public DateTime CommentDate { get; set; }
   public string CommentText { get; set; }

  public int ClientId { get; set; }   //зовнішній ключ
  public Client Client { get; set; }    //навіг. властивість
}
```

Для організації зв'язку **один-до-одного** в підпорядкованому класі потрібно задавати зовнішній ключ і навігаційну властивість, аналогічно до попереднього прикладу. В основний клас додається тільки навігаційне поле, що посилається на екземпляр підлеглого класу.

Наведемо приклад додавання зв'язку **один-до-одного** для класів

Client (інформація про зареєстрованого користувача) і

UserProfile (додаткові дані користувача).

Жирним шрифтом виділені навігаційні властивості, які реалізують зв'язок **один-до-одного.**

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

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

//навіг. властивість
   public UserProfile UserProfile { get; set; }
}
public class UserProfile
{
   public int Id { get; set; }
   public string FIO { get; set; }
   public string Address { get; set; }
   public string Phone { get; set; }
   public string Mail { get; set; }
   public DateTime RegDate { get; set; }

   public int ClientId { get; set; } //зовнішній ключ
   public Client Client { get; set; } //навіг. властивість
 }
```

За допомогою *Entity Framework (EF)* можна реалізувати в застосуванні зв’язок класами, який відповідтиме зв’язку між таблицями реляційної бази даних типу **багато-до-багатьох**. Таблиці в БД зв’язуються з використанням асоційованої (проміжної) таблиці. Кожна з двох основних таблиць зв'язується з проміжною через зв’язок **один-до-багатьох.** В результаті, між основними таблицями утворюється зв'язок **багато- до-багатьох.**

За допомогою запитів до контесту БД, класів моделі та навігаційних властивостей їх об'єктів можна отримувати необхідні дані з БД. Необхідно враховувати, що дані з пов'язаних об'єктів для гарантованого доступу до них в коді повинні довантажуватися явно. Для цього можна використовувати метод **Load**, який викликається через об'єкт контексту бази даних.

Нижче показано використання методу **Load** для завантаження пов'язаного об'єкта через навігаційне властивість **Client** .

```
var query =from b in db.Comments orderby b.CommentDate descending select b;

foreach (var item in query)
   db.Entry(item).Reference(p => p.Client).Load();
```

Якщо через навігаційну властивість завантажується колекція пов'язаних об'єктів, то замість методу **Reference** викликається метод **Collection**, якому також передається делегат, який повертає навігаційну властивість.

3.4. Редагування і видалення записів засобами Entity Framework

Стандартні операції

Entity Framework дозволяє виконувати стандартні операції з даними таблиць, такі як створення, отримання, оновлення та видалення даних.

Видалення здійснюється за допомогою методу **Remove**:

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

Метод встановить статус об'єкта в **Deleted**, завдяки чому *Entity Framework* при виконанні методу **db.SaveChanges()** згенерує *SQL-вираз DELETE*.

Якщо необхідно *видалити відразу декілька об'єктів*, то можна використовувати метод **RemoveRange ():**

```
User user1 = db.Users.FirstOrDefault();
```

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

Для *редагування об'єкта* досить змінити його дані і викликати метод **db.SaveChanges()**. В результаті буде сформовано *SQL-вираз UPDATE* для даного об'єкта, яке оновить об'єкт в базі даних. Однак, якщо посилання на об'єкт, який змінився, було отримана в іншому методі, потрібно додатково викликати метод метод **Update** перед викликом **SaveChanges**:

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

Для одночасного оновлення декількох об'єктів використовують метод **UpdateRange()**:

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

Каскадне видалення

Каскадне видалення - це автоматичне видалення залежної сутності після видалення головної. За замовчуванням, для сутностей застосовується каскадне видалення, якщо наявність пов'язаної сутності обов'язкова.

