# Model Validation

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

## Lesson Goals

Learn about different viable ways to validate user input data in MVC

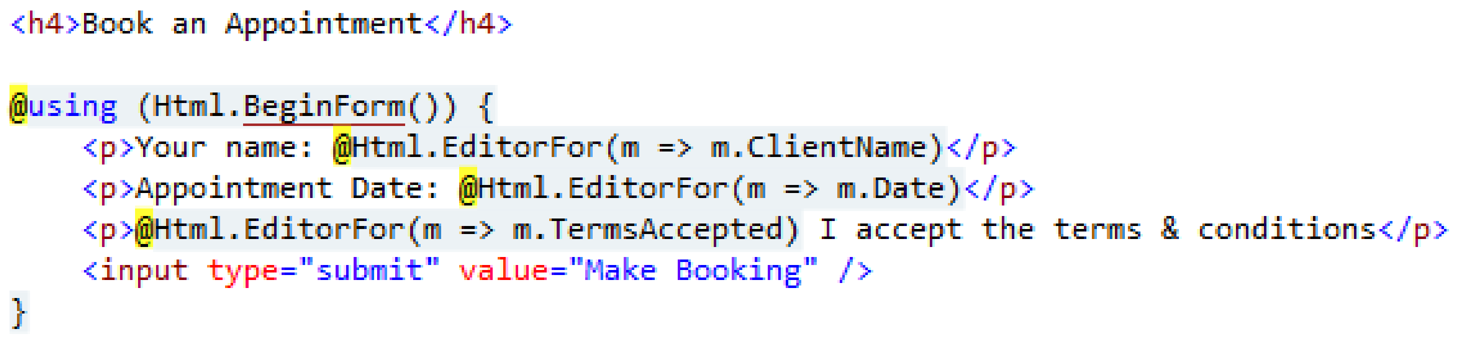
# Example Project

Let’s create a Basic MVC application to keep track of appointments.

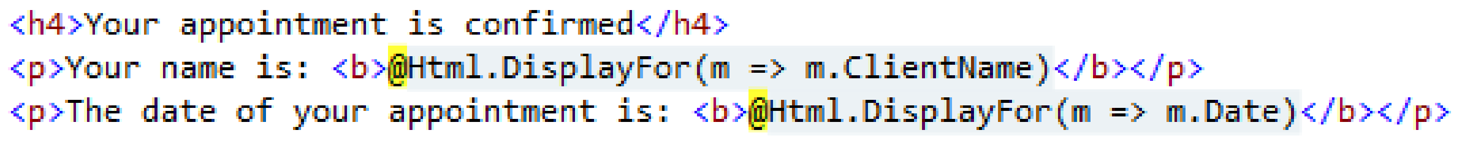
```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models
{
    public class Appointment
    {
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

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

```
public class HomeController : Controller
{
    public ViewResult MakeBooking()
    {
        return View(new Appointment { Date = DateTime.Now });
    }

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt)
    {

        // statements to store new Appointment in a
        // repository would go here in a real project

        return View("Completed", appt);
    }

}
```

## And Some Views



```
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```



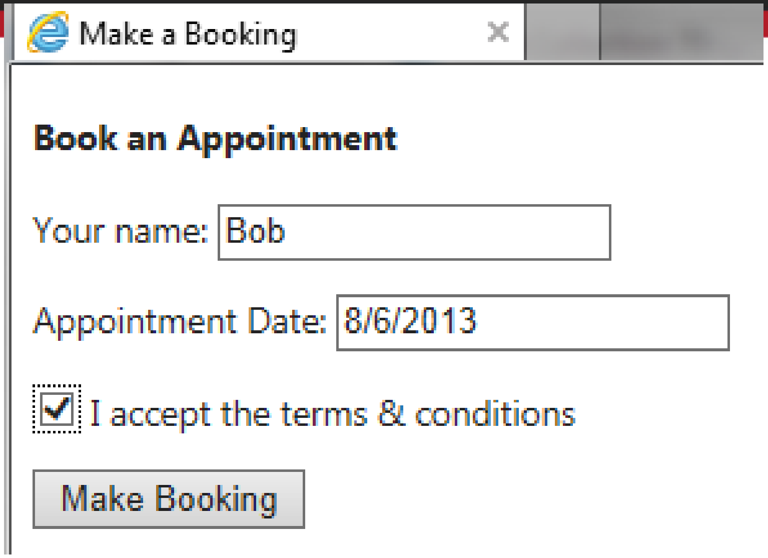
```
<h4>Your appointment is confirmed</h4>
<p>Your name is: <b>@Html.DisplayFor(m => m.ClientName)</b></p>
<p>The date of your appointment is: <b>@Html.DisplayFor(m => m.Date)</b></p>
```

## The Data Type Annotation

This annotation allows us to specify a data type on a model that has various effects.  In this case, DataType.Date displays the date without a time.

A full list of DataType values is available here.

