06. Data Annotations and Validation

Buch Kap. 6

Validierung mit Data Annotationen

Eigene Validierungslogik

Modell Metadata Annotationen

- Validierung im Browser.
- Validierung auf dem Server ist ein **muss**.
  - Vertraue **nie** den eingehenden Informationen vom Netz, bzw. eines Client.
- Benutzerfreundliche Fehlermeldungen, d.h. in der Sprache des Benutzers (Lokalisierung).
- Validierung des Modells, bzw. der Modellwerte.
  - Deklarativer Stiel (data annotations).
  - Validierungsfunktionen erweiterbar.

Orders - Bestellungen

```
//Order.cs
public class Order
{
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Username { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public decimal Total { get; set; }
    public List<OrderDetail> OrderDetails { get; set; }
}
```

Validierungsannotationen

- `System.ComponentModel.DataAnnotations` Namespace.
  - Nur das `Remote` Validierungsattribut ist im Namespace `System.Web.Mvc`.
- Unterstützt/Generiert Server- und Clientseitige Validierung.
  - Falls JavaScript auf dem Client deaktiviert ist, so generiert die Serverseitige Validierung exakt die gleiche Fehlermeldung.

Required

- Das Attribut `Required` wirft einen Validierungsfehler, falls:
  - das Property `null` ist.
  - das Property **leer** ist, bzw. ein leerer String ("") ist.
  - das Property nur White-Space Zeichen enthält
- Liefert Server- und Clientseitige Validierungslogik.

```
//Order.cs
public class Order
{
   //...
   [Required]
   public string FirstName { get; set; }

   [Required]
   public string LastName { get; set; }
   //...
}
```

StringLength

- Das Attribut `StringLength(int maximumLength)` wirft einen Validierungsfehler, falls:
  - das Property mehr Zeichen als maximumLength aufweist.
  - *(das Property mehr Bytes als maximumLength aufweist)*
- `MinimumLength` ist ein optionaler, benannter Parameter

```
//Order.cs
public class Order
{
   //...
   [Required]
   [StringLength(160)]
   public string FirstName { get; set; }

   [Required]
   [StringLength(160, MinimumLength=5)]
   public string LastName { get; set; }
}
```

RegularExpression

- Das Attribut `RegularExpression(string pattern)` wirft einen Validierungsfehler, falls:
  - das Property nicht dem regulären Ausruck `pattern` entspricht.
- Vorsicht: Es wird kein Fehler geworfen, falls das Property `null` oder ein leerer String ("") ist.

```
//Order.cs
public class Order
{
   //...
   [RegularExpression(
              @"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
   public string Email { get; set; }
   //...
}
```

Range

- Das Attribut `Range(minimum, maximum)` legt das Minimum und Maximum eines nummerischen Wertes fest:
  - Minimum und Maximum sind inklusive.
  - `int, double, IComparable`
- Liefert Server- und Clientseitige Validierungslogik.

```
//Order.cs
public class Order
{
   //...
   [Range(35,44)]
   public int Age { get; set; }

   [Range(typeof(decimal), "0.00", "49.99")]
   public decimal Price { get; set; }
   //...
}
```

Compare

- Das Attribut `Compare(string otherProperty)` garantiert, dass zwei Properties eines Modells identische Werte haben.
- Liefert Server- und Clientseitige Validierungslogik.

```
//Order.cs
public class Order
{
   //...
   [RegularExpression(
              @"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
   public string Email { get; set; }

   [Compare("Email")]
   public string EmailConfirm { get; set; }
   //...
}
```

Remote

- `System.Web.Mvc` Namespace.
- `Remote` führt eine Client-Seitige Validierung mit einem Server Callback durch.
  - Z.B. ein username muss eindeutig sein. Aber nur der Server kennt alle usernames…
  - `Remote(action, controller)`

```
//Order.cs
public class Order
{
   //...
   [Remote("CheckUserName", "Account")]
   public string UserName { get; set; }
   //...
}
```

Remote

```
//AccountController.cs
//...
public JsonResult CheckUserName(string username)
{
   var result = Membership.FindUsersByName(username).Count == 0;
   return Json(result, JsonRequestBehavior.AllowGet);
}
//...
```

- Die Funktion gibt das Resultat, ein `true` oder `false`, als JSON (JavaScript Object Notation) zurück.
- Später in Kapitel 8.

