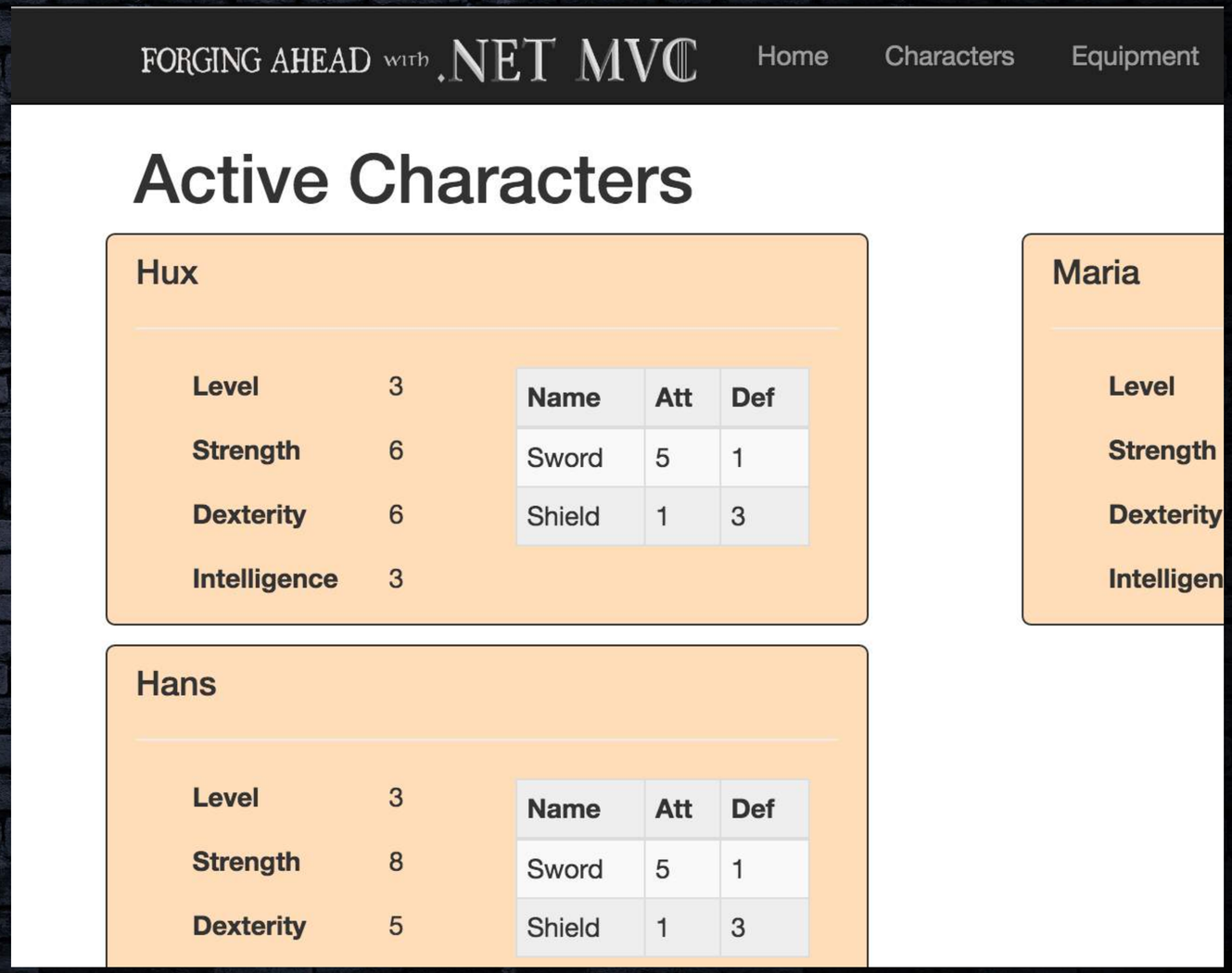
FORGING AHEAD
with
.NET
MVC

Expanding Our Idea

Our tabletop players want a more complete character sheet application.

- Store characters and equipment
- Reuse characters between games
- Reuse equipment between games
- Take steps to keep our data clean



ASP.NET Core vs. ASP.NET Framework

This course uses ASP.NET Core for its cross-platform support. Some content won't work in Framework.

All items on this list are ASP.NET Core-specific items and will not work in ASP.NET Framework.

- Any **using directives** or references with "core" in their name
- **Startup.cs** file

Level 1 – Section 1

Data Annotations

Making Our Models Smarter

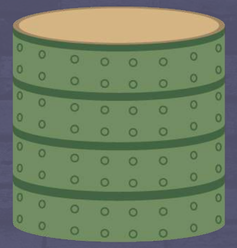
Options for Storing Data Long Term

Our data is relational, so we'll be better served using a database.

File System

- *Less system overhead for smaller datasets*
- *Less configuration/setup overhead*

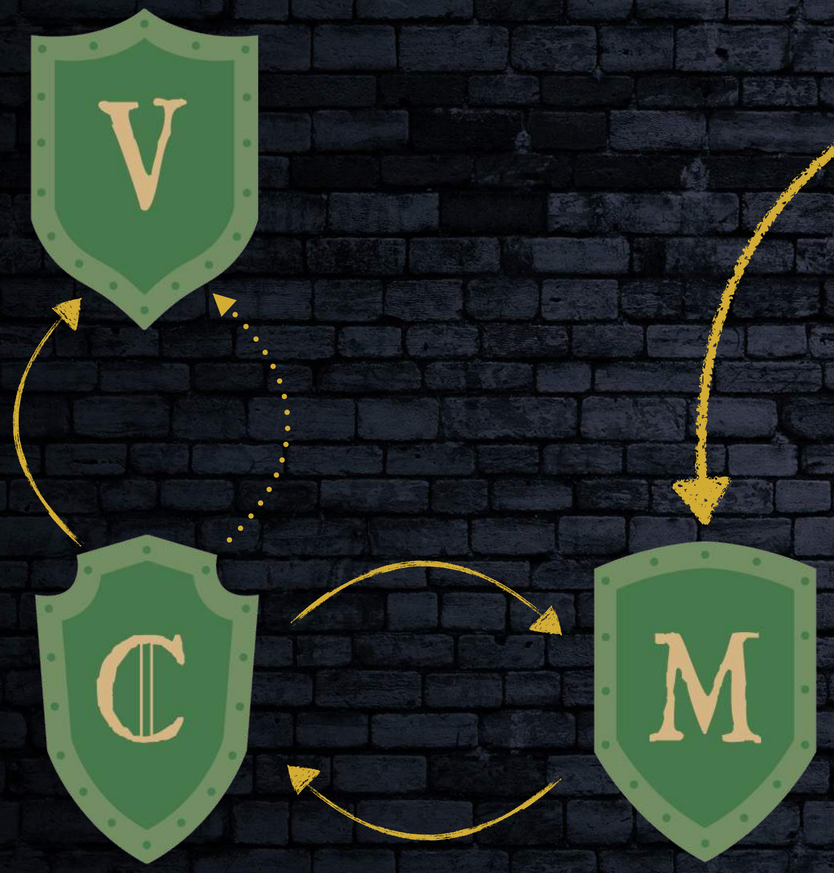
Databases



- *Optimized for relational data*
- *Less issues with multiple active users*
- *"Bad data" is easier to mitigate*

Storing Our Data

As we previously covered, all of our data and business logic go in our model.