# It Works…



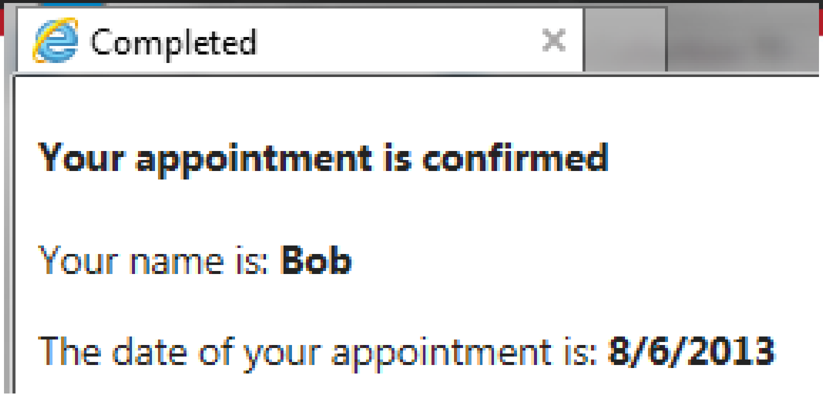
Make a Booking

**Book an Appointment**

Your name: Bob

Appointment Date: 8/6/2013

☑ I accept the terms & conditions



Completed

**Your appointment is confirmed**

Your name is: **Bob**

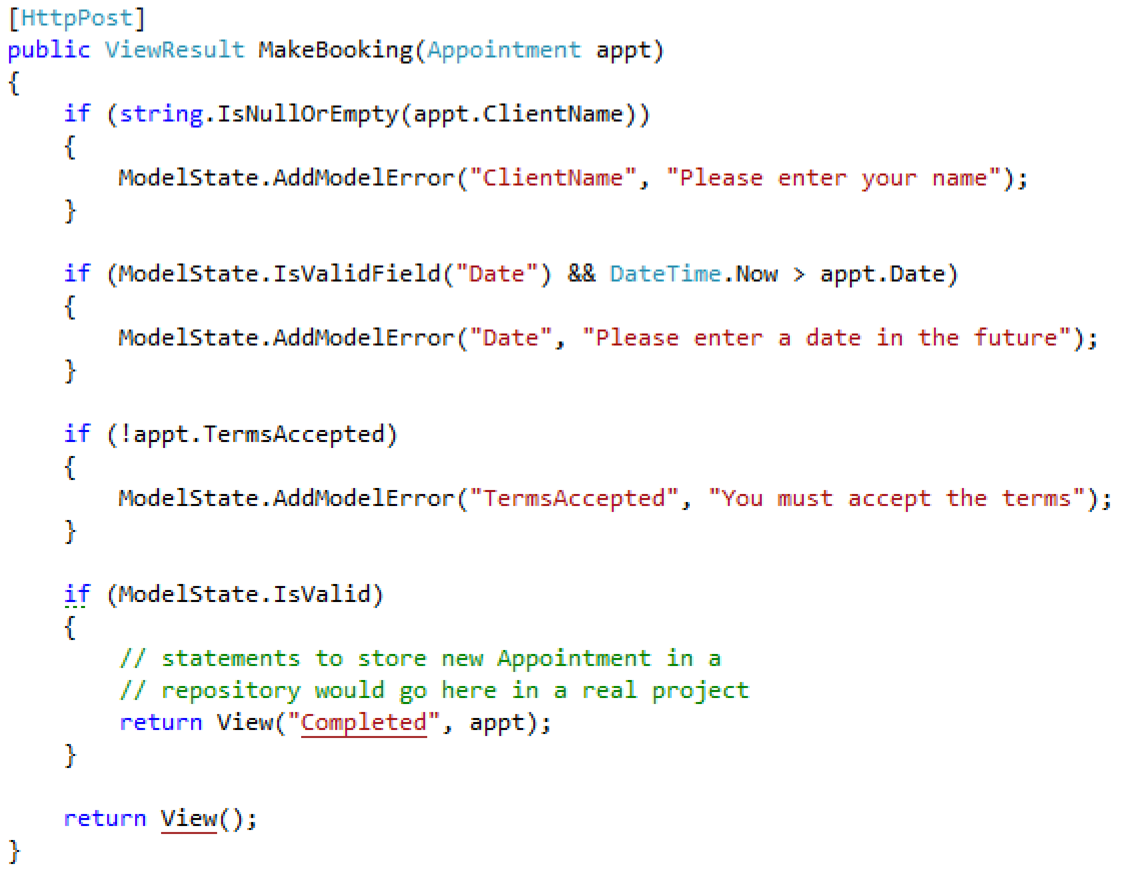
The date of your appointment is: **8/6/2013**

But it accepts any data we throw at it.  Ideally, the users must provide a name, provide a future date, and check the box to accept the conditions.

## Technique #1: Explicit Validation

One method for validating is to explicitly add errors directly to the ModelState property of the controller.

