

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет програмування та комп'ютерних
і телекомунікаційних систем

Кафедра інженерії програмного забезпечення

Лабораторна робота № 5

з дисципліни «Програмування Інтернет»
на тему:

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

Виконав:

студент 3 курсу, групи ПЗ-18-1

(підпис)

В.В.Охота
(Ініціали, прізвище)

Перевірив:

(підпис)

О.М. Яшина

Мета. Отримати навички 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.

Хід роботи

1. Редагуємо модель Car даним чином:

```
public class Car
{
    [ScaffoldColumn(false)]
    public virtual int Id { get; set; }
    [Display(Name = "Марка та модель")]

    public virtual string Name { get; set; }
    [Display(Name = "Рік виробництва")]

    public virtual int Age { get; set; }
    [Display(Name = "Ціна")]

    public virtual string Prise { get; set; }

    public virtual int? SalonId { get; set; }

    public Salon Salon { get; set; }
}
```

2. Тепер створимо контролер, який буде управляти об'єктами даної моделі. Зробимо контролер CarController типізований: як шаблон виберемо MVC controller with read/write actions and views, using Entity Framework, а в якості класу моделі вкажемо нашу модель

У результаті у нас буде за замовчуванням створений набір представлень з управління об'єктами моделі. Запустимо додаток і звернемося до дії Create нашого контролера (запит Car / Create), яка повинно додавати новий запис у БД. Не заповнюючи поля, ми можемо відразу натиснути на кнопку відправки форми

Код контролера:

```
namespace CarShop.Controllers
{
    public class CarController : Controller
    {
        private CarContext db = new CarContext();

        // GET: Car
        public ActionResult Index()
        {
            var cars = db.Cars.Include(c => c.Salon);
            return View(cars.ToList());
        }

        // GET: Car/Details/5
        public ActionResult Details(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Car car = db.Cars.Find(id);
```

```

        if (car == null)
        {
            return HttpNotFound();
        }
        return View(car);
    }

    // GET: Car/Create
    public ActionResult Create()
    {
        ViewBag.SalonId = new SelectList(db.Salons, "Id", "Name");
        return View();
    }

    // POST: Car/Create
    // Чтобы защититься от атак чрезмерной передачи данных, включите
    // определенные свойства, для которых следует установить привязку.
    // Дополнительные
    // сведения см. в разделе https://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Create([Bind(Include = "Id,Name,Age,Prise,SalonId")]
    Car car)
    {
        if (ModelState.IsValid)
        {
            db.Cars.Add(car);
            db.SaveChanges();
            return RedirectToAction("Index");
        }

        ViewBag.SalonId = new SelectList(db.Salons, "Id", "Name",
        car.SalonId);
        return View(car);
    }

    // GET: Car/Edit/5
    public ActionResult Edit(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Car car = db.Cars.Find(id);
        if (car == null)
        {
            return HttpNotFound();
        }
        ViewBag.SalonId = new SelectList(db.Salons, "Id", "Name",
        car.SalonId);
        return View(car);
    }

    // POST: Car/Edit/5
    // Чтобы защититься от атак чрезмерной передачи данных, включите
    // определенные свойства, для которых следует установить привязку.
    // Дополнительные
    // сведения см. в разделе https://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Edit([Bind(Include = "Id,Name,Age,Prise,SalonId")]
    Car car)
    {
        if (ModelState.IsValid)
        {
            db.Entry(car).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }

```

```

    }
    ViewBag.SalonId = new SelectList(db.Salons, "Id", "Name",
car.SalonId);
    return View(car);
}

// GET: Car/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Car car = db.Cars.Find(id);
    if (car == null)
    {
        return HttpNotFound();
    }
    return View(car);
}

// POST: Car/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Car car = db.Cars.Find(id);
    db.Cars.Remove(car);
    db.SaveChanges();
    return RedirectToAction("Index");
}

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
}
}

```

3. Створюємо власний провайдер валідації

```

public class MyValidationProvider : ModelValidatorProvider
{
    public override IEnumerable<ModelValidator>
GetValidators(ModelMetadata metadata, ControllerContext context)
    {
        if (metadata.ContainerType == typeof(CarShop.Models.Car))
        {
            return new ModelValidator[] { new
CarPropertyValidator(metadata, context) };
        }

        if (metadata.ModelType == typeof(CarShop.Models.Car))
        {
            return new ModelValidator[] { new CarValidator(metadata, context) };
        }
        return Enumerable.Empty<ModelValidator>();
    }
}