Наприклад:

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

public class Player
 {
  public int Id { get; set; }
  public string Name { get; set; }
  public int TeamId { get; set; } // зовнішній ключ
  public Team Team { get; set; } // навігаційна властивість
 }
```

Властивість зовнішнього ключа має тип **int**, її значення не може бути *null* і вимагає наявності значення **id** у пов'язаного об'єкта **Team** . Тобто для об'єкта **Player** обов'язкова наявність пов'язаного об'єкта **Team**, тому при його видаленні будуть видалені всі об'єкти **Players**, що містять його **TeamId**.

Можна змінити моделі, вказавши необов'язковість наявності об'єкта Team:

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

public class Player
 {
   public int Id { get; set; }
   public string Name { get; set; }
   public int? TeamId { get; set; } // зовнішній ключ
   public Team Team { get; set; } // навігаційна властивість
 }
```

Тепер зовнішній ключ має тип *Nullable <int>,* тобто він допускає значення *null*. Коли гравець не буде належати жодній команді, ця властивість матиме значення *null*. Видалення команди, до якої належав гравець, тепер не призведе до каскадного видалення запису для гравця.

Додаткову інформацію про використання *Entity Framework Core* можна знайти в довідковій документації [5].

4. АУТЕНТИФІКАЦІЯ ТА АВТОРИЗАЦІЯ В ASP.NET MVC

Під *аутентифікація* будемо розуміти перевірку облікових даних користувача.

Використання cookie-файлів для аутентифікації користувачів

Після успішної аутентифікації користувача, в усі наступні запити додається *cookie-файл* з ідентифікатором користувача. *ASP.NET Core* має вбудовану підтримку аутентифікації на основі *Cookies*. *ASP.NET* містить компонент, який серіалізує дані користувача в зашифровані аутентифікаційні *cookies-файли* і передає їх на сторону клієнта. Після отримання запиту від клієнта, в якому *містяться* аутентифікаційні *cookies*, відбувається їх валідація, десеріалізація і ініціалізація властивості **User** об'єкта **HttpContext**. Підтримка аутентифікації на основі *Cookies* включається викликом відповідного методу в методі **ConfigureServices** класу **Startup**:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).
        AddCookie(
```