The ModelBinder has already done its work, so we can check out the resulting object and add any errors we find by using the ModelState.AddModelError("PropertyName", "Error Message"); method.



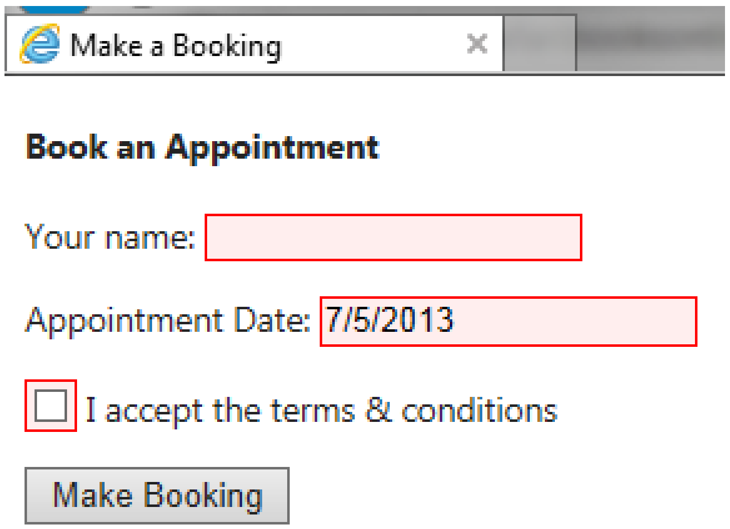
```
[HttpPost]
public ViewResult MakeBooking(Appointment appt)
{
    if (string.IsNullOrEmpty(appt.ClientName))
    {
        ModelState.AddModelError("ClientName", "Please enter your name");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date)
    {
        ModelState.AddModelError("Date", "Please enter a date in the future");
    }

    if (!appt.TermsAccepted)
    {
        ModelState.AddModelError("TermsAccepted", "You must accept the terms");
    }

    if (ModelState.IsValid)
    {
        // statements to store new Appointment in a
        // repository would go here in a real project
        return View("Completed", appt);
    }

    return View();
}
```

## When Validation Fails…

The fields now turn a shade of red. If we examine the rendered HTML using the View Source context menu or the developer tools (F12), we can see that each input now has a class on it called “input-validation-error.”

If we check our Site.css file, we will see the error styles defined there.

Of course, we haven’t told MVC how to display error messages, so none are shown. The impacted field names in the AddModelError are colored red.


Make a Booking

Book an Appointment

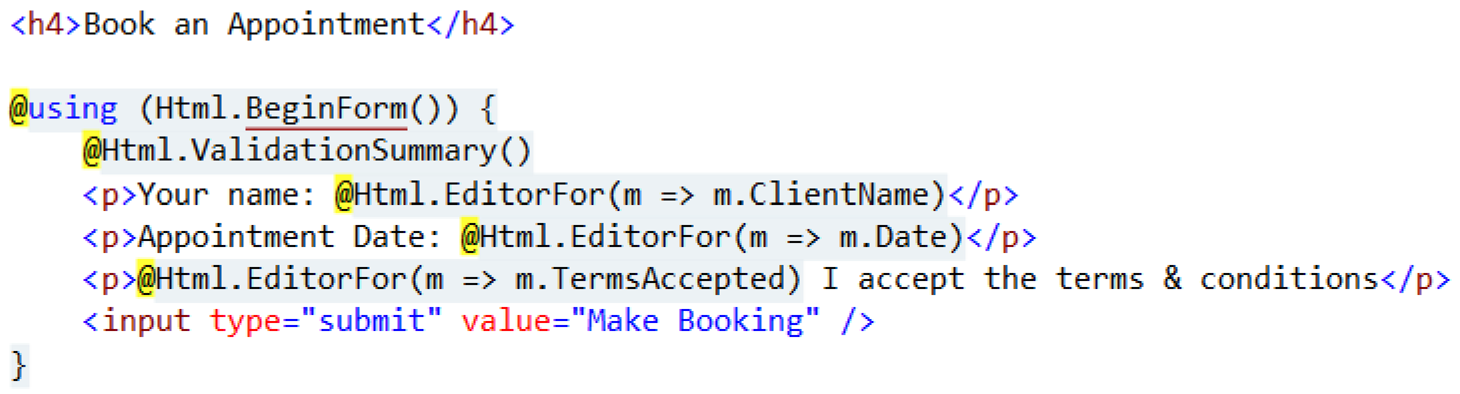
Your name:

Appointment Date: 7/5/2013

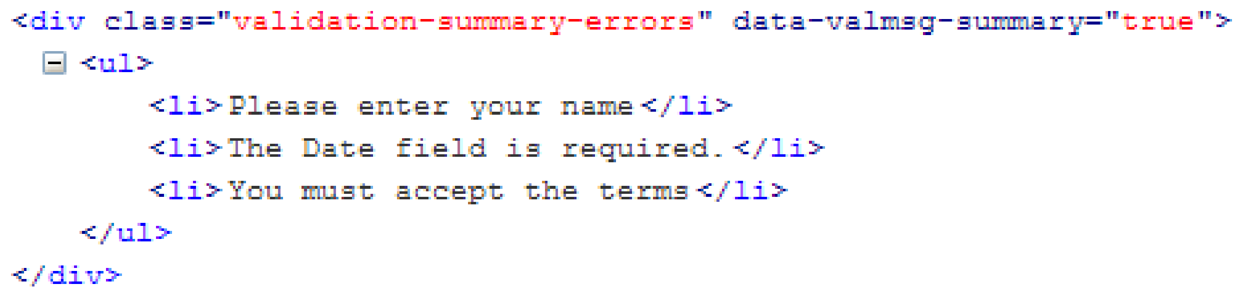
☐ I accept the terms & conditions

## Html.ValidationSummary()

The easiest way to catch validation errors on the form is to use the validation summary.  We can add it to the form using the helper, and a list of error messages (if any) will appear in line.



```
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    @Html.ValidationSummary()
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```



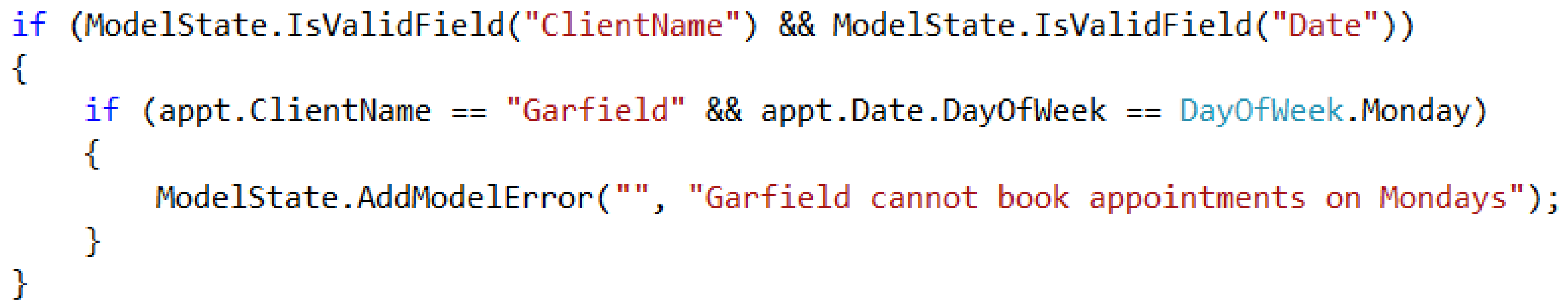
```
<div class="validation-summary-errors" data-valmsg-summary="true">
  <ul>
      <li>Please enter your name</li>
      <li>The Date field is required.</li>
      <li>You must accept the terms</li>
  </ul>
</div>
```

## Using ValidationSummary

| Method | Description |
|---|---|
| Html.ValidationSummary() | Displays all validation errors |
| Html.ValidationSummary(true) | Only model-level errors are displayed |
| Html.ValidationSummary(string) | Displays the string message before the summary of validation errors (Header) |
| HtmlValidationSummary(true, string) | Summary with only model level errors |

# Model-Level Errors

Let’s say that Garfield, who hates Mondays, shouldn’t be allowed to book appointments on that day.

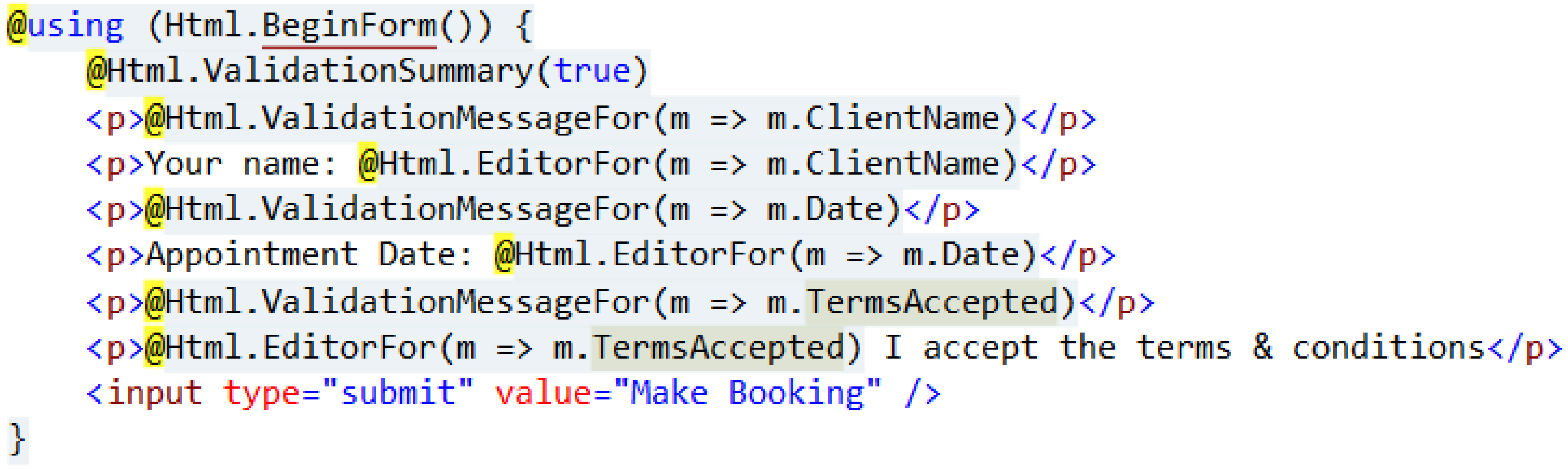


