Модель

Проектирование и разработка веб-сервисов

Вводные понятия

- Понятие контроллера
- Атрибуты контроллера и действий
- Передача данных в контроллер
- Результаты действий
- Переопределение контроллеров
- Контекст контроллера

Понятие модели

Одним из ключевых компонентов паттерна MVC являются **модели**. Ключевая задача моделей - описание структуры и логики используемых данных.

Как правило, все используемые сущности в приложении выделяются в отдельные модели, которые и описывают структуру каждой сущности. В зависимости от задач и предметной области мы можем выделить различное количество моделей в приложении.

Понятие модели

Все модели оформляются как обычные классы на языке C#.

Например, если мы работаем с приложением интернет-магазина мобильных телефонов, то мы могли бы определить в проекте следующую модель, представляющую телефон:

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public int Price { get; set; }
}
```

Понятие модели

Модель Phone определяет ряд свойств: уникальный идентификатор Id, название, компанию производителя и цену.

Это классическая **анемичная модель**. Анемичная модель не имеет поведения и хранит только состояние в виде свойств.

Однако модель необязательно должна состоять только из свойств. Кроме того, она может иметь конструктор, какие-нибудь методы, поля, т.е. представлять стандартный класс.

Модели, которые также определяют поведение, называют **"толстыми" моделями** (Rich Domain Model / Fat Model / Thick Model).

Какой бы способ описания сущности не был выбран, необходимо помнить:

- предназначение модели – описывать данные;
- модель должна описывать только одну сущность.

Виды моделей

Модели можно разделить по степени применения на несколько групп:

1. Модели, объекты которых хранятся в специальных хранилищах данных (например, в базах данных, файлах xml и т.д.).
2. Модели, которые используются для передачи данных представление или наоборот, для получения данных из представления. Такие модели еще называтся **моделями представления.**
3. Вспомогательные модели для промежуточных вычислений.

Для хранения моделей создается в проекте отдельная папка **Models**. Модели представления нередко помещаются в отдельную папку, которая называется **ViewModels**.

Пример работы с моделью представления

Модель представления используется, когда в представление необходимо передать сразу несколько моделей.

Объект модели передается в качестве параметра методу View().

В представлении работа с моделью осуществляется:

```
@using ModelsApp.ViewModels
@using ModelsApp.Models

@model IndexViewModel

<table class="table">
    @foreach (Phone p in Model.Phones)
    {
        <tr><td>@p.Name</td><td>@p.Manufacturer?.Name</td></tr>
    }
</table>
```

Привязка модели

Привязка модели или **Model binding** представляет механизм сопоставления значений из HTTP-запроса с параметрами метода контроллера.

При этом параметры могут представлять как простые типы (int, float и т. д.), так и более сложные типы данных, например, объекты классов.

Для поиска значений привязчик модели используется следующие источники в порядке приоритета:

1. Данные форм. Хранятся в объекте Request.Form
2. Данные маршрута, то есть те данные, которые формируются в процессе сопоставления строки запроса маршруту. Хранятся в объекте RouteData.Values
3. Данные строки запроса. Хранятся в объекте Request.Query

Привязка модели

В случае, если параметры метода представляют сложные данные, например, класс, привязчик модели будет действовать подобным образом. Он использует рефлексию и рекурсию для прохода по всем свойствам параметра сложного типа для сопоставления свойств со значениями из запроса.

В частности, привязки модели ищет значения с ключами наподобие [имя_параметра].[имя_свойства]. Если подобных значений не будет найдено, то привязчик ищет значения просто по имени свойства.

Для таких типов как коллекции привязчик модели ищет значения с ключами имя_параметра[index]. Если параметр представляет объект Dictionary, то привязчик модели также ищет в источниках запроса значения с ключами имя_параметра[ключ].

Привязка модели

При этом свойства, к которым осуществляется привязка, должны быть объявлены с модификатором public и быть доступными для записи. А сам класс должен иметь общедоступный конструктор по умолчанию.

И когда будет осуществляться механизм привязки для создания объекта будет использоваться этот стандартный конструктор, и затем у созданного объекта будут устанавливаться свойства.

Вполне возможна ситуация, когда привязчик не найдет требуемое значение. В этом случае перед использованием параметра метода желательно проверять свойство **ModelState.IsValid**.

Управление привязкой

Для тех свойств модели, для которых не переданы значения, устанавливаются значения по умолчанию, например, для строковых свойств - пустые строки, для числовых свойств - число 0.

Поэтому фреймворк MVC предоставляет ряд атрибутов, с помощью которых мы можем изменить стандартный механизм привязки.

BindRequired и BindNever

- **BindRequired** требует обязательного наличия значения для свойства модели.
- **BindNever** указывает, что свойство модели надо исключить из механизма привязки.

