ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет програмування та комп’ютерних

і телекомунікаційних систем

Кафедра інженерії програмного забезпечення

**Лабораторна робота №**5

з дисципліни «Програмування в інтернет»

на тему:

«Валідація моделі. Анотації даних для відображення властивостей. Атрибути валідації. Валідація моделі в контролері. Відображення помилок валідації. Розробка власної логіки валідації»

**Виконав:**

студент 1 курсу, групи ІПЗс-20-1, варіант 12 Д.А.Ямборко

**Перевірив:**\_\_\_\_\_\_\_\_\_\_\_\_\_\_ О.М.Яшина

(підпис)

Хмельницький – 2020 р.

Мета. Отримати навики розробки власної логіки валідації моделі.

Завдання

Реалізувати валідацію даних Web - додатка розробленого в завданні до лабораторної роботи №4. Реалізувати самовалідацію даних. Розробити власний провайдер валідації.

Код Програми:

Моделі:

DTP.cs

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Lr1.Models

{

public class DTP

{

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public string Type { get; set; }

[Required]

public int Price { get; set; }

public virtual ICollection<PDR> Pdrs { get; set; }

}

public class DTPPropertyValidator : ModelValidator

{

public DTPPropertyValidator(ModelMetadata metadata, ControllerContext context)

: base(metadata, context)

{ }

public override IEnumerable<ModelValidationResult> Validate(object container)

{

DTP d = container as DTP;

if (d != null)

{

switch (Metadata.PropertyName)

{

case "Name":

if (string.IsNullOrEmpty(d.Name))

{

return new ModelValidationResult[]{

new ModelValidationResult { MemberName="Name", Message="Введіть назву ДТП"}

};

}

break;

case "Type":

if (string.IsNullOrEmpty(d.Type))

{

return new ModelValidationResult[]{

new ModelValidationResult { MemberName="Type", Message="Введіть тип"}

};

}

break;

case "Price":

if (d.Price > 2000 || d.Price < 50)

{

return new ModelValidationResult[]{

new ModelValidationResult { MemberName="Price", Message="Введіть ціну"}

};

}

break;

}

}

return Enumerable.Empty<ModelValidationResult>();

}

}

public class MyValidationProvider : ModelValidatorProvider

{

public override IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata, ControllerContext context)

{

if (metadata.ContainerType == typeof(DTP))

{

return new ModelValidator[] { new DTPPropertyValidator(metadata, context) };

}

if (metadata.ModelType == typeof(DTP))

{

return new ModelValidator[] { new DTPValidator(metadata, context) };

}

return Enumerable.Empty<ModelValidator>();

}

}

public class DTPValidator : ModelValidator

{

public DTPValidator(ModelMetadata metadata, ControllerContext context)

: base(metadata, context)

{ }

public override IEnumerable<ModelValidationResult> Validate(object container)

{

DTP d = (DTP)Metadata.Model;

List<ModelValidationResult> errors = new List<ModelValidationResult>();

if (d.Name == "Наїзд на нерухомий транспортний засіб" && d.Type == "Аварія" && d.Price == 1900)

{

errors.Add(new ModelValidationResult { MemberName = "", Message = "Створіть інше ДТП!" });

}

return errors;

}

}

}

PDR.cs

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web;

namespace Lr1.Models

{

public class PDR

{

public int Id { get; set; }

[Required]

[Display(Name = "Назва правила")]

public string Name { get; set; }

public virtual ICollection<DTP> Dtps { get; set; }

}

}

Global.asax:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Optimization;

using System.Web.Routing;

using Lr1.Models;

namespace Lr1

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

ModelValidatorProviders.Providers.Add(new MyValidationProvider());

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

}

Контролери:

PDRController.cs

using Lr1.Models;

using System;

using System.Collections.Generic;

using System.Data.Entity;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Lr1.Controllers

{

public class PDRController : Controller

{

ApplicationDbContext db = new ApplicationDbContext();

public ActionResult Index()

{

IEnumerable<PDR> pdrs = db.Pdrs;