```
        o => o.LoginPath = new PathString("/Home/Login")
    );
```

Крім того, в метод **Configure** класу **Startup** необхідно додати виклик

app.UseAuthentication();

Атрибут авторизації, властивість User

У ASP.NET MVC використовується атрибут авторизації **[Authorize]**, який застосовується до контролерів і методів дій для того, щоб обмежити доступ неавторизованим користувачам.

Для перевірки статусу користувача атрибут використовує властивість **User** об'єкта **HttpContext**, яке встановлюється інфраструктурою ASP.NET.

Властивість **HttpContext.User** представляє об'єкт інтерфейсу **IPrincipal**, який визначений в просторі імен **System.Security.Principal**. Цей інтерфейс визначає метод **IsInRole()** і властивість **Identity** .

Властивість **Identity** повертає об'єкт інтерфейсу **IIdentity**, який пов'язаний з поточним запитом.

Метод **IsInRole ()** в якості параметра приймає роль і повертає **true**, якщо поточний користувач належить цій ролі.

Об'єкт **Iidentity** надає інформацію про поточного користувача через такі властивості:

• **AuthenticationType**: тип аутентифікації у вигляді рядка.

• **IsAuthenticated**: повертає *true*, якщо користувач аутентифікований.

• **Name**: повертає ім'я користувача, яке, найчастіше, є логіном, за яким користувач входить в застосування.

Як було зазначено вище, об'єкт **HttpContext.User** має тип **IPrincipal** і у цього об'єкта є властивість **Identity** типу **IIdentity**. Для обох цих інтерфейсів ASP.NET Core надає реалізації за замовчуванням - класи **ClaimPrincipal** і **ClaimsIdentity** . Тобто властивість **User** (**HttpContext.User**), фактично, є об'єктом **ClaimsPrincipal**.

Об’єкти claim

Обидва класи дозволяють працювати з об'єктами *claim*. Об'єкти *claim* містять інформацію про користувача, яку можна використовувати для авторизації. Наприклад, у користувача може бути визначений вік, місто, країна проживання,

улюблена музична група та інші ознаки. Всі ці ознаки можуть представляти окремі об'єкти *claim*. В залежності від їх значень користувачеві можна надати доступ до того чи іншого ресурсу. Таким чином, *claims* є більш загальним механізмом авторизації ніж стандартні логіни або ролі, які прив'язані до однієї ознаки користувача.

Кожен об'єкт *claim* є екземпляром класу **Claim**, в якому визначені такі властивості:

• **Issuer** – "видавець" або назва системи, яка видала цей *claim.*

• **Subject** – повертає інформацію про користувача в вигляді об'єкта **ClaimsIdentity.**

• **Type** – повертає тип об'єкта *claim.*

• **Value** – повертає значення об'єкта *claim.*

У найпростішому випадку для створення аутентифікаційних *Cookies* може використовуватися наступний об'єкт *Claim*:

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

Для створення *claim* його конструктору передається тип і значення. Тип **ClaimsIdentity.DefaultNameClaimType,** фактично, є логіном. А **userName**, в даному випадку, буде значенням, яке можна отримати через вираз **User.Identity.Name**.

Для роботи з об'єктами **Claim** в класі **ClaimsPrincipal** є такі властивості і методи:

• **Identity**: повертає об'єкт **ClaimsIdentity**, який реалізує інтерфейс **IIdentity** і являє поточного користувача;

• **FindAll (type)** / **FindAll (predicate)** – повертає всі об'єкти *claim*, які відповідають певному типу або умові;

• **FindFirst (type)** / **FindFirst (predicate)** – повертає перший об'єкт claim, який відповідає певному типу або умові;

• **HasClaim (type, value)** / **HasClaim (predicate) –** повертає значення true, якщо користувач має *claim* певного типу з певним значенням;

• **IsInRole (name)** – повертає значення **true**, якщо користувач належить ролі з назвою **name**.

За допомогою об'єкта **ClaimsIdentity**, який повертається властивістю **User.Identity**, можна керувати об'єктами *claim* у поточного користувача. Зокрема, клас **ClaimsIdentity** визначає такі властивості і методи:

• **Claims** – властивість, яка повертає набір асоційованих з користувачем об'єктів *claim* ;

• **AddClaim(claim)** – додає для користувача об'єкт *claim* ;

• **AddClaims(claims)** – додає набір об'єктів *claim* ;

• **FindAll(predicate)** – повертає всі об'єкти *claim*, які відповідають певній умові;

• **HasClaim(predicate)** – повертає значення **true**, якщо користувач має *claim*, що відповідає певному умові;

• **RemoveClaim(claim)** – видаляє об'єкт *claim* .

Для створення об'єкта **ClaimsIdentity** в його конструктор передається набір *claim*, тип аутентифікації (**ApplicationCookie**), тип для *claim*, що представляє логін і тип для *claim*, що представляє роль.

Створений об'єкт **ClaimsIdentity** передається в конструктор **ClaimsPrincipal**. Цей об'єкт **ClaimsPrincipal** буде надалі доступний через властивість **HttpContext.User** .