We'll teach our application our data structure in our models.

Creating Our Character Model

Create our Character.cs class in our Models folder.

Models/Character.cs CS

```
namespace ForgingAhead.Models
{
  public class Character
  {
    public string Name { get; set; }
    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; }
  }
}
```

Creating Our Database Context File

Create** ApplicationDbContext.cs **in our** Models **folder.

To read from and write to our database, we'll need to:

- Reference **EntityFramework**
- Add a property for our characters
- Create a **CharacterController** that loads the **ApplicationDbContext**
- Create a character through the context in the controller
- Save the new character in the database

Setting Up Our ApplicationDbContext Class

EntityFramework is what we'll use to access our database.

Models/ApplicationDbContext.cs

CS

```
using System;
using Microsoft.EntityFrameworkCore;

namespace ForgingAhead.Models
{
  public class ApplicationDbContext : DbContext
  {

  }
}
```

Add our using directive to Entity Framework.

Inherit DbContext

DbContext teaches EntityFramework about our database through a collection of DbSets.

Adding Our DbSets to Define Our Tables

DbSet is a collection similar to a List that represents an individual database table or view.

Models/ApplicationDbContext.cs

CS

```
using System;
using Microsoft.EntityFrameworkCore;

namespace ForgingAhead.Models
{
  public class ApplicationDbContext : DbContext
  {
    public DbSet<Character> Characters { get; set; }
  }
}
```

Create a DbSet for Character. We will access our character data through this property.

DbSet **Naming Convention**

Typically, the name of your DbSet will be the plural form of the class of your DbSet.

Models/ApplicationDbContext.cs

CS

```
using System;
using Microsoft.EntityFrameworkCore;

namespace ForgingAhead.Models
{
  public class ApplicationDbContext : DbContext
  {
    public DbSet<Character> Characters { get; set; }
  }
}
```

Singular *Plural*

Creating Our Character Controller

We'll do a lot with the Character object, so we should create a controller for it.

Controllers\CharacterController.cs CS

```
using Microsoft.AspNetCore.Mvc;

namespace ForgingAhead.Controllers
{
    public class CharacterController : Controller
    {

    }
}
```

Create CharacterController.cs in our Controllers folder.

Adding Our Using Directives

We will need to add some using directives we're going to need in our CharacterController.

Controllers\CharacterController.cs

CS

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

using ForgingAhead.Models;

namespace ForgingAhead.Controllers
{
    public class CharacterController : Controller
    {

    }
}
```

We will need functionality from these three classes.

Creating a private readonly **Context Variable**

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;
}
…
```

We'll add a new variable for our ApplicationDbContext class named _context.

private **Restricts Access to Current Scope**

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;
}
…
```

We're making this variable private, as we don't want anything accessing it outside our CharacterController.

readonly **Prevents Changing the Variable**

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;
}
…
```

We don't want to be able to change this variable, so we'll make it readonly.

private readonly **Naming Conventions**

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;
}
…
```

When a variable is readonly, we typically precede that variable with an underscore.

Constructor Injection

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context)
    {
        _context = context;
    }
}
…
```

Here, we'll set up what's known as constructor injection, which allows us to inject ApplicationDbContext into our controller.

Injecting ApplicationDbContext

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context)
    {
        _context = context;
    }
}
…
```

Our application will inject ApplicationDbContext into the constructor parameter.

ASP.NET is set up to use dependency injection, which is a design pattern that allows for easier testability and adaptability through loosely coupled dependencies.

Dependency Injection Is Built Into ASP.NET!

Dependency injection is a design pattern to achieve Inversion of Control.

CharacterController.cs

```
new ApplicationDbContext();
```

ApplicationDbContext.cs

Without Inversion of Control, CharacterController has to call its dependency ApplicationDbContext directly, tightly coupling the two.

Inversion of Control

Inversion of Control allows us to loosen the coupling of dependencies.

ServicesCollection

CharacterController.cs

```
CharacterController(ApplicationDbContext context)
```

ApplicationDbContext.cs

With Inversion of Control, instead of CharacterController calling its dependency, we have the ServicesCollection that injects the dependency.

Making Our Injected Class Accessible

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context)
    {
        _context = context;
    }
}
…
```

We then set our private ApplicationDbContext to the injected instance to give us readonly access to it throughout our controller.

Creating Our Create Method

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context) {…}

    public IActionResult Create(Character character)
    {

    }
}
…
```

Create a standard action method Create and give it the Character object as a parameter.

Adding an Object to Our Characters Collection

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context) {…}

    public IActionResult Create(Character character)
    {
        _context.Characters.Add(character);
    }
}
…
```

To add our new character to our database, we just use the .Add() method exactly like we would with a List object.

Committing Changes to Our Database

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context) {…}

    public IActionResult Create(Character character)
    {
        _context.Characters.Add(character);
        _context.SaveChanges();
    }
}
…
```

In order to actually push our changes to the database, you need to call the SaveChanges() method.

You can make multiple changes before calling SaveChanges() to commit all of those changes at the same time.

Committing Changes to Our Database

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context) {…}

    public IActionResult Create(Character character)
    {
        _context.Characters.Add(character);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
}
…
```

To prevent the user from accidentally double submitting, let's redirect to our Index action.

Results of Calling CharacterController.Create

Calling our Create **method is going to do a lot more than just create a new** Character.

- If the **database** doesn't exist yet, then it will be created.
- If our **Characters table** doesn't exist yet, it will be created.
- A Character **record** will be created in the **Characters table.**

When we call our Create method with a valid Character object, it'd result in a database record similar to this.

