
FORGING AHEAD
with
.NET
MVC

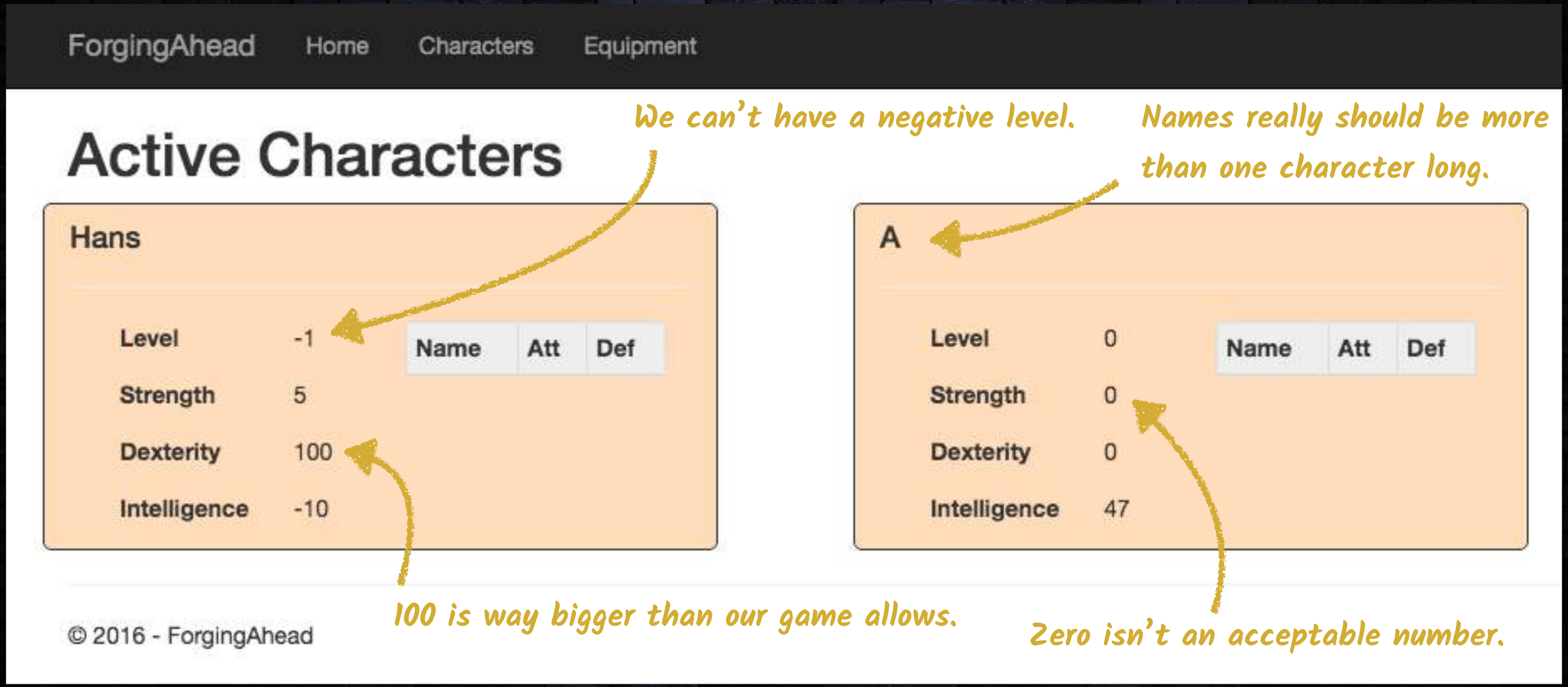
Level 5 – Section 1

Validating Our Input

Validation With DataAnnotations

Validating Input

We're seeing problems in our data — let's fix them.



DataAnnotations **Allow Us to Define Rules**

These rules can be used by front-end, back-end, and our database for validation.

Model With DataAnnotations

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

Front-end: Tag Helpers

```
<input id="Name" type="Text"
  data-val-required="…"/>
```

Back-end: ModelState

```
if(string.IsNullOrEmpty(Name))
    ModelState.AddModelError(…)
```

Data: Database Definition

```
Column Name VarChar not null
```

Having the Required attribute will effectively implement all of these.

Referencing DataAnnotations

Models/Character.cs

CS

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

public class Character
{
    public string Name { get; set; }
    [Display(Name = "Is Active")]
    public bool IsActive { get; set; }
    public int Level { get; set; }
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Intelligence { get; set; }
}
```

To use DataAnnotations, we'll need to reference it through a using directive.

Preventing Duplicate Names

The attribute Key will cause database validation to fail if a record already has the same value.

Models/Character.cs

CS

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

public class Character
{
    [Key]
    public string Name { get; set; }
    [Display(Name = "Is Active")]
    public bool IsActive { get; set; }
    public int Level { get; set; }
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Intelligence { get; set; }
}
```

Prevents multiple characters from having the same name in the database.