ViewBag.Pdrs = pdrs;

return View();

}

public ActionResult Add()

{

return View();

}

[HttpPost]

public ActionResult Add(PDR pdr)

{

if (!ModelState.IsValid)

{

Response.Write("<script>window.alert('Заповніть всі поля!');</script>");

return View();

}

else

{

db.Pdrs.Add(pdr);

db.SaveChanges();

return RedirectToAction("../home/");

}

}

[HttpGet]

public ActionResult Edit(int? id)

{

if (id == null)

{

return HttpNotFound();

}

PDR pdr = db.Pdrs.Find(id);

if (pdr != null)

{

return View(pdr);

}

return HttpNotFound();

}

[HttpPost]

public ActionResult Edit(PDR pdr, int[] selectedShtrafs)

{

PDR newPdr = db.Pdrs.Find(pdr.Id);

newPdr.Name = pdr.Name;

db.Entry(newPdr).State = EntityState.Modified;

db.SaveChanges();

return RedirectToAction("Index");

}

[HttpGet]

public ActionResult Delete(int id)

{

PDR b = db.Pdrs.Find(id);

if (b == null)

{

return HttpNotFound();

}

return View(b);

}

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id)

{

PDR s = db.Pdrs.Find(id);

if (s == null)

{

return HttpNotFound();

}

db.Pdrs.Remove(s);

db.SaveChanges();

return RedirectToAction("Index");

}

}

}

Представлення:

PDR/Add.cshtml

@model Lr1.Models.PDR

@{ ViewBag.Title = "Додавання правила дорожнього руху";

Layout = "~/Views/Shared/\_Layout.cshtml"; }

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

</head>

<body>

<div id="content">

@Html.ValidationSummary(false)

<Form method="post" action="">

<Table>

<Tr>

<td> <p> Введіть назву: </p> </td>

<Td>

@Html.EditorFor(model => model.Name)

</Td>

</Tr>

<Tr>

<td> <input type="submit" value="Додати" onclick="add()" /> </td>

</Tr>

</Table>

</Form>

</div>

</body>

</html>

Home/Add.cshtml

@model Lr1.Models.DTP

@{ ViewBag.Title = "Додавання ДТП";

Layout = "~/Views/Shared/\_Layout.cshtml"; }

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Додавання ДТП</title>

</head>

<body>

<div id="content">

@Html.ValidationSummary(false)

<Form method="post" action="">

<Table>

<Tr>

<td> <p> Введіть назву: </p> </td>

<Td>

@Html.EditorFor(model => model.Name)

</Td>

</Tr>

<Tr>

<td> <p> Введіть тип: </p> </td>

<td>

@Html.EditorFor(model => model.Type)

</td>

</Tr>

<Tr>

<td> <p> Введіть ціну: </p> </td>

<td>

@Html.EditorFor(model => model.Price)

</td>

</Tr>

<Tr>

<td> <input type="submit" value="Додати" onclick="add()" /> </td>

</Tr>

</Table>

</Form>

</div>

</body>

</html>

Вигляд програми:

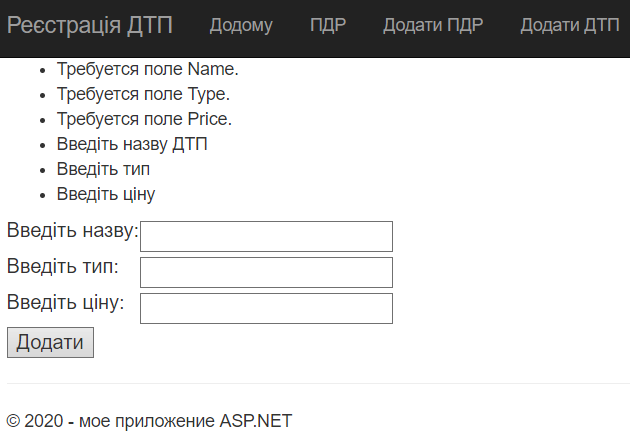


Рис 1.- Додавання ДТП

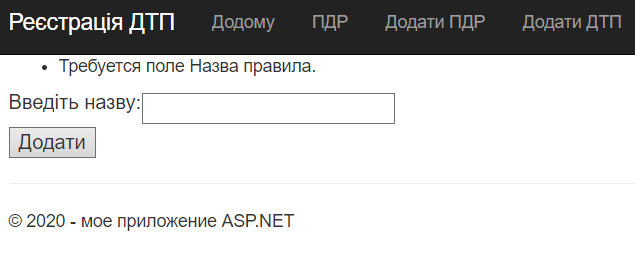


Рис 2.- Додавання станції

Контрольні питання:

1. Метадані в MVC

При роботі з моделлю в MVC велике значення відіграють метадані. Вони дозволяють вказати деяку додаткову інформацію про об'єкт, наприклад, про те, як відображати його властивості в представленні, або про те, як здійснювати валідацію введення.

1. Анотації даних

Анотації даних для відображення властивостей. Анотації даних представляють собою атрибути, які знаходятся в просторі імен.

1. Атрибут Display

Як видно, заголовки у властивостей точно такі ж, як і назви властивостей, що не дуже зручно. Адже ми могли назвати властивості як завгодно. До того ж ми не хочемо, щоб назви на українськомовному сайті були англійською мовою. Звичайно, в цьому випадку ми могли б використовувати @Html.LabelFor для налаштування назви для кожної окремого властивості, але в цьому випадку у нас би збільшився в кілька разів обсяг коду, і ми не змогли б одним рядком вивести всю інформацію про модель. До того ж може знадобитися в декількох представленнях виводити інформацію про модель. Тому даний підхід не оптимальний. І тут нам на допомогу приходять анотації даних, а саме атрибут Display. Нам достатньо поправити модель, вказавши у неї параметри відображення:

1. Атрибут HiddenInput

Атрибут HiddenInput. У попередньому прикладі у нас залишилася одна проблемка - це поле Id. Іноді, звичайно, може знадобитися виведення поля Id. Але, наприклад, якщо б ми виводили модель в режимі редагування за допомогою хелпера @Html.EditorForModel(), то дане поле було б доступно для редагування, що не дуже добре, особливо коли ідентифікатори не повинні змінюватися. Щоб приховати це поле ми можемо застосувати атрибут HiddenInput:

1. Атрибут ScaffoldColumn

Атрибут ScaffoldColumn. При редагуванні моделі атрибут HiddenInput повністю не приховує поля, так як ми можемо подивитися вихідний код сторінки і знайти відповідні поля. Щоб повністю приховати властивість від хелперів, використовується атрибут ScaffoldColumn:

[ScaffoldColumn(false)]

public virtual int Id { get; set; }

Тепер хелпери редагування не побачать дану властивість і не створять для нього навіть приховане поле на сторінці.

1. Атрибут DataType

Атрибут DataType. Атрибут DataType дозволяє надавати середовищі виконання інформацію про використання властивості. Наприклад, припустимо, у нас є властивість Password:

[DataType (DataType.Password)]

public string Password {get; set; }

Для властивості з атрибутом DataType.Password HTML-хелпери створюють елемент введення, у якого атрибут type має значення "password". Тоді в браузері при введенні даних не побачите символи, що вводяться. Перерахування DataType може приймати кілька різних значень

1. Атрибут UIHint

Атрибут UIHint. Даний атрибут вказує, який буде використовуватися шаблон відображення при створенні розмітки html для даної властивості. Шаблон управляє, як властивість буде рендерится на сторінці. Є наступні вбудовані шаблони:

- Boolean - хелпери редагування створюють прапорець (checkbox) для булевих значень. Для значень типу bool? (nullable) створюється елемент select з параметрами True, False і Not Set. Хелпери відображення генерують ті ж елементи html, що і хелпери редагування, тільки з атрибутом disabled;

- Collection - Використовується відповідний шаблон для рендеринга кожного елемента колекції. Причому елементи можуть