```
//створюємо об'єкт ClaimsIdentity
ClaimsIdentity id = new ClaimsIdentity(claims, "ApplicationCookie",
ClaimsIdentity.DefaultNameClaimType,
ClaimsIdentity.DefaultRoleClaimType); // установка аутентифікаційних
                                      // cookies
await HttpContext.SignInAsync("Cookies", new ClaimsPrincipal(id));
```

Для установки *Cookies* застосовується асинхронний метод контексту **HttpContext.SignInAsync().**

Як параметр він приймає схему аутентифікації, яка була використана при створенні конфігурації в класі **Startup**. У прикладі це рядок " **Cookies** ". В якості другого параметра передається об'єкт **ClaimsPrincipal**, який представляє користувача. Після виклику цього методу користувач є авторизованим.

Для виходу користувача (*видалення аутентифікаційних Cookies*) застосовується асинхронний метод **HttpContext.SignOutAsync()**, який також приймає як параметр схему аутентифікації.

5. ПРАКТИЧНЕ ЗАВДАННЯ

5.1. Теми завдань

1. Розробка веб-сайту "Бронювання місць в готелі" засобами ASP.NET.

2. Розробка веб-сайту "Замовлення столиків в ресторані" засобами ASP.NET.

3. Розробка веб-сайту "Замовлення піци" засобами ASP.NET.

4. Розробка веб-сайту "Замовлення послуг СТО" засобами ASP.NET.

5. Використання засобів Code First Entity Framework та ASP.NET MVC для розробки веб-застосувань (на прикладі сайту "Подарункові набори").

6. Використання засобів Data First Entity Framework та ASP.NET MVC для розробки веб-застосувань.

7. Розробка навчального інформаційного каналу для студентів університету.

8. Використання WPF та веб-сервісів для front end розробки (на прикладі застосування "Мережа “Смачна кава. Каса").

9. Використання WPF та веб-сервісів для front end розробки (на прикладі застосування "Мережа “Смачна кава. Адміністратор").

10. Використання ORM на основі Entity Framework для back end розробки (на прикладі застосування "Мережа “Смачна кава”).

11. Використання технологій Windows Forms та веб-сервісів для Front end розробки (на прикладі застосування "Торгівельна мережа. Каса").

12. Використання Windows Forms та веб-сервісів для front end розробки (на прикладі застосування "Торгівельна мережа. Склад").

13. Використання ORM для розробка back end частини клієнт-серверного застосування (на прикладі застосування "Торгівельна мережа").

14. Використання технологій WPF та веб-сервісів для front end розробки (на прикладі застосування "Туристична агенція. Менежер з продажу").

15. Використання Windows Forms та веб-сервісів для front end розробки (на прикладі застосування "Туристична агенція. Адміністратор").

16. Використання ORM для розробки Back end частини клієнт-серверного застосування (на прикладі застосування "Туристична агенція.").

17. Розробка застосування "Бронювання місць в готелі" засобами ASP.NET.

18. Розробка веб-сайту "Замовлення столиків в ресторані" засобами ASP.NET.

19. Розробка веб-сайту "Бронювання авіквитків" засобами ASP.NET.

20. Розробка кросплатформенних застосувань за технологією ASP.NET Core

(на прикладі розробки веб-сайту "Блог").

21. Розробка веб-сайту "MyInstgram" засобами ASP.NET.

22. Використання Web-сервісів в WPF-застосуваннях на прикладі застосування "Аукціон творів мистецтва".

23. Порівняння засобів технологій WPF та Windows Forms для відображення графічного та мультимедійного контенту на прикладі розробки Windows застосування навчальної системи.

24. Порівняння засобів технологій WPF та Windows Forms для відображення графічного та мультимедійного контенту на прикладі розробки WPF застосування навчальної системи.

25. Використання WPF для розробки системи тестування знань з графічним та мультимедійним контентом.

26. Використання OPM та Entity Framework на прикладі застосування "Система тестування знань".

27. Синхронізація потоків в розподілених Desktop застосуваннях на прикладі застосування "Театральна каса".

