GeekBrains

ASP.NET MVC и углубление в изучение C#

MVC. Итоговый проект

На этом уроке

1. Рассмотрим валидацию
2. Узнаем, какую валидацию нужно применять
3. Тонкости про контроллеры
4. Отправку сообщений с помощью SMTP
5. Итоговой проект

Оглавление

Валидация моделей

Каждый запрос, поступающий в ASP, проходит проверку. Результаты проверки хранятся в ModelState, который создается и заполняется на каждый запрос. ModelState живет в контексте запроса. Его можно получить, напрямую обратившись в свойство ModelState внутри метода контроллера. Состояние модели может пополняться из валидации модели и привязки (байдинга) модели. Ошибки от байдинга, которые попадают в стейт валидации, в основном связаны с ошибками конвертации данных (например, невозможность привести строку к числу) или с отсутствием обязательного значения. Ошибки валидации добавляются после привязки модели и сообщают о нарушении каких-либо бизнес-правил. Например, возраст человека не может быть меньше 0 или отрицательным. Оба механизма отрабатывают до запуска метода контроллера или обработчика RazorPages. Проверка состояния модели на ошибки является задачей разработчика. Для проверки корректности можно проверить поле ModelState.IsValid. Обычно в случае наличия ошибок, например, после отправки веб-формы происходит отрисовка той же страницы, но уже с указанием самих ошибок. Сам по себе механизм валидации автоматический его не нужно явно вызывать, но в случае необходимости можно очистить текущее состояние ошибок через ModelState.ClearValidationState и повторно вызвать TryValidateModel.

Валидационные атрибуты

Существует целое множество проверок данных, которые необходимо проводить с пользовательским вводом до основной работы с данными. Хорошим тоном считается запускать функциональные и логические проверки на любых данных, которые приходят от пользователя. Для решения этой задачи есть готовый набор атрибутов, которые облегчают проведение проверки: CreditCard, Compare EmailAddress, Phone Range, RegularExpression, Required, StringLength Url.

```
public class Movie
{
	public int Id { get; set; }

	[Required]
	[StringLength(100)]
	public string Title { get; set; }

	[ClassicMovie(1960)]
	[DataType(DataType.Date)]
	[Display(Name = "Release Date")]
	public DateTime ReleaseDate { get; set; }

	[Required]
	[StringLength(1000)]
	public string Description { get; set; }

	[Range(0, 999.99)]
	public decimal Price { get; set; }

	public Genre Genre { get; set; }

	public bool Preorder { get; set; }
}
```

Например, StringLength имеет значение максимальное и минимальное значение входной строки.

Сообщение об ошибках

При наступлении ошибки в части валидации входных данных модель состояния будут добавляться ошибки. Текст этих ошибок определен в атрибутах и может быть переопределен. Например [StringLength(8, ErrorMessage = "Длина имени не может быть больше 8 символов")]. В тексте ошибки могут использоваться маски шаблонизации [StringLength(8, ErrorMessage = "{0} длина должна быть в пределах {2} и {1}.", MinimumLength = 6)]. Шаблоны строк можно посмотреть в документации или исходниках.

Необязательные поля и nullable типы

Подсистема валидации смотрит на типы, которые не могут содержать null-значение как на обаятельные, как если бы они были помечены атрибутом [Required]. В зависимости от конфигурации nullable поля могут приводить к появлению ошибок. В базе виде ошибки для nullable полей будут отсутствовать;

```
public class Person

{

      public string ShouldPresentWillTreatAsError { get; set; }

      public string? CanBeNullWillNotTreatAsError { get; set; }

}
```

Собственные валидационные атрибуты

В случае отсутствия готового атрибута можно сделать собственный. Для этого нужно сделать собственный класс, унаследованный от ValidationAttribute и переопределить метод IsValid, в котором определяется логика проверки:

```
public class DoNotHaveWords : ValidationAttribute

{

   private string[] _words;



   public DoNotHaveWords(string[] words)

   {

      _words = words;

   }



        protected override ValidationResult IsValid(object? value,
ValidationContext validationContext)

   {

      var bad = new HashSet<string>(_words,
StringComparer.InvariantCultureIgnoreCase);

      var request =
(CreateWeatherForecastRequest)validationContext.ObjectInstance;

      if (string.IsNullOrEmpty(request.Summary)) return
ValidationResult.Success;

      var words = new HashSet<string>(request.Summary.Split("  "),
StringComparer.InvariantCultureIgnoreCase);

      foreach (var word in words)

      {

            if (bad.Contains(word))

            return new ValidationResult("Описание содержит запрещенные
слова", new[] {nameof(request.Summary)});

      }

      return ValidationResult.Success;

   }

}
```

Различие между валидацией

В целом, стоит учитывать, что валидация существует двух видов – серверная и клиентская. Серверная валидация срабатывает уже после того, как пользователь отправил запрос на сервер. Клиентская же валидация не дает отправить неправильные данные на сервер. Стоит ли два раза перепроверять данные от пользователя и на клиенте, и на сервере? Краткий ответ – да, стоит. Стоит учитывать, что запрос может быть отправлен и не через ваше приложение, а, к примеру, через утилиту Postman, и тут клиентская валидация не отработает никак. Серверная валидация - один из самых важных и нужных элементов построения веб-приложений. Клиентская же помогают пользователю показать там, где он ошибся при вводе данных и не дает делать лишние запросы на сервер.