бути різних типів;

- Decimal - Хелпери редагування створюють однорядкове текстове поле - елемент input;

- EmailAddress - Хелпери редагування створюють однорядкове текстове поле. Хелпери відображення генерують елемент посилання, де атрибут href має значення mailto: url;

- HiddenInput - Створюється приховане поле - елемент hidden input;

- Html - Хелпери редагування створюють однорядкове текстове поле. Хелпери відображення просто показують текст;

- MultilineText -Хелпери редагування створюють багаторядкове текстове поле (елемент textarea);

- Object - Хелпери вивчають властивості об'єкта і вибирають найбільш відповідні для нього шаблони;

- Password - Хелпери редагування створюють текстове поле для введення символів з використанням маски. Хелпери відображення показують пароль, без використання маски;

- String - Хелпери редагування створюють однорядкове текстове поле;

- Url - Хелпери редагування створюють текстове поле. Хелпери відображення створюють елемент посилання для даного Url;

1. Атрибут Required

Атрибут Required. Застосування цього атрибуту до властивості моделі означає, що дана властивість має бути обов'язково встановлена.

Щоб при валідації ми не отримували безглуздих повідомлень про помилку, цей атрибут дозволяє налаштувати текст повідомлення

1. Типізований контролер

Контролер одержує вхідні дані й перетворює їх на команди для моделі чи вигляду.

1. Використання атрибутів валідації

За допомогою атрибутів валідації моделі ми можемо управляти валідацією і укладати нескладну логіку перевірки значень властивостей вже в атрибути цих властивостей, не вдаючись до коду. Розглянемо атрибути валідації, які ми можемо застосувати в додатку на ASP.NET Core.

1. Використання хелперів валідації

Для валідації на стороні клієнта застосовується клас ValidationMessageTagHelper. Даний tag-хелперів використовується за допомогою застосування до елементу <span> атрибута asp-validation-for:

<Label asp-for = "Name"> </ label>

<Input type = "text" asp-for = "Name" />

<Span asp-validation-for = "Name" />

Атрибут asp-validation-for в якості значення приймає назву властивості моделі, для якого буде виводитися повідомлення про помилку валідації.

Даний елемент span буде генерувати наступну розмітку:

<Span class = "field-validation-valid" data-valmsg-for = "Name" data-valmsg-replace = "true" />

1. Валідація на стороні сервера

В фреймворку передбачена також валідація на стороні сервера:

[HttpPost]

public ActionResult Create(Book book)

{

if (ModelState.IsValid)

{

db.Books.Add(book);

db.SaveChanges();

return RedirectToAction("Index");

}

return View(book);

}

return View (book);

}

1. Властивість ModelState.IsValid

За допомогою властивості ModelState.IsValid ми дізнаємося, проходить модель валідацію чи ні, і залежно від результату здійснюємо ті чи інші дії. Для демонстрації вибрано додаток за шаблоном Basic, однак якщо ви працюєте з проектом програми по шаблону Empty, то вам доведеться додавати всі ці файли jquery, код css та інше, щоб створити механізм валідації. Такий у загальному механізм валідації програми за замовчуванням. Тепер більш детально розглянемо основні моменти валідації.

1. Атрибут StringLength

Атрибут StringLength. Щоб користувач не міг ввести дуже довгий текст, використовується атрибут StringLength. Особливо це актуально, якщо в базі даних встановлено обмеження на розмір рядків. Першим параметром йде максимальна допустима довжина рядка. Іменовані параметри, зокрема MinimumLength і ErrorMessage, дозволяють поставити додаткові опції відображення

1. Атрибут RegularExpression

Атрибут RegularExpression. Застосування даного атрибута допускає, що значення яке вводиться має відповідати зазначеному в цьому атрибуті регулярному виразу. Найбільш поширений приклад - це перевірка коректності адреси електронної пошти.

1. Атрибут Range

Атрибут Range. Атрибут Range визначає мінімальні та максимальні обмеження для числових даних.