Don’t Allow Properties to Be Null

The attribute Required will cause validation to fail if the property is null.

Models/Character.cs

CS

```
…
public class Character
{
    [Key]
    [Required]
    public string Name { get; set; }
    [Required]
    [Display(Name = "Is Active")]
    public bool IsActive { get; set; }
    [Required]
    public int Level { get; set; }
    [Required]
    public int Strength { get; set; }
    [Required]
    public int Dexterity { get; set; }
    …
}
```

Makes the property non-nullable

Ensuring Names Are More Than One Letter

The attribute MinLength will cause validation to fail if the property is too short.

Models/Character.cs

CS

```
…
public class Character
{
  [Key]
  [Required]
  [MinLength(3)]
  public string Name { get; set; }
  [Required]
  public bool IsActive { get; set; }
  [Required]
  public int Level { get; set; }
  [Required]
  public int Strength { get; set; }
  [Required]
  public int Dexterity { get; set; }
  …
}
```

MinLength(3) makes sure our Name is at least three characters long.

Ensuring Input Is Not Too Big or Too Small

The attribute Range will cause validation to fail if the property is outside the specified range.

Models/Character.cs

CS

```
…
public class Character
{
  [Key]
  [Required]
  [MinLength(3)]
  public string Name { get; set; }
  [Required]
  public bool IsActive { get; set; }
  [Required]
  [Range(1, 20)]
  public int Level { get; set; }
  [Required]
  [Range(1, 18)]
  public int Strength { get; set; }
  …
}
```

Range(1, 20) makes sure Level is no lower than 1 and no bigger than 20.

DataAnnotations **Are Set Up and Ready**

At this point, we used several attributes to establish our validation rules.

- Key - Characters are required to have a unique name
- MinLength - Character names must be three or more characters long
- Range - Stats must fall between between a specified range