```

За допомогою переданого як параметр об'єкта ModelMetadata ми отримуємо деяку інформацію стосовно об'єктів валідації. Так, ми можемо отримати доступ до наступних властивостей даного об'єкта: 1. ContainerType. Ця властивість повертає тип

перевіряємої моделі, яка містить вказану властивість. 2. `PropertyName`. Ця властивість повертає ім'я перевіряємої властивості. 3. `ModelType`. Ця властивість повертає тип об'єкту моделі

4. Інкапсулюємо логіку валідації для окремих властивостей

```
public class CarPropertyValidator : ModelValidator
{
    public CarPropertyValidator(ModelMetadata metadata, ControllerContext
context): base(metadata, context){ }
    public override IEnumerable<ModelValidationResult> Validate(object
container)
    {
        CarShop.Models.Car b = container as CarShop.Models.Car;
        if (b != null)
        {
        }
        return Enumerable.Empty<ModelValidationResult>();
    }
}
```

У методі `Validate` ми визначаємо перевіряємі властивості і викликаємо відповідні дії з валідації об'єкта. Потім в об'єкт `ModelValidationResult` додаємо відомості щодо виниклої помилки: властивість `MemberName` вказує на ім'я перевіряємої властивості, а властивість `Message` - на повідомлення про помилку валідації. Схожим чином виглядає валідатор для всієї моделі - `CarValidator`, тільки в об'єкт `ModelValidationResult` як значення властивості `MemberName` ми передаємо порожні лапки:

5. Клас `CarValidator`

```
public class CarValidator : ModelValidator
{
    public CarValidator(ModelMetadata metadata, ControllerContext context) :
base(metadata, context)
    { }
    public override IEnumerable<ModelValidationResult> Validate(object
container)
    {
        CarShop.Models.Car b = (CarShop.Models.Car)Metadata.Model;
        List<ModelValidationResult> errors = new
List<ModelValidationResult>();
        if (string.IsNullOrEmpty(b.Name))
        {
            errors.Add(new ModelValidationResult
            {
                MemberName = "",
                Message = "Неприпустима марка та модель"
            });
        }

        if (b.Age > 2020 || b.Age < 1950)
        {
            errors.Add(new ModelValidationResult
            {
                MemberName = "",
                Message = "Неприпустимий рік виробництва"
            });
        }

        if (string.IsNullOrEmpty(b.Prise))
        {
            errors.Add(new ModelValidationResult
```

```

        {
            MemberName = "",
            Message = "Неприпустима ціна"
        });
    }

    if (b.Name == "Kamaz")
    {
        errors.Add(new ModelValidationResult
        {
            MemberName = "",
            Message = "Неприпустима марка"
        });
    }
    return errors;
}
}
}

```

Скріншоти виконання програмного продукту

Новий автомобіль

• Неприпустима марка та модель

| | |
|-----------------|--------------------------------------|
| Марка та модель | <input type="text"/> |
| Рік виробництва | <input type="text" value="2015"/> |
| Ціна | <input type="text" value="231213"/> |
| SalonId | <input type="text" value="AutoMax"/> |

[Повернутися на головну сторінку](#)

© 2020 - Lab 5 - Охота Вадим

Рисунок 1 Помилка про недопустиму марку та модель

Новий автомобіль

• Неприпустима марка

| | |
|-----------------|--------------------------------------|
| Марка та модель | <input type="text" value="Kamaz"/> |
| Рік виробництва | <input type="text" value="2015"/> |
| Ціна | <input type="text" value="231213"/> |
| SalonId | <input type="text" value="AutoMax"/> |

[Повернутися на головну сторінку](#)

© 2020 - Lab 5 - Охота Вадим

Рисунок 2 Помилка про недоступну марку

Новий автомобіль

• Неприпустимий рік виробництва

Марка та модель

Lada 110

Рік виробництва

20152

Ціна

3000

SalonId

AutoMax

Зберегти

[Повернутися на головну сторінку](#)

© 2020 - Lab 5 - Охота Вадим

Рисунок 3 Помилка дати виробництва

Новий автомобіль

• Неприпустима ціна

Марка та модель

Lada 110

Рік виробництва

2015

Ціна

SalonId

AutoMax

Зберегти

[Повернутися на головну сторінку](#)

© 2020 - Lab 5 - Охота Вадим

Рисунок 4 Помилка вартості

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.