```
if (ModelState.IsValidField("ClientName") && ModelState.IsValidField("Date"))
{
    if (appt.ClientName == "Garfield" && appt.Date.DayOfWeek == DayOfWeek.Monday)
    {
        ModelState.AddModelError("", "Garfield cannot book appointments on Mondays");
    }
}
```

By leaving the field blank, it becomes a model level variable, so ValidationSummary(true) will only show the Garfield message, not the others.  Try a few.

# Property-Level Validation Errors

Oftentimes we want to put model errors at the top and field errors in-line.  The Html.ValidationMessageFor method does this for us.

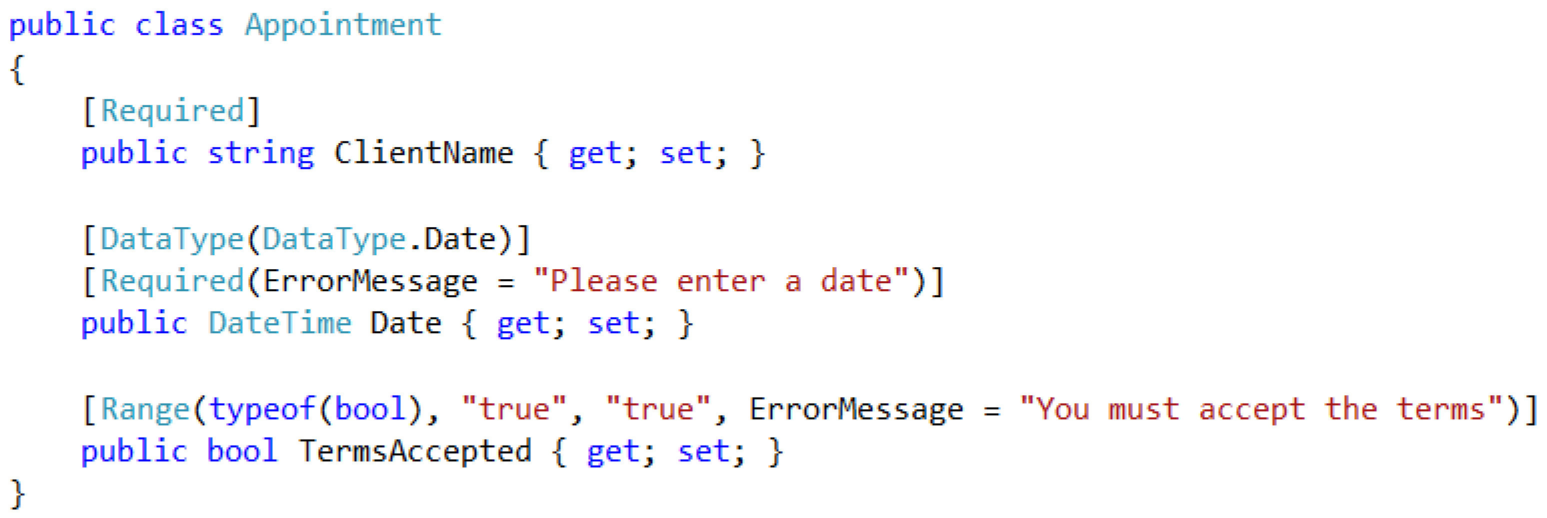


```
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>@Html.ValidationMessageFor(m => m.Date)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```

## Using Attributes

The DataAnnotations namespace has some property attributes we can use to validate common scenarios.  Let’s remove the controller code and use attributes instead.