Level 5 – Section 2

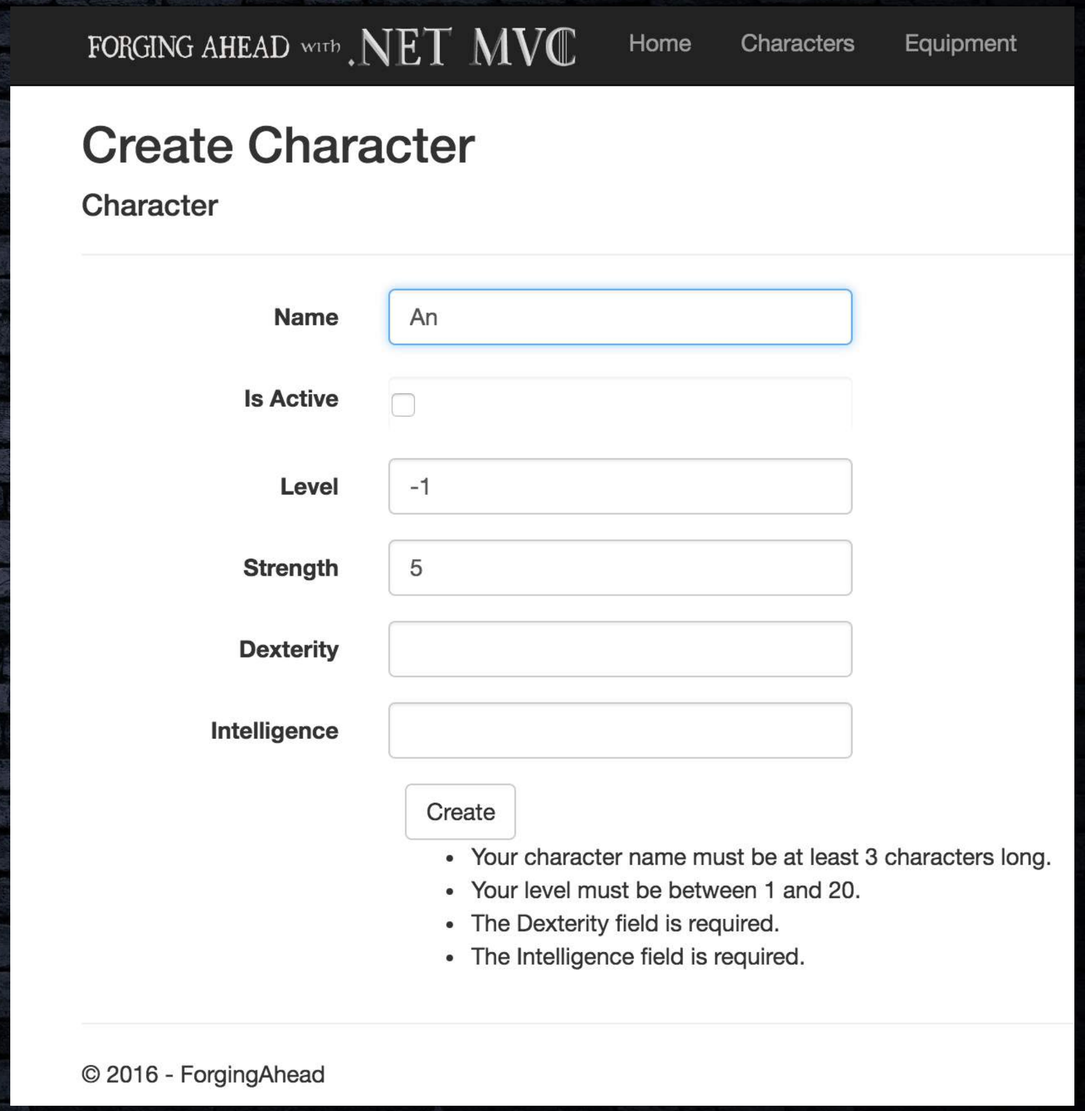
Validating Our Input

Front-end Validation

Setting Up Front-end Validation

The first layer of validation is client-side, providing the best user experience.

- It's the fastest form of validation
- Doesn't add any unnecessary load to our server
- Provides user-friendly feedback



You need both front- AND back-end validation. Front-end validation performs way better than back-end, but can be broken and bypassed.

Adding Front-end Validation JavaScript

Let's add jquery.validation.js **and** jquery.validate.unobtrusive.js **references to our layout's scripts.**

Views/Shared/_Layout.cshtml

CSHTML

```cshtml
…
  </div>

  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
  <script src="~/lib/jquery-validation-unobtrusive/