2. Анотації даних

Анотації даних для відображення властивостей. Анотації даних представляють собою атрибути, які знаходяться в просторі імен `System.ComponentModel.DataAnnotations`

3. Атрибут Display

Атрибут `Display` містить рядок, який буде відображатися замість імені властивості.

4. 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`

5. Атрибут ScaffoldColumn

Атрибут ScaffoldColumn. При редагуванні моделі атрибут HiddenInput повністю не приховує поля, так як ми можемо 97 подивитися вихідний код сторінки і знайти відповідні поля. Щоб повністю приховати властивість від хелперів, використовується атрибут ScaffoldColumn

6. Атрибут DataType

Атрибут DataType дозволяє надавати середовищі виконання інформацію про використання властивості.

7. Атрибут UIHint

Атрибут UIHint. Даний атрибут вказує, який буде використовуватися шаблон відображення при створенні розмітки html для даної властивості. Шаблон управляє, як властивість буде рендеритися на сторінці.

8. Атрибут Required

Застосування цього атрибуту до властивості моделі означає, що дана властивість має бути обов'язково встановлена. Щоб при валідації ми не отримували безглузких повідомлень про помилку, цей атрибут дозволяє налаштувати текст повідомлення

9. Типізований контролер

Як шаблон виберемо MVC controller with read/write actions and views, using Entity Framework, а в якості класу моделі вкажемо нашу модель. У результаті у нас буде за замовчуванням створений набір представлень з управління об'єктами моделі.

10. Використання атрибутів валідації

Використання атрибутів валідації при оголошенні моделі. Ми вказали для кожної властивості атрибут Required, завдяки чому фреймворк знає, що дана властивість обов'язково повинно містити деяке значення.

11. Використання хелперів валідації

Використання хелперів валідації. При кожній властивості ми використовуємо хелпер валідації Html.ValidationMessageFor:

```
@Html.LabelFor(model => model.Name)
```

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

Завдяки чому і відображається повідомлення про помилку. Щоб налаштувати стиль відображення на стороні

12. Валідація на стороні сервера

```
[HttpPost]
```

```
public ActionResult
```

```
Create(Car car) {
```

```
if (ModelState.IsValid)
```

```
{ db.Books.Add(book); db.SaveChanges(); return RedirectToAction("Index"); }
```

```
return View(car);
```

```
}
```

```
return View (car);
```

```
}
```

13. 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

14. 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.

15. 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.

16. Атрибут Range

Атрибут Range визначає мінімальні та максимальні обмеження для числових даних.

17. Атрибут Remote

Атрибут Remote на відміну від попередніх атрибутів знаходиться в просторі імен System.Web.Mvc. Він дозволяє 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

18. Валідація моделі в контролері

Крім валідації на стороні клієнта, ми можемо здійснювати валідацію і всередині контролера. Робиться це 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

19. Перевірка значень окремих властивостей моделі

Метод GetValidators викликається для кожної властивості і окремо для всієї моделі. Тому ми використовуємо два валідатора - BookPropertyValidator (для властивостей моделі) і BookValidator (для моделі в цілому).

20. Метод ModelState.AddModelError

ModelState.AddModelError додає для властивості, зазначеної в якості першого параметра (в даному випадку Name) помилку "Недопустима довжина рядка". При використанні хелперів ми можемо вивести дане повідомлення про помилку

21. Хелпер Html.ValidationSummary

Відображає загальний список помилок зверху

22. Хелпер Html.ValidationMessageFor

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

23. 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.

24. Атрибути валідації на рівні моделі

Атрибути валідації на рівні моделі застосовуються до перевірки комбінації властивостей.

25. 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`

26. Власний провайдер валідації

Для його створення ми повинні створити клас похідний від класу `ModelValidatorProvider` і перевизначити його метод `GetValidators`.