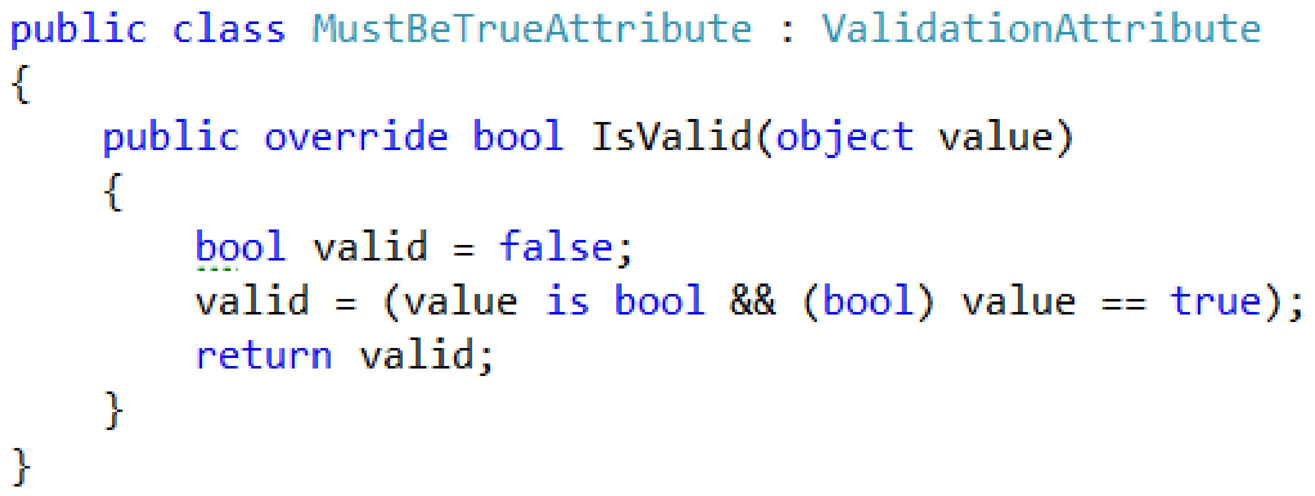
For a list of all attributes, see MSDN.



```
public class Appointment
{
    [Required]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [Range(typeof(bool), "true", "true", ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
}
```

## Rolling Your Own Attributes

We can roll our own attributes if we inherit from the ValidationAttribute class and override the IsValid() method.  Here is one that forces a Boolean to be true:



```
public class MustBeTrueAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        bool valid = false;
        valid = (value is bool && (bool) value == true);
        return valid;
    }
}
```

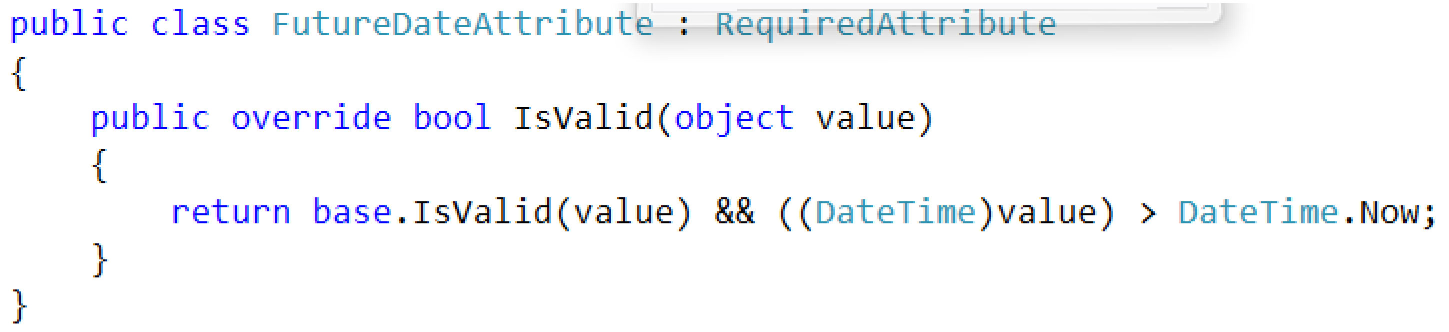
## Then, Attach to the Model

```
public class Appointment
{
    [Required]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
}
```

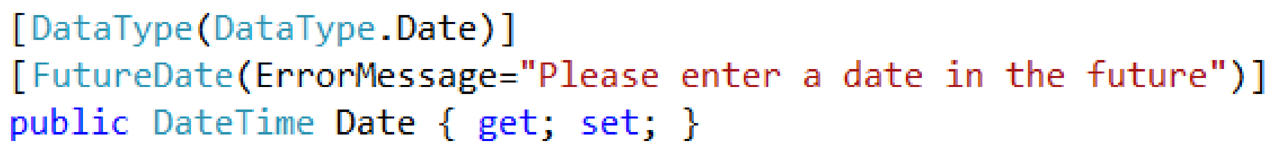
SOFTWARE GUILD

# Extending the Future Date

Let’s also make an attribute for a future date. Here, we will mix the base class IsValid with our own, so we can let the model binder do the type checking, and we can handle the future date check:

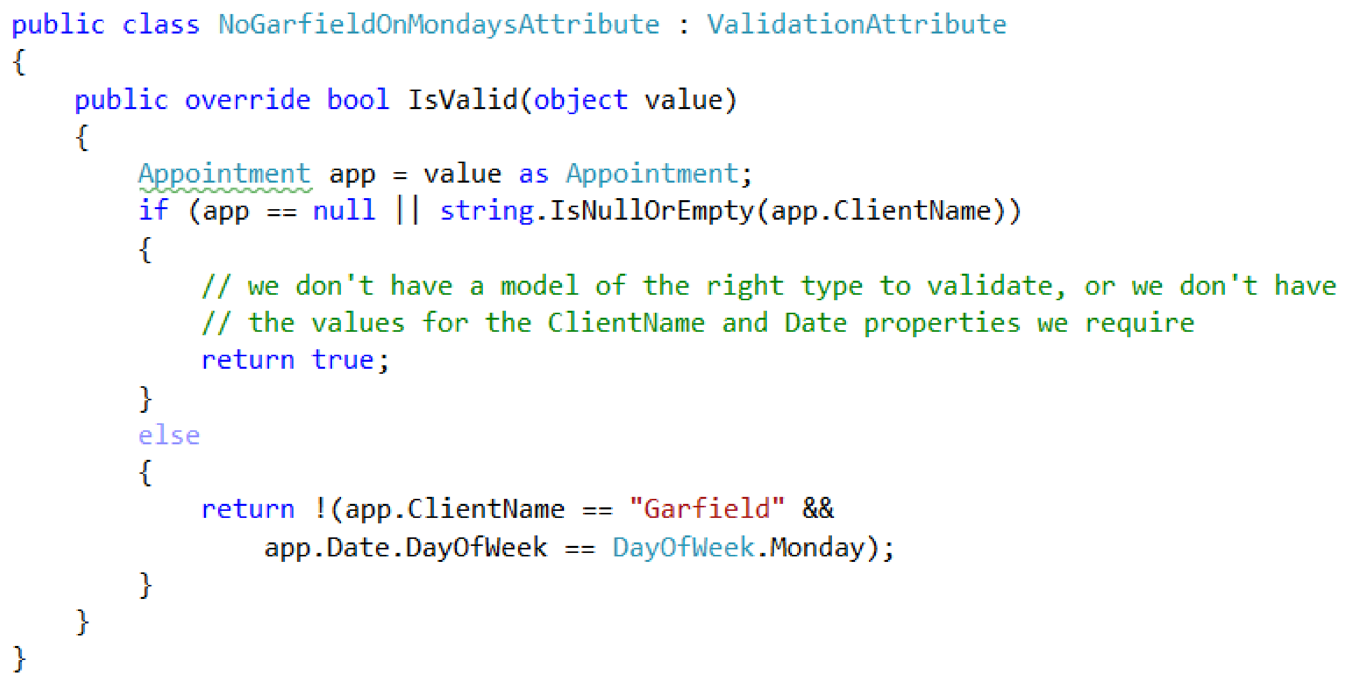


```
public class FutureDateAttribute : RequiredAttribute
{
    public override bool IsValid(object value)
    {
        return base.IsValid(value) && ((DateTime)value) > DateTime.Now;
    }
}
```

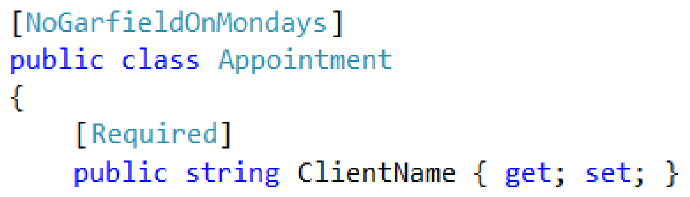


```
[DataType(DataType.Date)]
[FutureDate(ErrorMessage="Please enter a date in the future")]
public DateTime Date { get; set; }
```

# Extending a Model-Level Validation



```
public class NoGarfieldOnMondaysAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        Appointment app = value as Appointment;
        if (app == null || string.IsNullOrEmpty(app.ClientName))
        {
            // we don't have a model of the right type to validate, or we don't have
            // the values for the ClientName and Date properties we require
            return true;
        }
        else
        {
            return !(app.ClientName == "Garfield" &&
                app.Date.DayOfWeek == DayOfWeek.Monday);
        }
    }
}
```