Characters

	Name	IsActive	Level	Strength	Dexterity	Intelligence
1	Hans	0	1	5	5	5

Entity Framework is an object-relational mapper (ORM). ORMs handle all the database stuff so we can focus on our code.

Level 1 – Section 2

Working With CRUD

Create, Read, Update, and Delete Database Records

Where Did ApplicationDBContext Come From?

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    public CharacterController(ApplicationDbContext context)
    {
        _context = context;
    }
}
…
```

ApplicationDbContext ***is injected using*** *ServicesCollection,* ***but we still need to add*** *ApplicationDbContext* ***to our*** *ServicesCollection.*

Configuring Our Project to Use Our ORM

Our ORM knows how to handle the data, but our project needs to know how to use our ORM.

- ForgingAhead
  - Controllers
  - Models
  - Views
  - project.json
  - Startup.cs
  - ...

To wire up our project to use our ORM, we need to open our Startup.cs file.

This is how we configure our project to use our ORM in ASP.NET Core — instructions will vary for other versions of ASP.NET.

What Is Startup.cs?

Startup.cs is where we configure what our application will use, including MVC, Entity Framework, dependency injection, etc.

Startup.cs

CS

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace ForgingAhead
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath);
```

Adding Reference to EntityFramework

Startup.cs

CS

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace ForgingAhead
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
```

We'll want to reference EntityFramework so we have access to it in our Startup class.

ConfigureServices **Method**

Startup.cs

CS

```
…
namespace ForgingAhead
{
    public class Startup
    {
        public Startup(IHostingEnvironment env) {…}

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        public void Configure(…) {…}
    }
}
```

Our ConfigureServices method configures dependencies to be injected through dependency injection.

Adding EntityFramework to Our Services

Startup.cs

CS

```
…
namespace ForgingAhead
{
    public class Startup
    {
        public Startup(IHostingEnvironment env) {…}

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddEntityFramework();

            services.AddMvc();
        }

        public void Configure(…) {…}
    }
}
```

AddEntityFramework() will make Entity Framework available throughout our application.

Setting Up EntityFramework **With Our** DbContext

Startup.cs

CS

```
…
namespace ForgingAhead
{
    public class Startup
    {
        public Startup(IHostingEnvironment env) {…}

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddEntityFramework()
                .AddDbContext<Models.ApplicationDbContext>();

            services.AddMvc();
        }

        public void Configure(…) {…}
    }
}
```

AddDbContext ***tells Entity Framework where to find our*** *DbContext*, ***which defines what our data looks like.***

CRUD Methods We'll Want

Here are some things we know our players will want to be able to do in our application.

- ✓ Create a character
- Read all characters
- Read a specific character's details
- Read all "active" characters
- Update a character
- Delete a character

We've already implemented our character Create method.

Adding System.Linq to Our Using Directives

Controllers\CharacterController.cs

CS

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

using ForgingAhead.Models;

namespace ForgingAhead.Controllers
{
    public class CharacterController : Controller {…}
}
```

System.Linq gives us access to lambda expressions, which we'll use in our CRUD methods.

Creating Our Index Method

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult Create(Character character) {…}

    public IActionResult Index()
    {

    }
}
…
```

Our Index method effectively is our "Read All" characters functionality.

Pulling a Full Dataset From Our Database

Our Characters dbset is a collection like a List, but we need to convert it to a List for our view.

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult Create(Character character) {…}

    public IActionResult Index()
    {
        var model = _context.Characters.ToList();
        return View(model);
    }
}
…
```

ToList() converts a collection into a List collection.

Getting Active Characters Using Lambda

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult Index() {…}

    public IActionResult GetActive()
    {
        var model = _context.Characters.Where(e => e.IsActive).ToList();
        return View(model);
    }
}
…
```

To get only active characters, we'll use the Where method with a lambda expression and filter to only get records where IsActive is true.

How Lambda Expressions Work

Lambda allows us to effectively write foreach loops in a condensed form.

CS

Collection using Where **method and lambda expression**

```
…
var model = _context.Characters.Where(e => e.IsActive).ToList();
…
```

The e in our lambda names our variable to represent each record.

After the => determines which records are returned — in our case, any record where IsActive is true.

Collection using foreach **Loop**

```
var model = new List<Character>();
foreach (var e in _context.Characters)
{
    if(e.IsActive)
        model.Add(e);
}
```

Getting Just One Record From Our Database

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult GetActive() {…}

    public IActionResult Details(string name)
    {
        var model = _context.Characters.FirstOrDefault();
        return View(model);
    }
}
…
```

To get a specific character's details we will want to use the FirstOrDefault() method. This ensures we'll only get one character back instead of a collection.

Lambda for Comparing Values

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult GetActive() {…}

    public IActionResult Details(string name)
    {
        var model = _context.Characters.FirstOrDefault(e => e.Name == name);
        return View(model);
    }
}
…
```

To make sure we get the correct character back, we can use a lambda expression to only return a character with a matching Name.

Creating Our Update Method

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult Details() {…}

    public IActionResult Update(Character character)
    {
        _context.Entry(character).State = EntityState.Modified;
    }
}
…
```

To update a record, we can use Entry to locate and set our data, then set its State to Modified. This lets EntityFramework know we've changed the record.

Don’t Forget to SaveChanges

Controllers\CharacterController.cs

CS

```
…
public IActionResult Update(Character character)
{
    _context.Entry(character).State = EntityState.Modified;
    _context.SaveChanges();
}
…
```

We need to make sure to call SaveChanges() so our database is updated.

Redirecting to Index When We're Done

Controllers\CharacterController.cs

CS

```
…
public IActionResult Update(Character character)
{
    _context.Entry(character).State = EntityState.Modified;
    _context.SaveChanges();
    return RedirectToAction("Index");
}
…
```

Once we're all done, we should redirect to our Index action to prevent accidental submissions.

Deletion Confirm You Found a Record

Controllers\CharacterController.cs

CS

```
…
public IActionResult Delete(string name)
{
    var original = _context.Characters.FirstOrDefault(e => e.Name == name);
    if(original != null)
    {
        _context.Characters.Remove(original);
        _context.SaveChanges();
    }
    return RedirectToAction("Index");
}
…
```

We need to make sure we find a record before we attempt to delete it.

CRUD Methods Are Implemented

We've now implemented all of our CRUD methods.

- Create a character
- Read all characters
- Read a specific character's details
- Read all "active" characters
- Update a character
- Delete a character

FORGING AHEAD
with
.NET
MVC

Level 2 – Section 1

Showing Off Our Data

Relating Data

Planning Out Our Data Structure

Before we can implement our data structure, we need to plan it out.

