

# ASP.NET Core MVC Modellvalidierung

Ingo Köster

Diplom Informatiker (FH)

# Motivation

---

- › Bevor eine Anwendung Daten eines Anwenders z.B. in einer Datenbank speichert, müssen diese überprüft werden
  - › Insbesondere auf Fehleingaben
  - › Sowie auf mögliche Sicherheitsbedrohungen
- › Anwenderdaten dürfen niemals ungeprüft übernommen werden!
  - › Führt zu inkonsistenten Daten und/oder zu Sicherheitslücken
- › Häufig werden die Daten daher beim Client und auch beim Server validiert

# Manuelle Validation

---

- › Die einfachste und wenig wartungsfreundliche Option
- › Überprüfung aller relevanten Modell-Eigenschaften in der ActionMethode
- › Muss bei Bedarf einzeln geändert werden

```
[HttpPost]
public ActionResult Create(Book book)
{
    try
    {
        if (book.Title != null && book.Title.Length >= 2)
        {
            ...
        }
    }
    catch
    {
        ...
    }
}
```

# Validierungsattribute

---

- › Durch Validierungsattribute kann die Validierung ähnlich dem Konzept zum Validieren von Feldern in Datenbanktabellen verwendet werden
- › Diese Attribute enthalten Validierungscode, sodass Entwickler weniger Code schreiben müssen
- › Diese Attribute beinhalten Einschränkungen wie z.B.
  - › Zuweisen von Pflichtfeldern
  - › Zuweisen von gültigen Bereichen
  - › Muster wie E-Mail, Telefonnummer, Kreditkarte, etc.

# Validierungsattribute

---

Attribut	Wirkung
Required	Prüft auf Pflichtfeld
Range	Prüft auf Wert im Intervall
RegularExpression	Prüft auf angegebenen Ausdruck
StringLength	Prüft auf maximale Länge (und minimale)
Compare	Prüft ob zwei Felder gleich sind, z.B. Passworte
CreditCard	Prüft auf das Format einer Kreditkartennummer
EmailAddress	Prüft auf das Format einer E-Mail-Adresse
Url	Prüft auf das Format einer Url
Phone	Prüft auf das Format einer US-Telefonnummer
FileExtensions	Prüft einen Dateinamen auf Endung(en)

# Beispiel

---

- › Modell Klasse mit Validierungsattributen

```
public class Book
{
    public int BookID { get; set; }

    [Required]
    [StringLength(200, MinimumLength = 2)]
    public string BookTitle { get; set; }

    [Range(1, 500000)]
    public int Pages { get; set; }
}
```

# Beispiel

---

- › Beispiel aus dem Model

```
[Required]
```

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

- › Das Attribut wird von den Helfern ausgewertet

```
@Html.EditorFor(model => model.Title)
```

```
@Html.ValidationMessageFor(model => model.Title)
```

- › Erzeugt eine Meldung, falls der Titel nicht angegeben wurde

- › Das Feld "Title" ist erforderlich.

- › Clientseitige Validierung wird beim Verlassen der Textbox ausgeführt

# Validierung und Datentypen

---

- › Zu einem Datentyp muss das passende Attribut gewählt werden
- › 5 bis 10 Zeichen
  - › [StringLength(10, MinimumLength = 5)]
  - › `public string Title { get; set; }`
- › 5 bis 10 Ziffern
  - › [MinLength(5)]
  - › [MaxLength(10)]
  - › `public int Seitenzahl { get; set; }`



# Notwendige JavaScript Bibliotheken

---

- › Damit die Validierung auf der Clientseite funktioniert, müssen über eine View oder eine Layoutseite die entsprechenden JavaScript-Verweise eingefügt werden

```
<script src="~/lib/jquery/dist/jquery.js"></script>
```

```
<script src="~/lib/jquery-  
validation/dist/jquery.validate.js"></script>
```

```
<script src="~/lib/jquery-validation-  
unobtrusive/jquery.validate.unobtrusive.js"></script>
```

- › Diese Dateien sind in der Projektvorlage bereits enthalten, können aber auch bei Bedarf z.B. mittels LibMan installiert werden

# \_ValidationScriptsPartial.cshtml

---

- › Die Datei \_ValidationScriptsPartial.cshtml im Ordner Shared enthält bereits alle Anweisungen zum Einbinden der Validierungsskripte
  - › Ist eine sog. Partielle View (Partial View)
- › Um diese Datei in eine View oder Layout-Seite einzubinden, kann folgende Anweisung verwendet werden
  - › `@{await Html.RenderPartialAsync("_ValidationScriptsPartial"); }`

```
73      <script src="~/js/site.js" asp-append-version="true"></script>
74
75      @{ await Html.RenderPartialAsync("_ValidationScriptsPartial"); }
76
77      @RenderSection("Scripts", required: false)
78  </body>
```