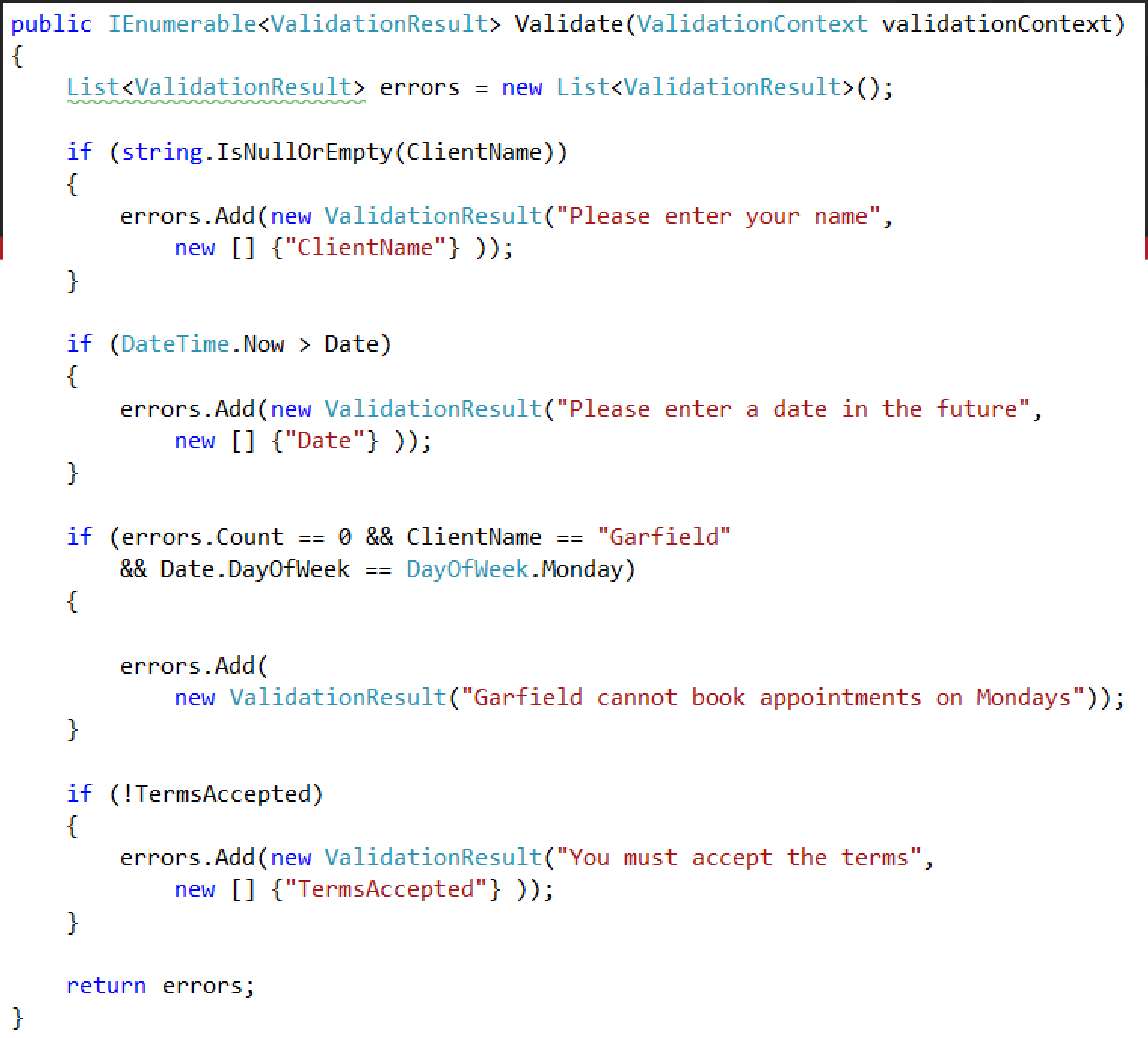
```
[NoGarfieldOnMondays]
public class Appointment
{
    [Required]
    public string ClientName { get; set; }
```

## Self-Validating Models

So we can validate in the controller and roll our own attributes; the last technique is self-validating models.  For this technique, we implement the IValidatableObject interface, which puts a method in our class that we need to return a list of all of the validation errors to the caller.

```
public class Appointment : IValidatableObject
```

This technique lets you do whatever you want, but it’s also the most verbose (amount of code required).



```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    List<ValidationResult> errors = new List<ValidationResult>();

    if (string.IsNullOrEmpty(ClientName))
    {
        errors.Add(new ValidationResult("Please enter your name",
            new [] {"ClientName"} ));
    }

    if (DateTime.Now > Date)
    {
        errors.Add(new ValidationResult("Please enter a date in the future",
            new [] {"Date"} ));
    }

    if (errors.Count == 0 && ClientName == "Garfield"
        && Date.DayOfWeek == DayOfWeek.Monday)
    {

        errors.Add(
            new ValidationResult("Garfield cannot book appointments on Mondays"));
    }

    if (!TermsAccepted)
    {
        errors.Add(new ValidationResult("You must accept the terms",
            new [] {"TermsAccepted"} ));
    }

    return errors;
}
```

## Gut Check

- What do we need to check in our controller to know if the validation was successful?
  - Check ModelState.IsValid
  - If not, return original view with model to edit

## Gut Check (2)

- How do we show all of the validation errors at once?
  - Pick an overload of @Html.ValidationSummary()
- How do we show a validation message for a specific control?
  - @Html.ValidationMessageFor(m => m.Property)

## Conclusion

There are a lot of different ways to validate your models.  I find the IValidateableObject to be the cleanest but most verbose.  Rolling your own attributes is great for reuse… and validating directly in the controller is kind of messy.