Benutzerdefinierte Fehlermeldungen und Lokalisierung

- Jedem Validierungsattribut kann der benannter Parameter `ErrorMessage` mit einer benutzerdefinierten Fehlermeldung übergeben werden.

```
//Order.cs
public class Order
{
   //...
   [RegularExpression(
               @"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
               ErrorMessage="Ungültige E-Mail-Adresse.")]
   public string Email { get; set; }

   [Required(ErrorMessage="Your last name is required")]
   [StringLength(160, ErrorMessage="Your last name is too long")]
   public string LastName { get; set; }

   //...
}
```

Benutzerdefinierte Fehlermeldungen und Lokalisierung

- `ErrorMessage` kann ein einfaches Formatelement `{0}` (single format item) enthalten.
  - Der benutzerfreundliche Anzeigename wird ausgegeben.

```
//Order.cs
public class Order
{
   //...
   [Required(ErrorMessage="Your {0} is required.")]
   [StringLength(160, ErrorMessage="{0} is too long.")]
   public string LastName { get; set; }
   //...
}
```

- Fehlermeldungen:
  - Your LastName is required.
  - LastName is too long.

Lokalisierung

- Mehrsprachige Fehlermeldungen können über Ressourcen definiert werden.
- Der folgende Code setzt eine Ressource `ErrorMessages.resx` mit den entsprechenden Einträgen `LastNameRequired` und `LastNameTooLong` im Projekt voraus.

```
//Order.cs
public class Order
{
   //...
   [Required(ErrorMessageResourceType=typeof(ErrorMessages),
             ErrorMessageResourceName="LastNameRequired")]
   [StringLength(160,
                 ErrorMessageResourceType = typeof(ErrorMessages),
                 ErrorMessageResourceName = "LastNameTooLong")]
   public string LastName { get; set; }
   //...
} //http://msdn.microsoft.com/en-us/library/bz9tc508.aspx
```

- Validierungsfeature von ASP.NET MVC ist Teil eines Systems von
  - Model binders
  - Model metadata
  - Model validators
  - Model state

Validierung und Model Binding

- Wann genau findet die Validierung statt?
- Per Standard bei der Modellbindung.
  - Implizit wenn Parameter an eine Action Methode übergeben werden.
  - Explizit über Aufruf von `UpdateModel()` oder `TryUpdateModel()`.
- Nachdem der Modellbinder alle Properties aktualisiert hat, führt er den **Model Validator** (`DataAnnotationsModelValidator`) aus.
  - Findet über Metadaten alle Validierungsattribute (DataAnnotations) und führt ihre Validierungslogik aus.
- Modellbinder fängt alle Validierungsfehler und speichert sie im Model State.

Validierung und Model State

- Model State
  - Zugreifbar über die Eigenschaft `ModelState` des `Controllers`.
  - Enthält die gebundenen Werte.
  - Enthält die vom Benutzer ursprünglich eingegeben Werte (mit Fehlern?).
  - Enthält alle den Properties zugeordneten Validierungsfehler.
  - Enthält alle dem Modell selbst (als ganzes) zugeordneten Validierungsfehler.
- `ModelState.IsValid`
- `ModelState.IsValidField(string fieldName)`
  - `ModelState.IsValidField("LastName")`
- `ModelState[string fieldName].Errors`
  - `ModelState["LastName"].Errors.Count`
  - `ModelState["LastName"].Errors[0].ErrorMessage`
- Zur Erinnerung:
  - `@Html.ValidationMessageFor(m => m.LastName)`

Controller Actions and Validation Errors

- Controller entscheidet, was zu tun ist, falls Validierungsfehler auftreten, bzw. falls das Modell erfolgreich und ohne Fehler gebunden wird.
  - Bei Validierungsfehler: in der Regel dieselbe View regenerieren.
    - Der Benutzer sieht die Fehler und kann diese korrigieren.
  - Valides Modell: notwendige Schritte um die Daten weiter zu verarbeiten, bzw. zu speichern oder zu aktualisieren.