We know our players will be playing as characters.

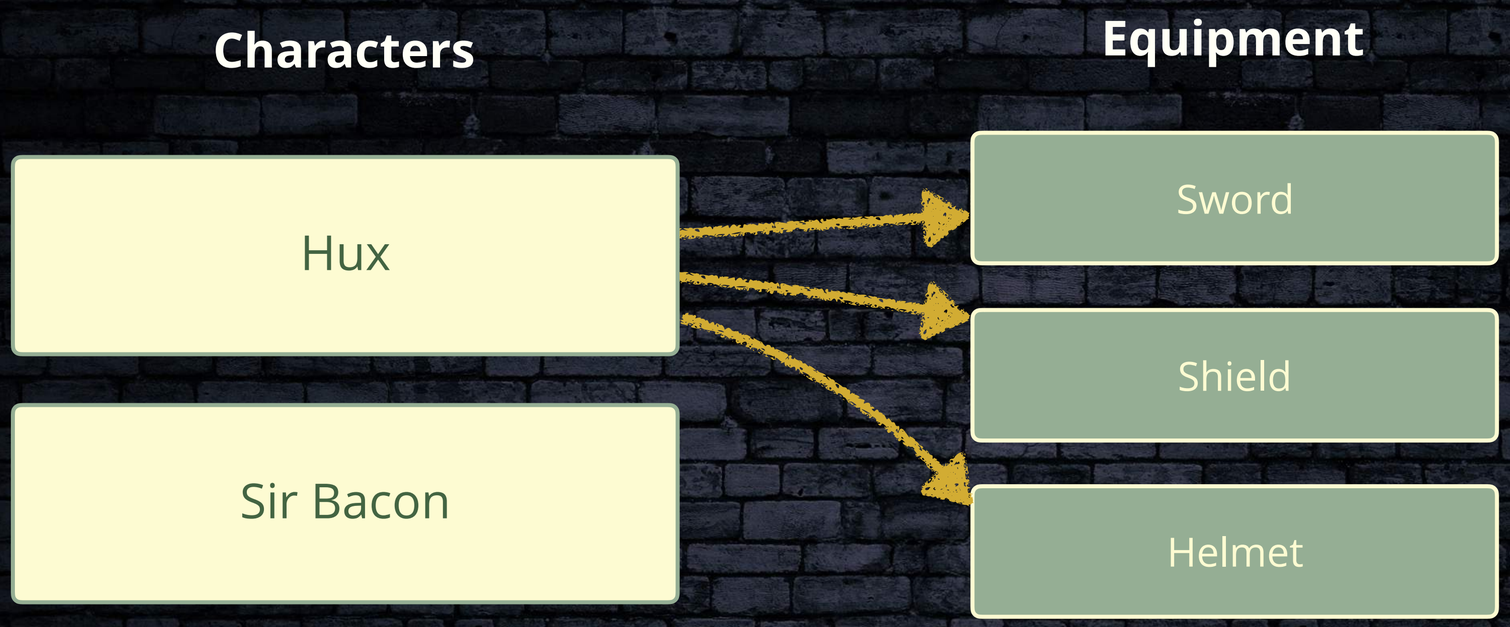
Characters

Hux

Sir Bacon

Characters Have Equipment

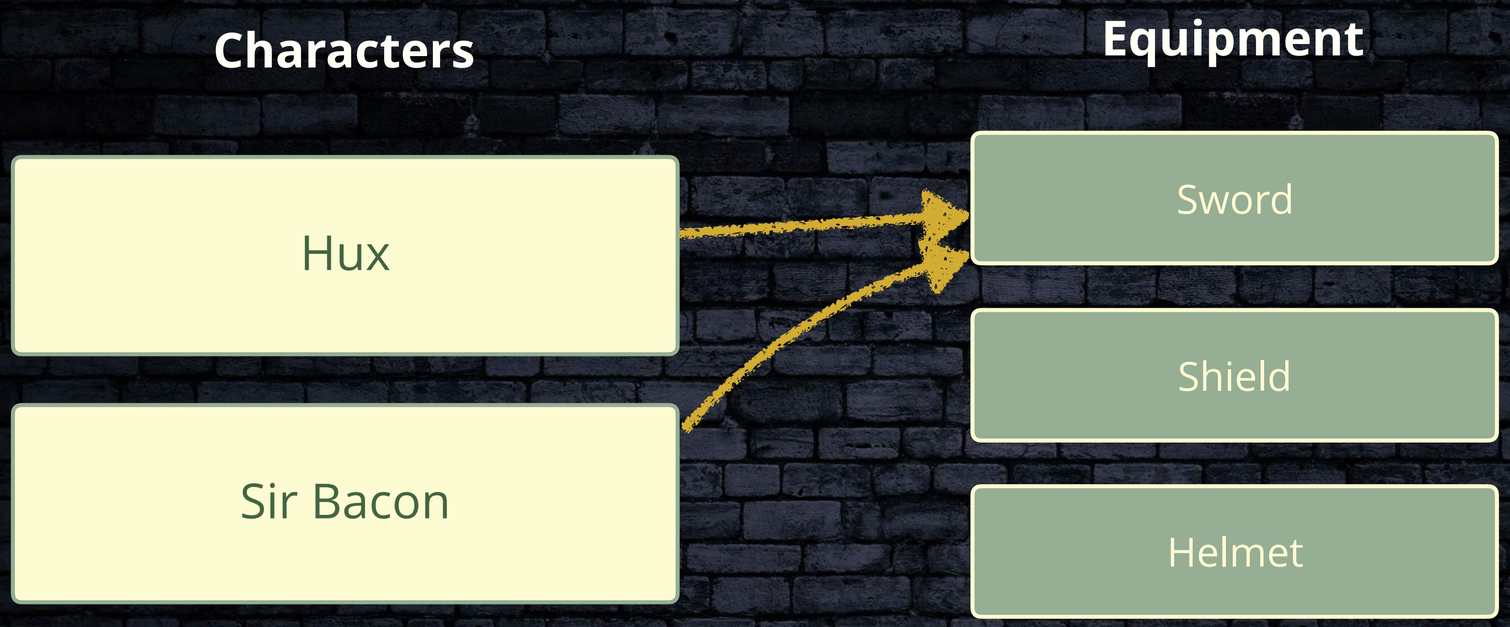
Each character can carry multiple pieces of equipment.



This implies a one-to-many relation between characters and equipment.