28. Дослідження можливості використання патерну Singleton в розподілених Desktop застосуваннях.

29. Реалізації CRUD операцій засобами LINQ в трирівневому застосуванні “Система відстеження завдань”.

30. Використання механізмів пізнього завантаження та відкладеного виконання запитів в розподілених застосуваннях.

31. Використання лямбда-виразів в представленнях front end частини ASP.NET MVC застосування "Система відстеження завдань".

32. Робота з мультимедійним контентом у WPF застосуванні ”Бібліотека контенту” (текст, аудіо, відео)

33. Реалізації універсальних CRUD операцій за допомогою збережених процедур та механізму Reflection.

34. Використання веб-сервісів у back end частині застосування "Планування подорожей".

35. Використання веб-сервісів у front end частині Web-застосування "Планування подорожей".

36. Використання RESTFull веб-сервісів в ASP.NET MVC (на прикладі розробки back end частини Web-застосування "Інтернет-магазин").

37. Використання RESTFull веб-сервісів в ASP.NET MVC (на прикладі розробки front end частини Web-застосування "Інтернет-магазин").

38. Використання веб-сервісів ASP.NET (на прикладі front end частини Web-застосування "Інтернет-аукціон").

39. Використання веб-сервісів ASP.NET (на прикладі back end частини Web-

застосування "Інтернет-аукціон").

40. Front end частина WPF застосування автоматизації роботи автомобільного салону. Back end частина застосування автоматизації роботи автомобільного салону засобами MVC.

41. Автоматизація служби доставки інтернет-магазину мобільних телефонів.

42. Розробка клієнт-серверного застосування "Бронювання місць в кінотеатрі" засобами ASP.NET.

43. Розробка клієнт-серверного застосування "Бронювання місць в потязі " засобами ASP.NET.

44. Розробка Back end частини клієнт-серверного застосування замовлення та доставки замовлення з японського ресторану.

45. Розробка front end частини клієнт-серверного застосування замовлення та доставки замовлення з японського ресторану.

46. Розробка клієнт-серверного застосування “Художня галерея” засобами ASP.NET.

47. Розробка клієнт-серверного застосування “Доставка десертів” засобами ASP.NET.

48. Використання JSON + jqury для реалізації асинхронних веб-застосувань

49. Розробка застосування "Служба виклику таксі " засобами ASP.NET.

50. Розробка online застосування "Замовлення канцелярських товарів”.

51. Використання JSON + jqury для реалізації асинхронних веб-застосувань.

52. Система “Абітурієнт”.

53. Використання JSON для реалізації API інтерфейсу Web-застосувань

54. Використання JSON + jqury для реалізації асинхронних веб-застосувань

5.2. Постановка задачі

Проект з технологічної практики - це зпроектоване та створене застосування з використанням однієї з технологій Windows Forms, WPF, ASP.NET Web Forms, ASP.NET Framework MVC, ASP.NET Core MVC в середовищі Visual Studio.

Студент повинен розробити застосування відповідно до своєї теми. У проекті має зберігатись структурована інформації в базі даних і відображатись на сторінках сайту - каталог товарів (або послуг) і т.д.

Для спрощення процесу інформаційного наповненням сторінок і пошуком графічного матеріалу, можна використати, як аналог, готовий шаблон, або будь-який існуючий сайту зі змінами, які відповідають завданням проекту.

5.3. Вимоги до проекту

1. Застосування повинно бути реалізовано за трирівневою клієнт-серверною архітектурою, містити адмін- та клієнтську частини.

2. Для ведення даних використовувати засоби SQL Server.

3. Для даних має використовуватись не менше трьох таблиць, одна з яких має бути асоційованою.

4. Для роботи з даними має використовуватись об’єктно-реляційне відображення Entity Framework (ORM) на основі Code First .

5. Для відображення схеми БД використовувати POCO-класи моделі даних, даних – контекст даних.

6. Для реалізації CRUD – операцій, вибірки та інших запитів використовувати LINQ або λ-вирази.