Controller Actions and Validation Errors

```
//implizit...
public ActionResult AddressAndPayment(Order newOrder)
{
    if (ModelState.IsValid)
    {
        newOrder.Username = User.Identity.Name;
        newOrder.OrderDate = DateTime.Now;
        storeDB.Orders.Add(newOrder);
        storeDB.SaveChanges();
        // Process the order
        var cart = ShoppingCart.GetCart(this);
        cart.CreateOrder(newOrder);
        return RedirectToAction("Complete",
                                new { id = newOrder.OrderId });
    }
    // Invalid -- redisplay with errors
    return View(newOrder);
}
```

Controller Actions and Validation Errors

```
//explizit...
public ActionResult AddressAndPayment(FormCollection collection)
{
    var newOrder = new Order();
    UpdateModel(newOrder);
    if(ModelState.IsValid) {//TryUpdateModel() noch besser!!
        newOrder.Username = User.Identity.Name;
        newOrder.OrderDate = DateTime.Now;
        storeDB.Orders.Add(newOrder);
        storeDB.SaveChanges();
        // Process the order
        var cart = ShoppingCart.GetCart(this);
        cart.CreateOrder(newOrder);
        return RedirectToAction("Complete",
                                new { id = newOrder.OrderId });
    }
    // Invalid -- redisplay with errors
    return View(newOrder);
}
```

Custom Validation Logic

- Validierungslogik im
  - eigener (Benutzer) Annotation
    - Vorteil der Wiederverwendbarkeit
  - Im Modelobjekt selbst
    - Vorteil: Ist einfacher zu implementieren

Benutzer Annotationen

- Alle Validierungsannotationen erben von der Basisklasse `ValidationAttribute`.
  - `System.ComponentModel.DataAnnotations`
- Eine der `IsValid()` Methoden muss überschrieben und die Validierungslogik implementiert werden.
  - IsValid(Object)
  - IsValid(Object, ValidationContext)
- Parameter `ValidationContext` gibt Zugriff auf
  - Modelltyp
  - Modellinstanz
  - Displayname der zu validierenden Eigenschaft
  - und weitere Informationen
- Validierungsparameter, z.B. MaxLenght oder MinValue werden über den Konstruktor definiert.

```
//überprüft, ob ein String eine maximale Anzahl Worte hat...
public class MaxWordsAttribute : ValidationAttribute
{
   private readonly int _maxWords;

   public MaxWordsAttribute(int maxWords)
   {
      _maxWords = maxWords;
   }
   protected override ValidationResult IsValid(object value,
                                 ValidationContext validationContext)
   {
      if (value != null)
      {
         var valueAsString = value.ToString();
         if (valueAsString.Split(' ').Length > _maxWords)
            return new ValidationResult("Too many words!");
      }
      return ValidationResult.Success;
   }
}
```

- Fest-codierte Fehlermeldung ist schlecht. Benutzer möchte u.U. eigene Fehlermeldung definieren.

```
public class MaxWordsAttribute : ValidationAttribute
{
  private readonly int _maxWords;

  public MaxWordsAttribute(int maxWords)
     :base("{0} has too many words.")
  {
    _maxWords = maxWords;
  }

  protected override ValidationResult IsValid(object value,
                                ValidationContext vc)
  {
    if (value != null) {
      var valueAsString = value.ToString();
      if (valueAsString.Split(' ').Length > _maxWords)
      {
        var errorMessage = FormatErrorMessage(vc.DisplayName);
        return new ValidationResult(errorMessage);
      }
    }
    return ValidationResult.Success;
  }
}
```

Benutzer Annotationen - Anwendung

```
//Order.cs
//...
[Required]
[StringLength(160)]
[MaxWords(10)]
public string LastName { get; set; }
//...
```

```
//Order.cs
//mit einer Benutzerfehlermeldung
//...
[Required]
[StringLength(160)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }
//...
```

- NuGet: Wrox.ProMvc5.Validation.MaxWordsAttribute

IValidatableObject

- Ein **selbst-validierendes Modell** weiss wie es sich selbst überprüft, bzw. validiert.
- Die Modellklasse muss die Schnittstelle `IValidatableObject` implementieren.

```
//Order.cs
public class Order : IValidatableObject
{
  public IEnumerable<ValidationResult> Validate(
                                ValidationContext validationContext)
  {
    if(LastName != null && LastName.Split(' ').Length > 10)
    {
      yield return new ValidationResult(
                                "The last name has too many words!",
                                new []{"LastName"});
    }
  }
}
```