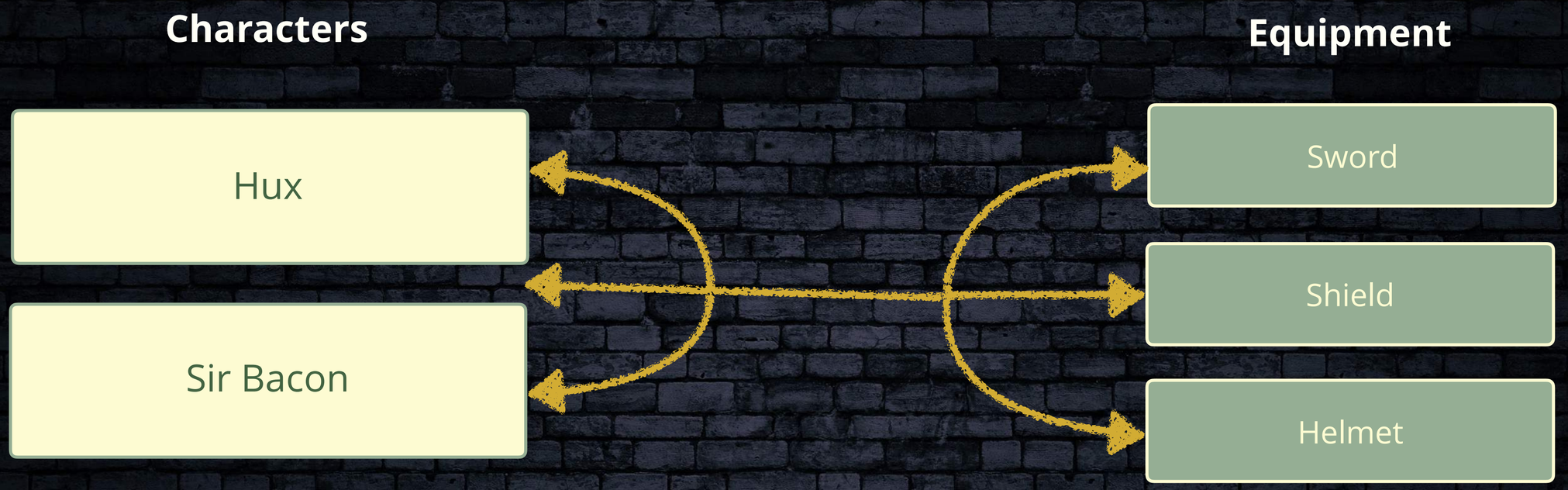
Equipment Isn't Unique

Multiple characters can also have the same type of equipment.



Many-to-many Relationship

This means characters and equipment have a many-to-many relationship.



How Databases Handle Many-to-many

Our database will need what's called a union table to accommodate this relationship.

Characters

Hux
Sir Bacon

Union Table

Hux : Sword
Hux : Shield
Sir Bacon : Shield

Equipment

Sword
Shield
Helmet

Object-relational mapping frameworks handle all this complicated database stuff so we don't have to, but you should be aware of what it's doing behind the scenes.

Creating Our Relationship in Code

Models/Character.cs

CS

```
using System.Collection.Generic;
using System.ComponentModel.DataAnnotations;

namespace ForgingAhead.Models
{
    public class Character
    {
        …
        public int Intelligence { get; set; }

        public List<Equipment> Equipment { get; set; }
    }
}
```

Making a List of Equipment in our Character class will create the relationship between Character and Equipment.

Don't forget you'll need a using directive for System.Collection.Generic to use List.

FORGING AHEAD
with
.NET
MVC

Level 2 – Section 2

Showing Off Our Data

Presenting Data in Views

We Need to Create Our Views

We've created our action methods to pass data to our views, so we just need to make the views.

- Character Create Page
- Character Index Page

Views

Character

Create.cshtml

Index.cshtml

Create our Create and Index views inside our Views/Character folder.

Creating Our Index.cshtml Page

./Views/Character/Index.cshtml

CSHTML

```
@model List<ForgingAhead.Models.Character>
<h1>
    Characters
</h1>
<ul>
    @foreach(var item in Model)
    {
      <li>
         @item.Name
      </li>
    }
</ul>
```

We'll add a link to our Create action after our heading.

Adding a Link to Our Create Action

./Views/Character/Index.cshtml

CSHTML

```
@model List<ForgingAhead.Models.Character>
<h1>
    Characters
</h1>
<p>
    <a asp-action="Create" asp-controller="Character"/>Create new character</a>
<p>
<ul>
    @foreach(var item in Model)
    {
      <li>
        @item.Name
      </li>
    }
</ul>
```

We can use asp-action and asp-controller to specify which controller and action to use.

The template will render this URL based on the two attributes we provided.

```
<a href="/Character/Create"/>Create new character</a>
```

Adding Parameters Through asp-route

asp-route allows us to specify parameters by adding the parameter name after a hyphen.

./Views/Character/Index.cshtml

CSHTML

The list of names is a good place for us to link to our Details action.

```
…
@foreach(var item in Model)
{
  <li>
    <a asp-action="Details" asp-controller="Character" asp-route-name="@item.Name"/>
        @item.Name
    </a>
  </li>
}
…
```

Adds a query string Name with item.Name's value

```
<a href="/Character/Details?Name=Hans">
```

If you have a character named Hans, our Tag Helpers would render this to the browser.

Creating Our Create.cshtml Page

./Views/Character/Create.cshtml

CSHTML

```
@model ForgingAhead.Models.Character
<h2>Create</h2>
<hr/>
<form asp-action="Create" asp-controller="Character">
    <div class="form-horizontal">
        <h4>Character</h4>
        <hr/>
        <div class="form-group">
            <div class="row">
                <label class="control-label col-md-2" asp-for="Name"></label>
                <div class="col-md-10">
                    <input class="form-control" asp-for="Name"/>
                </div>
            </div>
        </div>
…
```

Our Create page is using CSS classes to keep things looking nice — otherwise, most of this is the same HTML and Razor we've seen before.

asp-for **Tag Helper**

./Views/Character/Create.cshtml

CSHTML

```
…
<h4>Character</h4>
<hr/>
<div class="form-group">
  <div class="row">
    <label class="control-label col-md-2" asp-for="Name"></label>
    <div class="col-md-10">
      <input class="form-control" asp-for="Name"/>
    </div>
  </div>
</div>
…
```

asp-for ***will populate the value of our*** *label* ***based on the field's*** *Name.*

it will also set our input tag's *type, id,* ***and*** *name* ***attributes.***

The tag helpers would render the resulting HTML.

```
<label … for="Name">Name</label>
<div class="col-md-10">
  <input … type="text" id="Name" name="Name"/>
</div>
```

What If We Want a Different Label?