7. Засоби адміністрування користувачів повинні бути побудовані на основі ASPNET Membership чи ASPNET Identity на основі використання ролей.

8. В клієнтській частині повинні використовуватись засоби валідації введених користувачем даних;

9. Система повинна містити конфігураційні файл, які задають параметри програми (наприклад, з даними про кольори тексту, шрифти, розміри вікон, малюнків, тощо). Конфігураційний файл розміщується на сервері, і може бінарним, XML-файлом або стандартним файлом .config. Рядок з’єднання – у конфігураційному файлі.

10. У Windows WPF-застосуваннях потрібно використовувати Web-сервіси.

11. У Web-застосуваннях потрібно використовувати маршрутизацію.

Дизайн застосування

1. Верхній блок – слайдер зображень (або відео) основне і меню сайту. Пункти меню розташовані горизонтально з відступом.

2. Обов’язковими є сторінки з категоріями об’єктів, детальною характеристикою об’єкту, оберненим зв’язком і авторизацією користувачів.

3. На сторінці з даними повинна бути можливість впорядкування, фільтрації та пошуку.

4. Сторінки веб-застосування повинні займати всю ширину вікна і коректно відображатися у всіх браузерах.

5. Дизайн сторінок має бути адаптивний. При зміні розмірів вікна браузера блок контенту повинен розтягуватися на весь доступний йому простір, за винятком лівого меню (якщо є), шапки і breadcrumbs.

6. На сторінці «Контакти» має бути карта із зазначенням місця розташування об’єкту, GPS координати і дані повинні братися з бази.

7. При верстці не можна використовувати фрейми. Таблиці можна використовувати лише тоді, коли дані є табличними за своєю структурою.

Література

ino Esposito. Programming Microsoft® ASP.NET 4, 2-е изд. — Redmond, Washington 98052-6399.: Microsoft Press, 2011. — 966, ISBN: 978-0-7356-4338-3

2. Разработка Web- приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс MCAD/MCSD/Пер. с англ. — М.: Издательско-торговый дом ≪Русская Редакция≫, 2003. — 704 с.: ил.

3. Мак-Дональд, Мэтью, Фримен, Адам, Шпушта, Марио. M15 Microsoft ASP.NET 4 с примерами на C# 2010 для профессионалов 4, 4-е изд.:Пер. с англ — М: ООО "И.Д. Вильяме", 2011. — 1424 с.:ил. - ISBN: 978-5-8459-1702-7, 978-1-43-

HYPERLINK "https://www.amazon.com/Tony-Northrup

5. Палермо Джеффри и др. ASP.NET MVC 4 в действии. – М.: Manning, 2012. – 408 с. – ISBN 978-1-617-29041-1

HYPERLINK "https://www.amazon.com/Adam-Freeman

Посилання

1. https://dotnet.microsoft.com/apps/aspnet/mvc

2. Опис компоненти Carousel

https://getbootstrap.com/docs/4.0/components/carousel/

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

3. Онлайн-підручник CSS:

https://www.w3schools.com/css/default.asp

4. Сайт розробника інструментального засобу Bundler & Minifier

https://marketplace.visualstudio.com/items?itemName=MadsKristensen.BundlerMinifier

5. Опис технології Entity Framework Core російською мовою

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

Додаткові посилання

1. Erik Reitan. Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013, last updated September 8, 2014 – https://www.asp.net/web-forms/overview/getting-started/getting-started-with-aspnet-45-web-forms/introduction-and-overview

2. Rick Anderson. Getting Started with ASP.NET MVC 5, | last updated May 28, 2015 – https://www.asp.net/mvc/overview/getting-started/introduction/getting-started

3. MVC recommended tutorials and articles By Rick Anderson | last updated May 22, 2015 – https://www.asp.net/mvc/overview/getting-started/mvc-learning-sequence

4. Rick Anderson,Tom Dykstra, Shayne Boyer. Getting started with ASP.NET Core MVC and Visual Studio, | last updated 14.10.2016 – https://docs.asp.net/en/latest/tutorials/first-mvc-app/start-mvc.html

5. Изучаем ASP.NET MVC 4 - http://metanit.com/sharp/mvc/

6. ASP.NET Core Deep Dive into MVC

https://channel9.msdn.com/Events/Build/2016/B812