```
using Microsoft.AspNetCore.Mvc.ModelBinding;

public class User
{
    public int Id { get; set; }

    [BindRequired]
    public string Name { get; set; }

    public int Age { get; set; }
    [BindNever]
    public bool HasRight { get; set; }
}
```

BindRequired и BindNever

Если для свойства с атрибутом BindRequired не будет передано значение, то в объект ModelState будет помещена информация об ошибках, а свойство ModelState.IsValid возвратит false. И в данном случае, проверяя значение ModelState.IsValid, мы можем проверить корректность создания объекта.

BindingBehavior

Кроме того, мы можем применять атрибут **BindingBehavior**, который устанавливает поведение привязки с помощью одно из значений одноименного перечисления BindingBehavior:

- **Required**: аналогично применению атрибута BindRequired
- **Never**: аналогично применению атрибута BindNever
- **Optional**: действие по умолчанию, мы можем передавать значение, а можем и не передавать, тогда будут применяться значения по умолчанию

```
[BindingBehavior(BindingBehavior.Required)]
[BindingBehavior(BindingBehavior.Optional)]
[BindingBehavior(BindingBehavior.Never)]
```

Источники привязки

Ранее говорилось про порядок обхода привязчиком модели различных источников для получения значений.

Но группа атрибутов позволяет переопределить это поведения, указав один целевой источник для поиска значений:

- [**FromHeader**]: данные берутся из заголовоков запроса
- [**FromQuery**]: данные берутся из строки запроса
- [**FromRoute**]: данные берутся из значений маршрута
- [**FromForm**]: данные берутся из полученных форм
- [**FromBody**]: данные берутся из тела запроса. Этот атриут может применяться, когда в качестве источника данных выступает не форма и не строка запроса, а, скажем, данные отправляются через код javascript. FromBody может применяться, если метод имеет только один параметр, иначе будет сгенерировано исключение.

Источники привязки

Например, получим данные о юзер-агенте из запроса:

```
public IActionResult GetUserAgent([FromHeader(Name="User-Agent")] string
userAgent)
{
    return Content(userAgent);
}
```

Атрибуты валидации

С помощью атрибутов валидации модели мы можем управлять валидацией и заключать несложную логику проверки значений свойств уже в атрибуты этих свойств, не прибегая к коду. Рассмотрим атрибуты валидации, которые мы можем применить в приложении на ASP.NET Core.

1. Атрибут Required

Применение этого атрибута к свойству модели означает, что данное свойство должно быть обязательно установлено.

```
public class Person
{
        [Required (ErrorMessage = "Не указано имя")]
        public string Name { get; set; }
        [Required]
        public string Password { get; set; }
}
```

Атрибуты валидации

2. Атрибут RegularExpression

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

Атрибуты валидации

3. Атрибут StringLength

Чтобы пользователь не мог ввести очень длинный текст, применяется атрибут StringLength. Первым параметром в конструкторе атрибута идет максимальная допустимая длина строки. Именованные параметры, в частности **MinimumLength** и ErrorMessage, позволяют задать дополнительные опции отображения.

```
public class Person
{
    [Required (ErrorMessage = "Не указано имя")]
    [StringLength(50, MinimumLength = 3, ErrorMessage = "Длина строки должна быть от 3 до 50 символов")]
    public string Name { get; set; }
}
```

Атрибуты валидации

4. Атрибут Range

Атрибут Range определяет минимальные и максимальные ограничения для числовых данных.

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

Атрибуты валидации

5. Атрибут Compare

Атрибут Compare гарантирует, что два свойства объекта модели имеют одно и то же значение. Если, например, надо, чтобы пользователь ввел пароль дважды:

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

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

Атрибуты валидации

5. Специальные атрибуты

Мы применяли регулярное выражение для проверки адреса электронной почты. Проверки на корректность электронной почты, адреса url, номера телефона и кредитной карты довольно часто встречаются, что для них определены специальные атрибуты:

[CreditCard]

[EmailAddress]

[Phone]

[Url]

Аннотации данных

Кроме атрибутов валидации модели также могут иметь дополнительные атрибуты, которые называются аннотации данных и которые располагаются в пространстве имен *System.ComponentModel.DataAnnotations*. Эти атрибуты определяют различные правила для отображения свойств модели.

1. Атрибут Display

Атрибут Display задает параметры отображения для свойства.

```
public class Person
{
    [Display(Name="Имя и фамилия")]
    public string Name { get; set; }

    [Display(Name = "Пароль")]
    public string Password { get; set; }
}
```

Аннотации данных

2. Исключение из модели

Иногда возникают ситуации, когда надо, наоборот, исключить сущность из модели. Например, в примере выше сущность Phone ссылается на класс Company, и, допустим, мы не хотим, чтобы в базе данных была таблица Company.

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }
    // навигационное свойство
    public Company Manufacturer { get; set; }
}

[NotMapped]
public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Аннотации данных

3. Сопоставление таблиц

Атрибут Table позволяет переопределить сопоставление с таблицей по имени:

```
[Table("People")]
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Аннотации данных

4. Сопоставление столбцов

Атрибут Column переопределяет сопоставление:

```
public class User
{
    [Column("user_id")]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Аннотации данных

5. Настройка ключей и индексов

По умолчанию в качестве ключа используется свойство, которое называется Id или [имя_класса]Id.

Для установки свойства в качестве первичного ключа с помощью аннотаций применяется атрибут [Key]:

```
public class User
{
    [Key]
    public int Ident { get; set; }
    public string Name { get; set; }
}
```

Аннотации данных

6. Альтернативные ключи

Альтернативные ключи представляют свойства, которые также, как и первичный ключ, должны иметь уникальное значение. В то же время альтернативные ключи не являются первичными. На уровне базы данных это выражается в установке для соответствующих столбцов ограничения на уникальность.

Осуществляется с использованием Fluent API (см. документацию).

Аннотации данных

7. Значение по умолчанию и вычисляемые столбцы

Осуществляется с использованием Fluent API (см. документацию).

8. Атрибут MaxLength

В аннотациях данных ограничение по длине устанавливается с помощью атрибута MaxLength:

```
public class User
{
    public int Id { get; set; }
    [MaxLength(50)]
    public string Name { get; set; }
}
```