./Views/Character/Create.cshtml

CSHTML

IsActive ***doesn't look very good as one word...***

We can fix this with *DataAnnotations.*

```
…
<div class="form-group">
  <div class="row">
    <label class="control-label col-md-2" for="IsActive">IsActive</label>
    <div class="col-md-10">
      <input class="form-control" type="checkbox" id="IsActive" name="IsActive" />
    </div>
  </div>
</div>
…
```

Setting Display Through DataAnnotations

Models/Character.cs

CS

Add a using directive for DataAnnotations

```
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; }
…
```

The attribute Display sets the display name of our property, which tells asp-for what you want the label to show.

Rendered Output With IsActive Changes

Setting the IsActive property's Display attribute made it so the label is now Is Active instead of IsActive.

```
…
<div class="form-group">
  <div class="row">
    <label class="control-label col-md-2" for="IsActive">Is Active</label>
    <div class="col-md-10">
      <input class="form-control" type="checkbox" id="IsActive" name="IsActive"/>
    </div>
  </div>
</div>
…
```

The for, id, and name attributes are still IsActive, which is what we want, so everything wires up properly.

You Can Now Add and View Data

We've added the views necessary for our Create **and** Index **actions to work.**

- Character Create Page
- Character Index Page

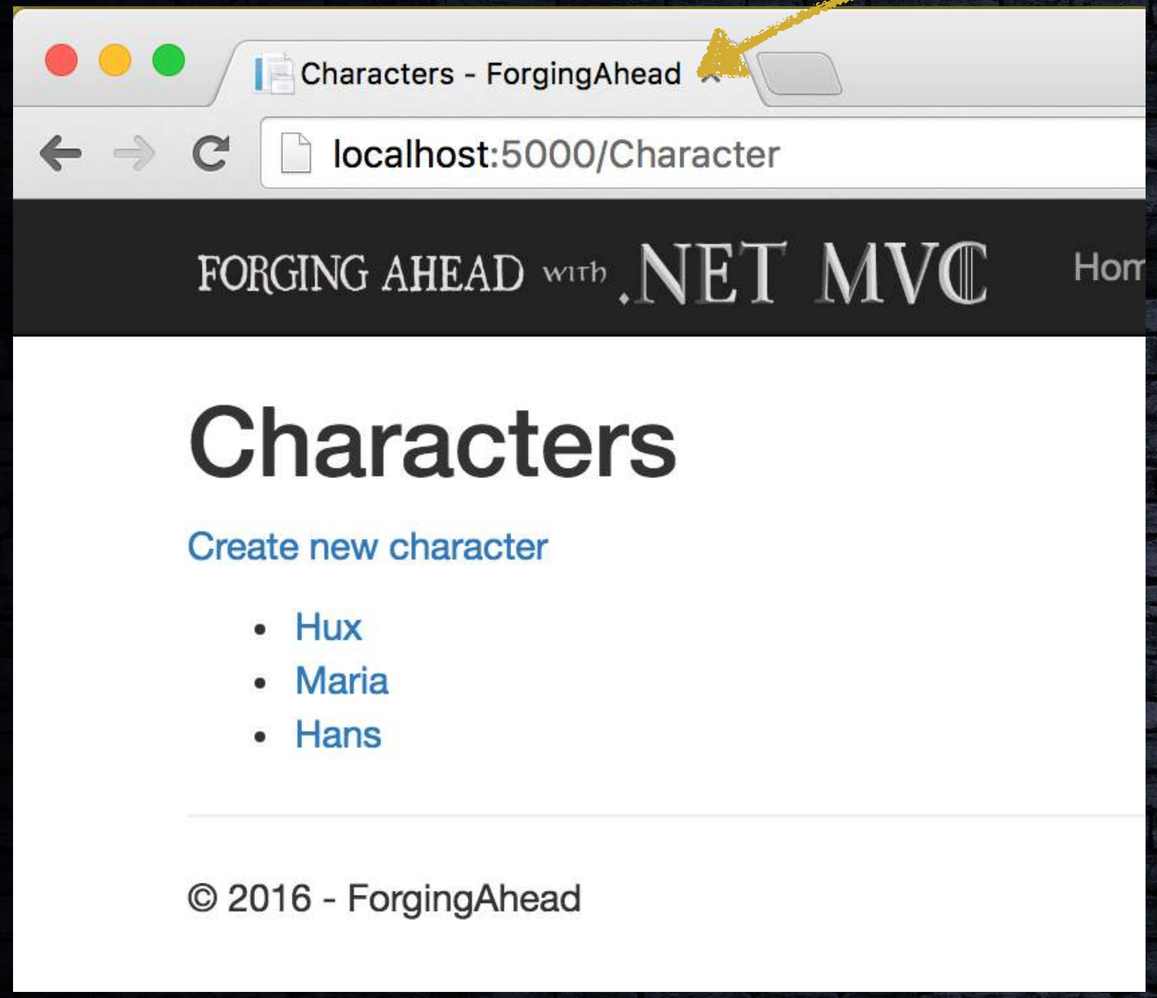
FORGING AHEAD
with
.NET
MVC

Level 3 – Section 1

Making Our Presentation Smarter

Layouts

We Want Our Title and Headers to Be Smarter



We want clear and concise titles to show up in both the title and search results.

Search Results Example

Characters - ForgingAhead
localhost:5000/Character
Lists all characters in ForgingAhead.