     jquery.validate.unobtrusive.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>

  @RenderSection("scripts", required: false)
</body>
</html>
```

These scripts are included in most ASP.NET project templates.

Setting Up Our View to Display Validation Errors

Creating a div with the attribute asp-validation-summary="All" will display validation errors.

Views/Character/Create.cshtml

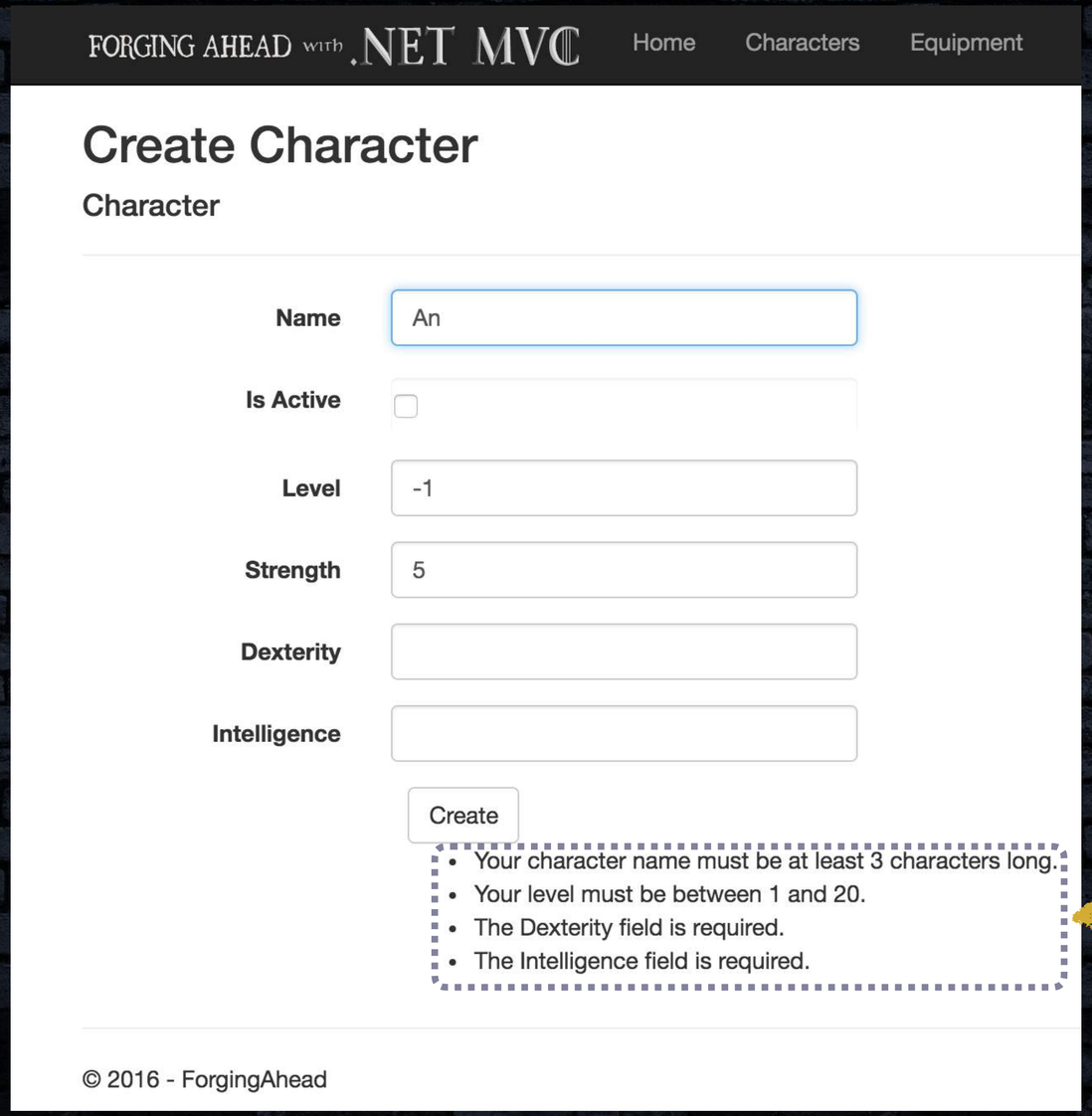
CSHTML

```
@model ForgingAhead.Models.Character
<h2>Create Character</h2>

<form asp-action="Create" asp-controller="Character">
  <div class="form-horizontal">
  …
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Create" class="btn btn-default"/>
        <div asp-validation-summary="All"></div>
      </div>
    </div>
  </div>
</form>
```