[Display(Name = "Рік")]

[Range (1700,2000, ErrorMessage = "Недопустимий рік")]

Атрибут Range може працювати як з цілочисельними значеннями, так і з числами з плаваючою крапкою:

[Range(typeof(decimal), "0.00", "49.99")]

public decimal Price { get; set; }

1. Атрибут Remote

бутів знаходиться в просторі імен System.Web.Mvc. Він дозволяє 105 виконувати валідацію на стороні клієнта із зворотними викликами на сервер. Наприклад, два користувача не можуть одночасно мати одне і теж значення UserName. Але за допомогою валідації на стороні клієнта важко гарантувати, що введене значення буде унікальним. А за допомогою атрибуту Remote ми можемо послати значення властивості UserName на сервер, а там воно вже порівнюється зі значеннями, що знаходяться в базі даних.

1. Валідація моделі в контролері

Крім валідації на стороні клієнта, ми можемо здійснювати валідацію і всередині контролера. Робиться це 106 за допомогою перевірки значення властивості ModelState.IsValid. Об'єкт ModelState зберігає всі значення, які користувач ввів для властивостей моделі, а також всі помилки, пов'язані з кожною властивістю і з моделлю в цілому. Якщо в об'єкті ModelState є якінебудь помилки, то властивість ModelState.IsValid поверне False

1. Перевірка значень окремих властивостей моделі

Ми також можемо перевіряти коректність значень окремих властивостей моделі:

[HttpPost]

public ActionResult Create(Book book) {

if (string.IsNullOrEmpty(book.Name)) {

ModelState.AddModelError("Name", "Некоректна назва книги");

}

else if (book.Name.Length > 5) {

ModelState.AddModelError("Name", "Недопустима довжина рядка");

}

if (ModelState.IsValid) {

107 ViewBag.Message = "Валідація пройдена";

db.Books.Add(book);

db.SaveChanges();

return RedirectToAction("Index");

}

ViewBag.Message = "Запит не пройшов валідацію";

return View(book);

}

1. Метод ModelState.AddModelError

Метод ModelState.AddModelError додає для властивості, зазначеної в якості першого параметра

1. Хелпер Html.ValidationSummary

Для відображення повідомлень про помилки при валідації моделі ми можемо використовувати хелпер Html.ValidationSummary:

@using (Html.BeginForm()) {

108 @Html.ValidationSummary(false)

1. Хелпер Html.ValidationMessageFor

Ми можемо використовувати хелпер Html.ValidationMessageFor для виведення повідомлення про помилку для окремої властивості, причому поряд з полем для введення властивості.

1. Базовий клас ValidationAttribute

Усі атрибути валідації утворені від базового класу ValidationAttribute, який знаходиться в просторі імен System.ComponentModel. DataAnnotations. Тому саме від цього класу ми будемо утворювати свій атрибут. Припустимо, нам треба, щоб яка-небудь книга була написана обмеженим колом авторів. Створимо відповідний атрибут, який буде це перевіряти:

public class MyAuthorsAttribute : ValidationAttribute {

//масив для зберігання допустимих авторів

private static string[] myAuthors;

public MyAuthorsAttribute(string[] Authors) {

myAuthors = Authors;

}

public override bool IsValid(object value) {

if (value != null) {

string strval = value.ToString();

for (int i = 0; i < myAuthors.Length; i++) {

if (strval == myAuthors[i]) return true;

}

}

return false;

}

}

1. Атрибути валідації на рівні моделі

Атрибути валідації на рівні моделі застосовуються до перевірки комбінації властивостей.

1. Самовалідація і IValidatableObject

Самовалідація являє собою процес, при якому модель запускає механізм валідації із себе самої. І сама інкапсулює всю логіку валідації. Для цього клас моделі повинен реалізувати інтерфейс IValidatableObject.

1. Власний провайдер валідації

Для створення провайдера валідації нам потрібно створити клас, наслідуваний від класу ModelValidatorProvider і реалізувати його метод GetValidators.