IValidatableObject - Annotation

- Unterschiede:
  - Validierungsmethode: `Valid` vs. `IsValid`.
  - Rückgabetyp und Parameter sind verschieden.
    - `IEnumerable<ValidationResult>`
    - `ValidationResult`
    - Kein `value` Parameter.
  - `Valid()` überprüft das komplette Modell, bzw. alle Eigenschaften und nicht nur eine Property.
  - An `Valid` wird kein `value` Parameter übergeben.
    - Wir befinden uns schon in der Instanz und haben bereits Zugriff auf alle Eigenschaften.
  - yield return ?
  - An `ValidationResult` muss zusätzlich der/die Property-Name(n) übergeben werden, mit dem die Fehlermeldung verknüpft ist.
  - Oft einfacher zu implementieren, vor allem wenn mehrere Properties überprüft oder verglichen werden müssen, bzw. voneinander abhängen.

Display

- Das Attribut `Display` setzt einen benutzerfreundlichen Anzeigename für eine Property.
- Optional kann der zwite Parameter `Order` übergeben werden.
  - Legt die Reihenfolge fest, in der die Properties angeordnet werden.
  - Default-Wert ist 10'000.
  - Aufsteigende Reihenfolge.

```
//Order.cs
[Required, StringLength(160)]
[Display(Name="Last Name", Order=15001)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }

[Required, StringLength(160, MinimumLength=3)]
[Display(Name="First Name", Order=15000)]
public string FirstName { get; set; }
```

ScaffoldColumn

- Das Attribut `ScaffoldColumn` verbirgt eine Property vor den HTML Helpern.
  - wie z.B. vor `EditorForModel` und `DisplayForModel`.
  - Im untestehenden Beispiel mit dem gesetzten Attribut zeigt z.B. `EditorForModel` weder ein Input-Feld noch ein Label für das Property Username.

```
//Order.cs
[ScaffoldColumn(false)]
public string Username { get; set; }
```

- Overposting
  - Falls über das Formular trotzdem ein Parameter namens Username übergeben wird, könnte der Model Binder versuchen den Wert zu setzen… siehe Overposting in Kap. 7.

DisplayFormat

- Das Attribut `DisplayFormat` erlaubt Formatierungsmöglichkeiten über benannte Parameter.
  - Alternativer Anzeigetext, falls das Property `null` enthält.
  - HTML-Encodierung deaktivieren für Properties mit Markup.
  - Data Format String für den anzuzeigenden Property-Wert.
  - Im Folgenden Beispiel soll der Wert als Währung angezeigt werden.

```
//Order.cs
[DisplayFormat(ApplyFormatInEditMode=true, DataFormatString="{0:c}")]
public decimal Total { get; set; }
```

- `ApplyFormatInEditMode` ist per Default `false`.
  - Falls auf `true` gesetzt, wird der Wert in einem Formular-Input-Feld auch formatiert.
  - Das Währungssymbol führt in diesem Beispiel aber beim **Model Binder auf Fehler**, da dieser nur Ziffern erwartet.
    - -> eigenes Validierungsattribut definieren??

ReadOnly

- Das Attribut `ReadOnly` bewirkt, dass der Default Model Binder dem Property keinen neuen Wert von der Anfrage (Request) zuweist.
  - Beache: `EditorForModel` zeigt das Formularinputfeld trotzdem noch an.

```
//Order.cs
//...
[ReadOnly(true)]
public decimal Total { get; set; }
```

DataType

- Das Attribut `DataType` erlaubt es, den Zweck einer Property zu definieren.
  - Z.B. ein String könnte folgendes sein:
    - E-Mail Adresse
    - URL
    - Passwort
  - Im Falle eines Passworts rendern die Helper ein Input Element mit `type="password"`, bei der Eingabe werden nur Sterne angezeigt.

```
//Order.cs
//...
[Required]
[DataType(DataType.Password)]
[Display(Name="Password")]
public string Password { get; set; }
```

- Ebenso: `Currency`, `Date`, `Time` und `MultilineText`.

HiddenInput

- Das Attribut `HiddenInput` teilt den Helpern mit, ein Input-Feld mit dem `type="hidden"` zu generieren.
  - Informationen, welche der Benutzer nicht sehen soll, können so wieder zurück an den Server gesendet werden.
  - Ist aber nicht sicher. Ein User könnte die Daten manipulieren.