The div will have all validation errors added in it in an unordered list on submit.

Front-end Validation Is Set Up

Now most of our validation rules are enforced by our view.



Summary of all our validation errors.


FORGING AHEAD
with
.NET
MVC

Level 5 – Section 3

Validating Our Input

ModelState **Validation**

Setting Up ModelState Validation

We need to set up our back-end to validate input as well because we can't trust our front-end.

- The user might disable JavaScript, which our front-end validation depends on.
- Someone might directly submit data to our back-end, bypassing our front-end.
- Our front-end could have bugs that allow bad data through.

ModelState is a property that tracks values submitted to the server. In addition to storing the name and value of each field, it also tracks the associated validation errors.

Checking That Our Input Was Valid

We can check ModelState.IsValid, as it will be false if any of our validation rules are broken.

Views/Character/Create.cshtml

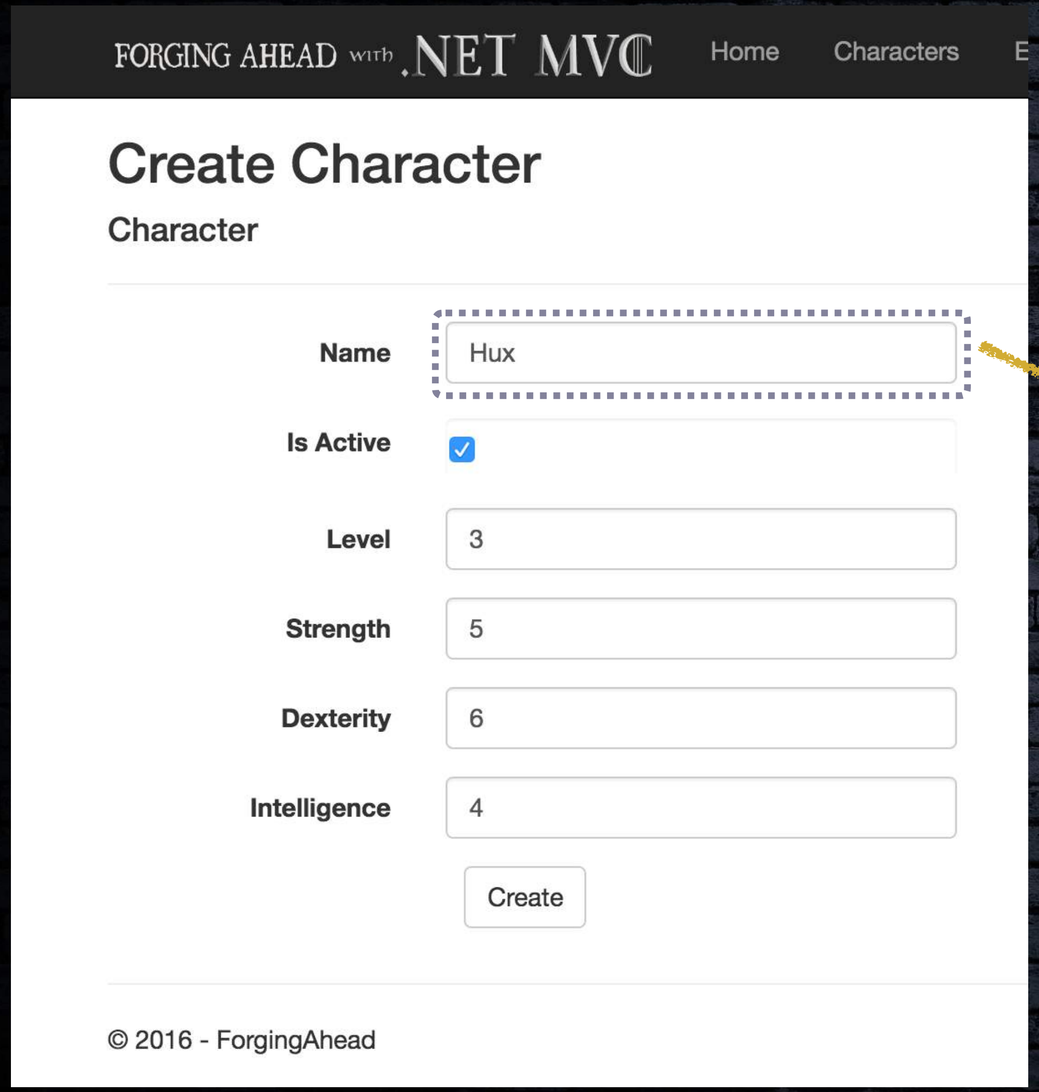
CSHTML

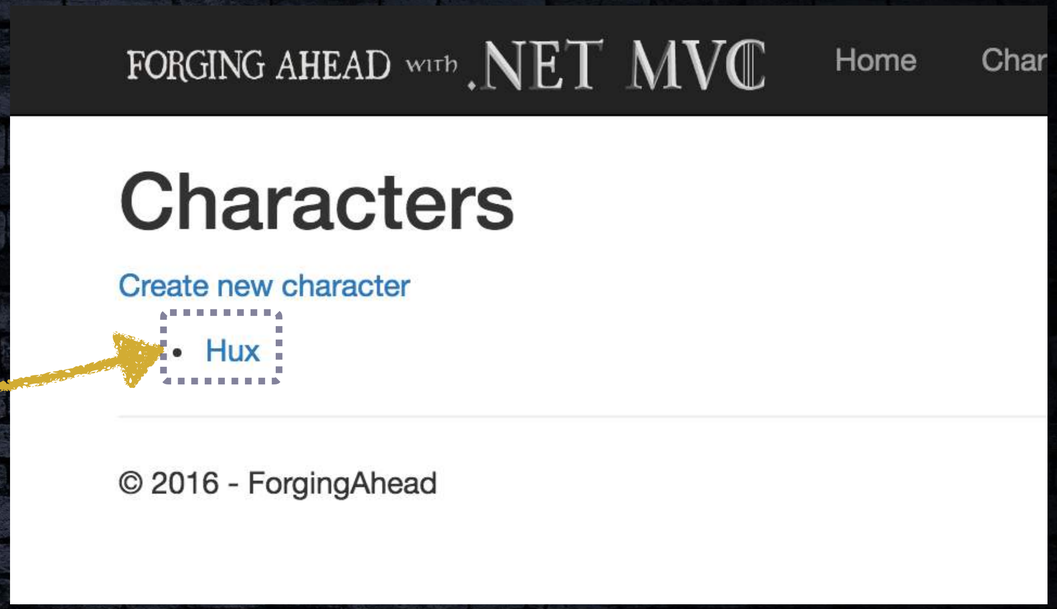
```
…
public class CharacterController : Controller
{
…
  public IActionResult Create(Character character)
  {
      if(!ModelState.IsValid)
        return View(character);
      _context.Characters.Add(character);
      _context.SaveChanges();

    return RedirectToAction("Index");
  }
…
}
```

If our Model isn't valid, we should return our view with the provided input. This will cause asp-validation-summary to display our errors.