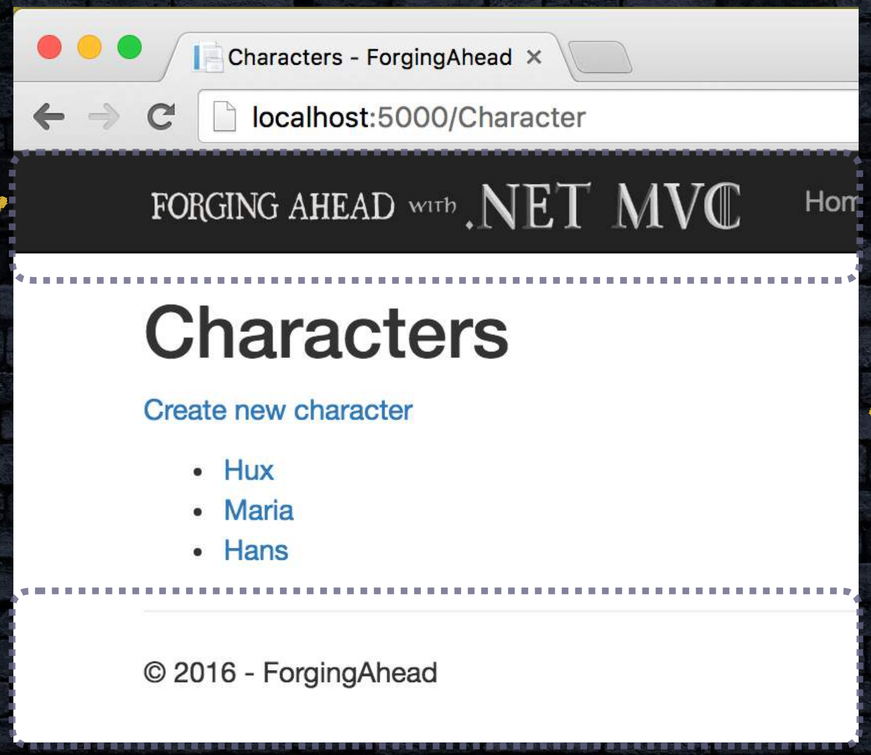
We want to specify our description data for our search results.

We want to handle this application-wide, but where could we set this?

Some Content Is the Same on All Pages

When viewing our application, you'll notice parts of the page don't change page-to-page.

Navigation is the same on all pages.

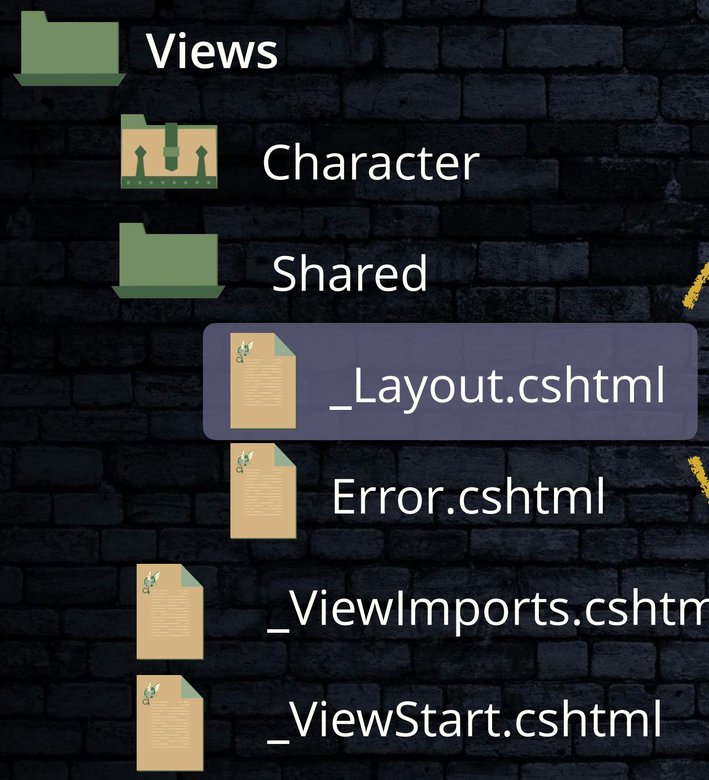


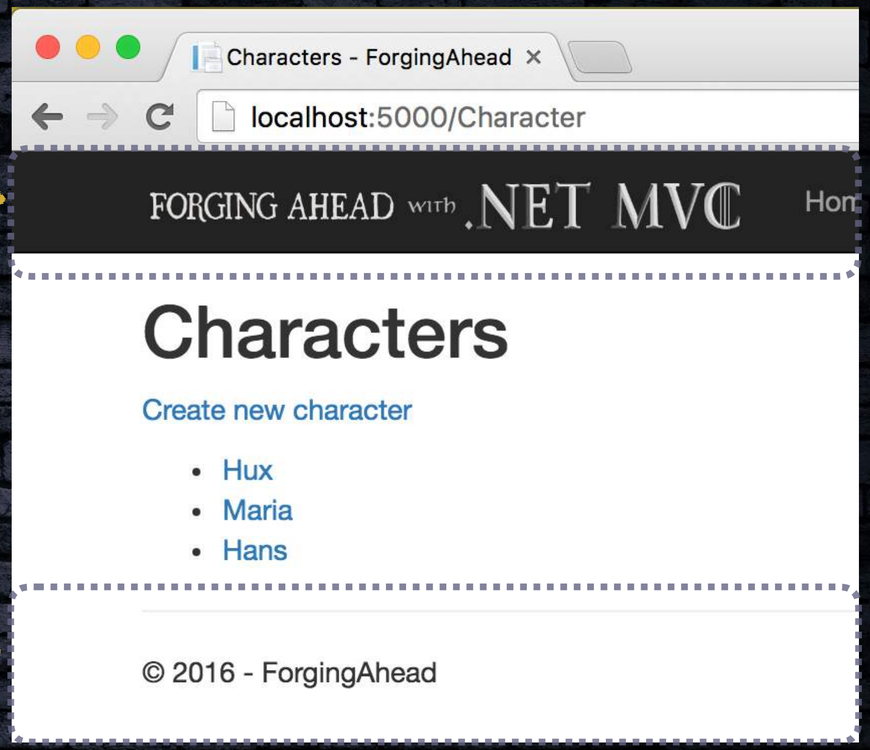
The only content that is changing is this part in the middle, known as the body.

Footer is the same on all pages.

Default Layout Page

The _Layout.cshtml file contain elements we can use on multiple pages.





Where Is Our Layout Set?

_ViewStart allows us to set code that applies when you return a view — for example, "Return View();".

- Views
  - Character
  - Shared
    - _Layout.cshtml
    - Error.cshtml
  - _ViewImports.cshtml
  - _ViewStart.cshtml

Views/_ViewStart.cshtml CSHTML

```
@{
    Layout = "_Layout";
}
```

Layout is a keyword in Razor that sets what layout the current view will use. We don't need the .cshtml after the layout's file name.

Default Layout Page

Views/Shared/_Layout.cshtml

CSHTML

So here is all the HTML we've been seeing and weren't sure where it was coming from!