Еще раз немного про контроллеры, валидацию и хороший тон

Когда вы работаете с контроллерами, стоит учитывать, что логики в них должно быть как можно меньше. Есть такая проблема под название «проблема жирных контроллеров». Заключается она в том, что контроллеры становятся слишком большими, содержат какую-либо бизнес-логику и сложно управляемы. К тому же слой бизнес-логики сложно будет использовать в другом каком-либо приложении. Старайтесь как можно меньше вносить кода в код контроллера. По факту контроллеры должны просто дергать нужные методы у слоя бизнес-логики и максимум только проводить первичную валидацию модели, но никак не проверять саму бизнес-логику.

Работа с протоколом SMTP

В net core встроенной поддержки SMTP протокола для отправки почтовых сообщений нет. Для этого стоит использовать nuget-пакеты сторонних разработчиков, к примеру пакт под названием MailKit. Это очень распространенная библиотека, которая активно используется, развивается и легко интегрируется в проект.

Для конфигурации нужно указать почтовый ящик, от которого будет отправлено, SMTP сервер и авторизовать сервис в этом сервере. Но для начала нужно создать класс, который будет олицетворять наше сообщение.

```
using System;
using System.Threading.Tasks;
using MailKit.Net.Smtp;
using MimeKit;

namespace Interview.Sample
{
      public sealed class Message
      {
      public string Subject { get; set; }

      public string Body { get; set; }

         public string To { get; set; }

      public string Name { get; set; }

      public bool IsHtml { get; set; }
      }
}
```

Обратите внимание, что письмо может быть сформировано как в виде HTML, так и в виде обычного текста. Следующем моментом нужно подготовить класс по работе с конфигурацией почтового сервера.

```
using System;
using System.Threading.Tasks;
using MailKit.Net.Smtp;
using MimeKit;

namespace Interview.Sample
{
	public sealed class MailGatewayOptions
	{
	private const int DefaultPort = 25;

	public MailGatewayOptions()
	{
		Port = DefaultPort;
	}

	public string SenderName { get; set; }

	public string SMTPServer { get; set; }

	public int Port { get; set; }

	public string Sender { get; set; }

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

Порт 25 является дефолтным портом для SMTP-сервера. Но зачастую его меняют, так как атаки на сетевые ресурсы по общепринятым портам сильно развиты среди киберпреступников.

Сам класс отправки сообщений очень прост. В конструкторе он будет принимать настройки для сервера, наследоваться будет от интерфейса IDisposable, для корректного завершения работы с SMTP сервером.

```
using System;
using System.Threading.Tasks;
using MailKit.Net.Smtp;
using MimeKit;

namespace Interview.Sample
{
	internal sealed class MessageGateway : IDisposable
	{
	private readonly MailGatewayOptions _options;
	private readonly SmtpClient _client = new SmtpClient();

	public MessageGateway(MailGatewayOptions options)
	{
		if (options is null)
		{
		  throw new ArgumentNullException(nameof(options));
		}

		_options = options;

		_client.Connect(options.SMTPServer, options.Port);
		_client.Authenticate(options.Sender, options.Password);
	}

	  public async Task SendMessage(Message message)
	{
		MimeMessage emailMessage = new MimeMessage();
```

```
            emailMessage.From.Add(new  MailboxAddress(_options.SenderName,
_options.Sender));

            emailMessage.To.Add(new           MailboxAddress(message.Name,
message.To));

            emailMessage.Subject = message.Subject;

            emailMessage.Body    =    new    TextPart(message.IsHtml    ?
MimeKit.Text.TextFormat.Html :  MimeKit.Text.TextFormat.Text)

            {

            Text = message.Body

            };



            await _client.SendAsync(emailMessage);

      }



      public void Dispose()

      {

            _client.Disconnect(true);

            _client.Dispose();

      }

      }

}
```

Итоговой проект

Для закрепления курса требуется сделать сервис, который занимается рассылкой сообщений зарегистрированным пользователям. В нем должны быть реализованы абстракции шлюзов, но действующим шлюзом (предусмотреть развитие на различные сообщения – смс, пуш и т. д.) должен быть только пока шлюз отправки электронной почты. Электронное письмо должно формироваться по

шаблону зарегистрированном в системе, к примеру Razor шаблон. Предусмотреть так же обязательно отправку сообщений в определенное время (вспомнить Quartz), к примеру - отчеты для менеджеров.

Сервисы отправки сообщений являются частыми запросами современных решений. Они либо агрегируют несколько типов сообщений либо пишутся на каждый тип свой сервис.

Практическое задание

1. Сервис по отправке различных сообщений. Должен работать как по принудительной отправки, так и по расписанию (отчеты). Текст сообщений должен проходить через шаблонизатор.

Дополнительные материалы

1. Статья «Razor».

Используемые источники

1. Статья «Razor Engine».
2. Статья Razor Engine