Test: Entering the Same Name Twice

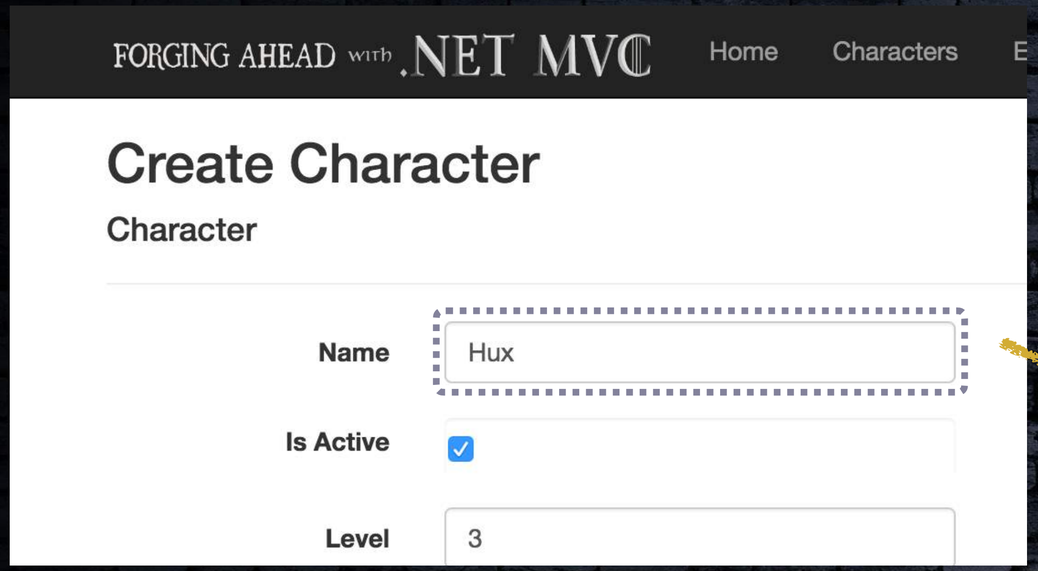
Entering a character with no validation errors works fine and adds the character to our list.

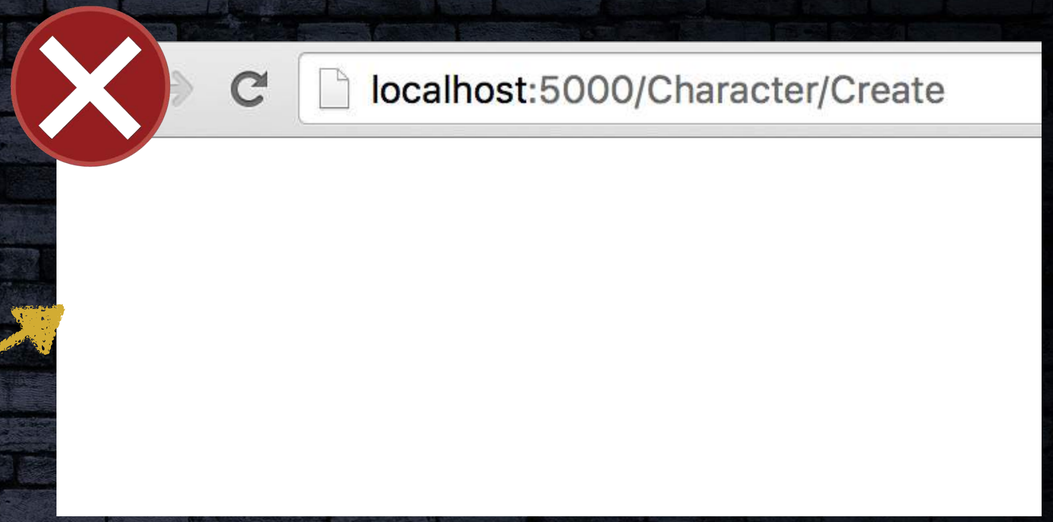




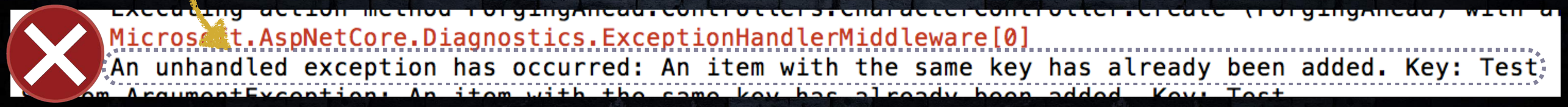
Using the Same Name Twice Is Broken

If we enter the same name a second time, we are just bounced to a blank page... Why?





If we look at the logs in our console, our Key attribute worked, but it didn't fail until it tried to save to the database.



Throwing a Validation Error on Duplicate Key

Our duplicate record is in the database, not our application, so we need to check manually.

Views/Character/Create.cshtml

CSHTML

```
…
  public IActionResult Create(Character character)
  {
      if(_context.Characters.Any(e => e.Name == character.Name))
        ModelState.AddModelError("Name", "Name is already in use.");
      if(!ModelState.IsValid)
        return View(character);
      _context.Characters.Add(character);
      _context.SaveChanges();

    return RedirectToAction("Index");
  }
…
```

We can add an error to our ModelState using the AddModelError method.