# Validierung serverseitig prüfen

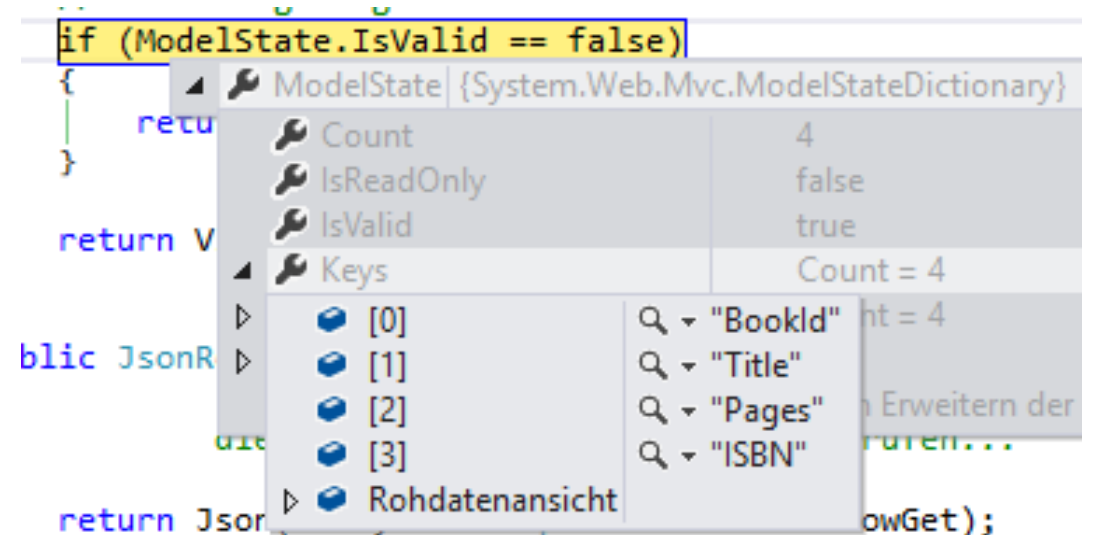
---

- › Wichtig z.B. bei komplexer Validierung durch eigene Validierungsattribute

```
[HttpPost]
public ActionResult Index(Book b) {
    if (ModelState.IsValid == false) {
        return View(b);
    }
    return View("Erfolg");
}
```

# Validierung serverseitig prüfen

- › ModelState ist ein Objekt des Controllers
- › Gibt Informationen über den Zustand der Modellbindung
- › Enthält Collections für Schlüssel, Werte und Fehler



# Komplexe Validierung

---

- › Die bisher vorgestellten Attribute sind für einfache aber häufig vorkommende Daten geeignet
- › Komplexere Prüfungen oder Prüfungen gegen Datenbestände können so nicht realisiert werden
- › Dafür gibt es:

Attribut	Wirkung
Remote	Prüft anhand eines Dienstes/einer Methode, die mittels AJAX angesprochen wird
ValidationAttribute	Basisklasse für die Entwicklung eigener, komplexer Attribute

# Remotevalidierung

---

- › In manchen Fällen sollen Daten auf dem Client mit Daten auf dem Server abgeglichen werden
- › Beispielsweise soll überprüft werden, ob eine E-Mail-Adresse oder ein Benutzername bereits verwendet wird
- › Um für die Validierung von z.B. nur einem Feld nicht das gesamte Modell posten zu müssen, kann eine sog. Remotevalidierung verwendet werden

# Remotevalidierung

---

› Das Model benötigt für das zu validierende Element das Attribut [Remote]

› Beispiel

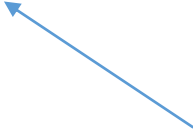
```
[Remote(action: "CheckISBN", controller: "Home")]  
public string ISBN { get; set; }
```

› Die Übergabeparameter setzen die ActionMethode und den Controller, an welche die zu validierenden Daten gesendet werden sollen

# Die ActionMethode für die Remotevalidierung

---

```
public IActionResult CheckISBN(string isbn) {  
    bool state = false;  
  
    if (isbn.Length == 10 || isbn.Length == 13)  
    {  
        // Hier die übergebene Isbn tatsächlich prüfen...  
        state = true;  
    }  
    return Json(state);  
}
```



Der Übergabeparameter  
muss mit dem Namen  
des Properties übereinstimmen  
(nicht Casesensitiv)



# Rückgabewert der Remotevalidierungsmethode

---

- › Die ActionMethode für die Remotevalidierung muss einen JSON-String zurück geben
- › "true"
  - › für gültige Elemente
- › "false", undefined oder null
  - › für ungültige Elemente
    - › Dafür wird eine Standardfehlermeldung verwendet
- › Wenn die ActionMethode einen String als JSON zurückgibt, wie z.B. "Der Benutzername ist bereits vergeben.", wird dieser anstelle der Standardmeldung angezeigt

# Remotevalidierung mit mehreren Werten

---

- › Mittels des Parameters `AdditionalFields`
  - › `[Remote(action: "CheckIt", controller: "Home", AdditionalFields = nameof(Username))]`
  - › `public int UserId { get; set; }`
  - › `public string Username { get; set; }`
- › Im Home-Controller
  - › `public IActionResult CheckIt(int userId, string username) { ... }`

# Eigene Validierungsattribute

---

- › Um eigene Attribute zu erstellen, wird eine Klasse erstellt, welche von der Klasse `ValidationAttribute` erbt
- › In dieser Klasse wird die `IsValid`-Methode überschrieben
- › Die `IsValid`-Methode hat den Parameter `value` vom Typ `Object`
- › `value` bezieht sich auf den Wert aus dem Feld, welches das eigene Validierungsattribut überprüft

# Beispiel

---

```
public class RatingAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        int rating = (int)value;

        return (rating > 0 && rating <= 5);
    }
}
```

# Verwendung in der Modellklasse

---

```
public class Film
{
    [ScaffoldColumn(false)]
    public int Id { get; set; }

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

    [Rating]
    public int Rating { get; set; }
}
```

## Film

Name

The Name field is required.

Rating

The field Rating is invalid.

Speichern

# Attribute in anderer Datei

---

## › Model

```
public partial class Person {  
    public string Name { get; set; }  
}
```

## › Model mit Metadaten

```
[ModelMetadataType(typeof(Metadata))]  
public partial class Person {  
    private sealed class Metadata {  
        [Required]  
        [Display(Name="Nachname:")]  
        public string Name { get; set; }  
    }  
}
```