Adding a Model Error to our ModelState

To add a model error to our ModelState, we need to specify the field and error message.

Views/Character/Create.cshtml

CSHTML

```
…
  public IActionResult Create(Character character)
  {
      if(_context.Characters.Any(e => e.Name == character.Name))
        ModelState.AddModelError("Name", "Name is already in use.");
      if(!ModelState.IsValid)
        return View(character);
      _context.Characters.Add(character);
      _context.SaveChanges();

    return RedirectToAction("Index");
  }
…
```

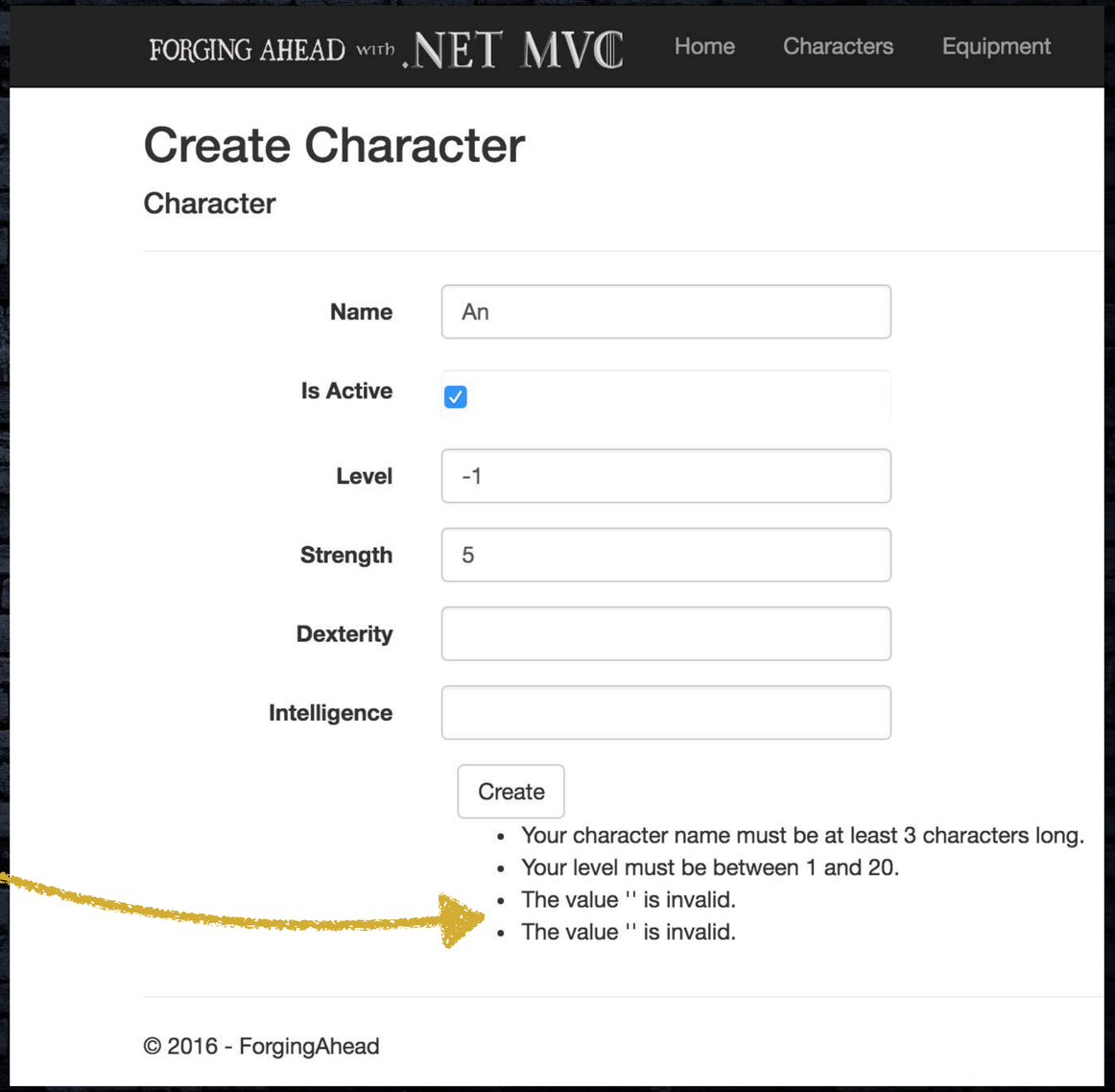
Field name

Error message

Back-end Validation Is Now Working...

...But some of the validation errors aren't exactly clear on what's wrong.

If our page had several dozen fields and we saw the message "The value '' is invalid"... how would we know which field isn't working?



Setting an Error Message

Adding ErrorMessage after a DataAnnotation allows us to specify the response to the user.

Models/Character.cs

CS

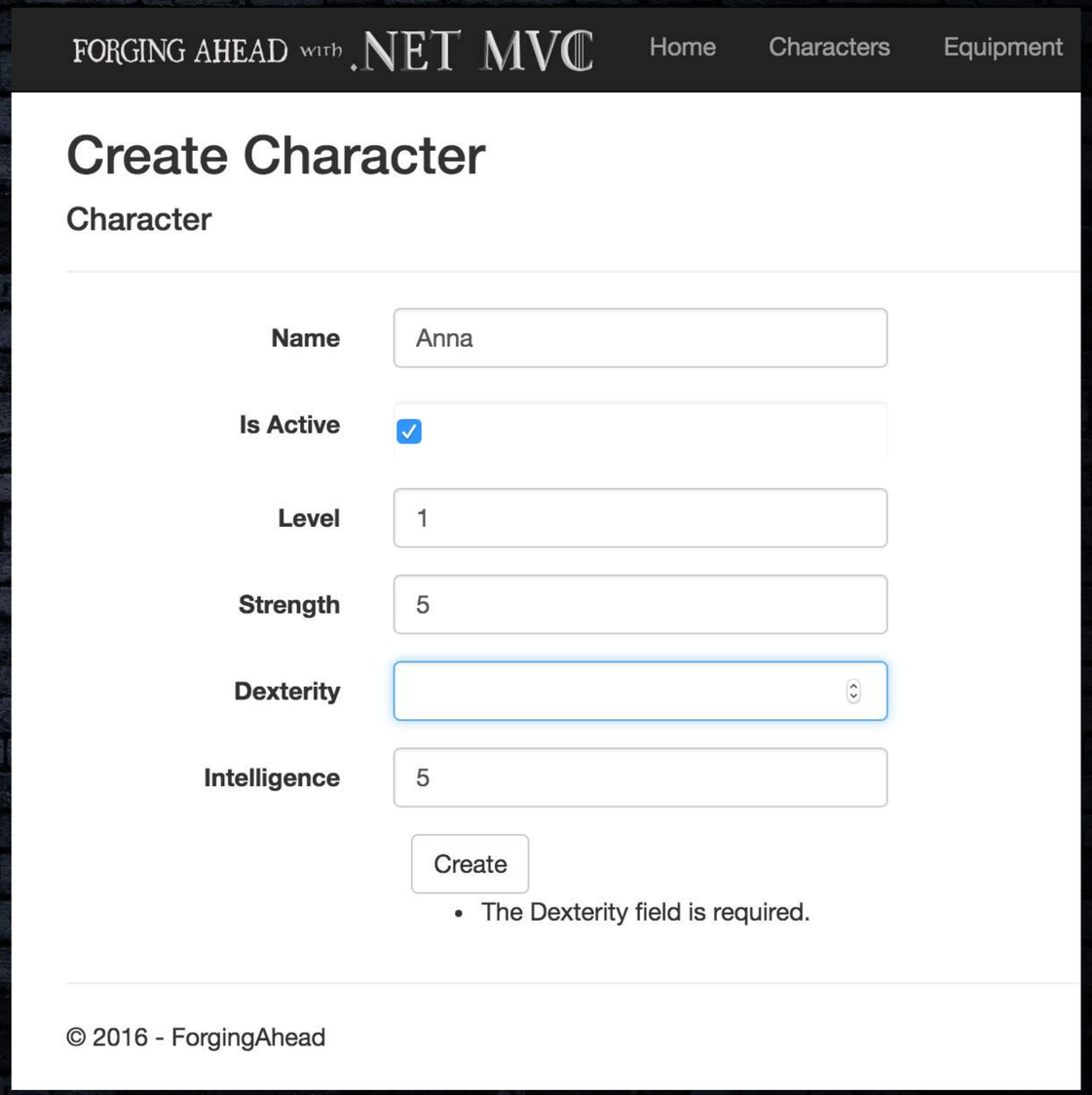
```
…
public class Character
{
  …
  [Required]
  [Range(1, 20)]
  public int Level { get; set; }
  [Required]
  [Range(1, 18)]
  public int Strength { get; set; }
  [Required(ErrorMessage = "The Dexterity field is required.")]
  [Range(1, 18)]
  public int Dexterity { get; set; }
  …
}
```

It's best to provide user-friendly error messages any time the default ones are unclear.

Now Our Input Is Properly Validated!

Not only are we validating on both the front-end and back-end, but it's user friendly too.

But wait — what about the database? It's already done! EntityFramework can understand DataAnnotations and takes care of it for us.




FORGING AHEAD
with
.NET
MVC