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - ForgingAhead</title>

    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    @RenderSection("header", required: false)
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
…
```

Layout Parts: ViewData

Views/Shared/_Layout.cshtml

CSHTML

```
…
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - ForgingAhead</title>

    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    @RenderSection("header", required: false)
</head>
…
```

ViewData is a dictionary that allows us to pass data from our controller to our view.

A dictionary contains key-value pairs where ["string"] specifies the key, and we set and use it like a variable.

Layout Parts: RenderSection

Views/Shared/_Layout.cshtml

CSHTML

```
…
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - ForgingAhead</title>

    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    @RenderSection("header", required: false)
</head>
…
```

RenderSection ***lets us render content to specific named sections of our page. They can be required or optional.***

Layout Parts: RenderBody

Views/Shared/_Layout.cshtml

CSHTML

```
<!DOCTYPE html>
<html>
    <body>
        <div class="navbar navbar-inverse navbar-fixed-tap">…<div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; 2016 - ForgingAhead</p>
            </footer>
        </div>
        …
    </body>
<html>
```

RenderBody is where all content not set to a specific section will be rendered.

Putting Our Layout to Use

Now that we see our layout's parts, let's use them to solve our problem.

- Use **ViewData** for our page titles
- Use **Sections** to set some page metadata
- Use **RenderBody** to display our content

RenderBody works automatically and has been working this whole time.

Adding ViewData to Our Controller

Controllers\CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
    …
    public IActionResult Create(Character character) {…}

    public IActionResult Index()
    {
        ViewData["Title"] = "Characters";
        var model = _context.Characters.ToList();
        return View(model);
    }
}
…
```

ViewData is a dictionary, so we need to set it as such. _Layout is expecting the "Title" key, so we need to set that for it to map up.

ViewData is a property of the controller that is directly passed into the ViewData property of the view automatically when a view is returned.

ViewData **Is Carried From Controller to View**

CS

./Controllers/CharacterController.cs

```cs
public class CharacterController : Controller
{

    public IActionResult Index()
    {
        ViewData["Title"] = "Characters";
        var model = _context.Characters.ToList();
        return View(model);
    }
}
```

This will set our layout's ViewData, resulting in "Characters - ForgingAhead".

Views/Shared/_Layout.cshtml

```cshtml
<head>
    …
    <title>@ViewData["Title"] - ForgingAhead</title>
    …
</head>
```

Setting Our View to Populate a Section

./Views/Character/Index.cshtml

CSHTML

```
@model List<ForgingAhead.Models.Character>

@section header{
    <meta name="description" content="Lists all characters in ForgingAhead.">
}

<h1>
    Characters
</h1>
…
```

We'll set metadata that search engines will display when listing our page in our header section.

Views/Shared/_Layout.cshtml

```
…
<head>
    …
    @RenderSection("header", required: false)
</head>
…
```

@section **Replaces the Matching** @RenderSection

./Views/Character/Index.cshtml

CSHTML

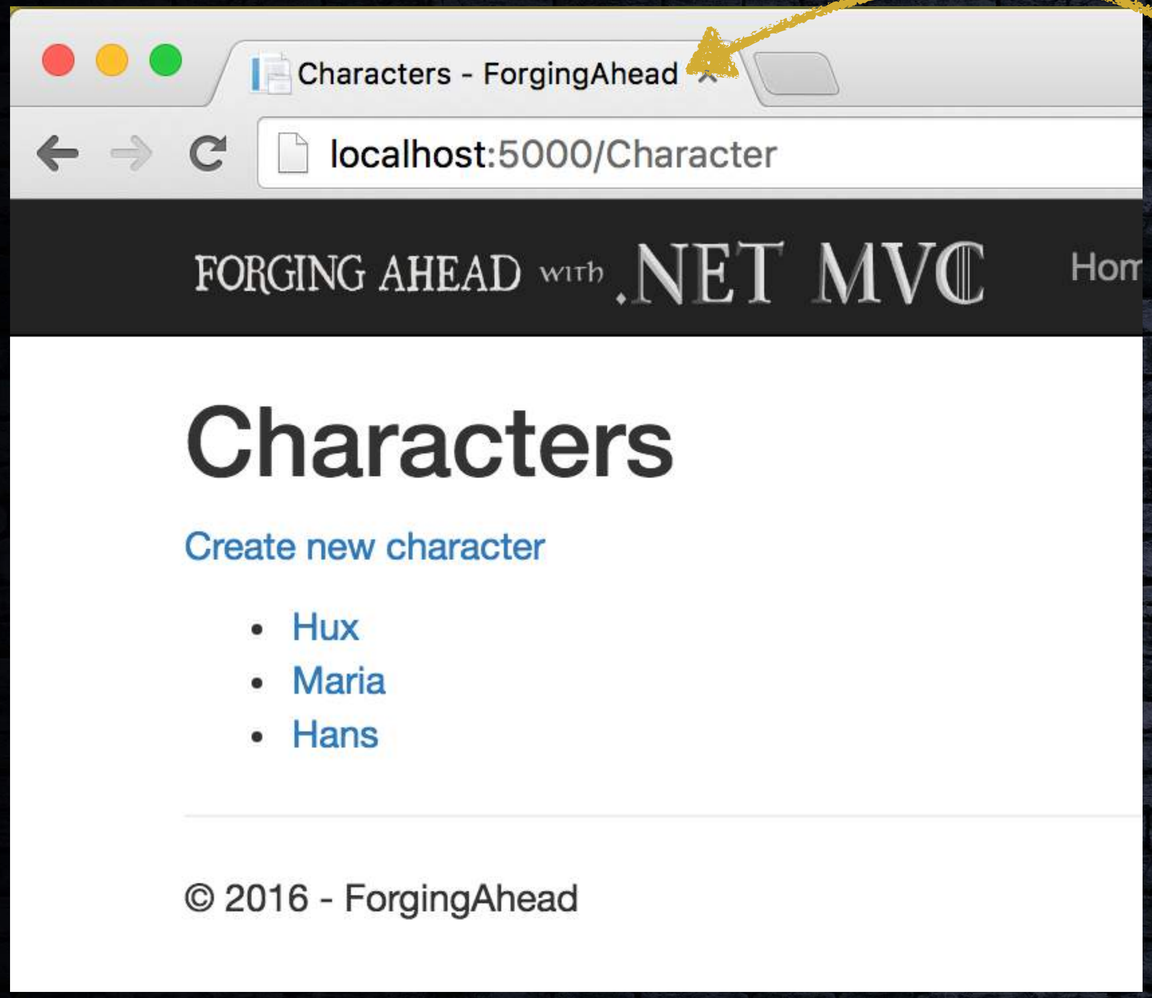
```
@model List<ForgingAhead.Models.Character>

@section header{
    <meta name="description" content="Lists all characters in ForgingAhead.">
}

<h1>
    Characters
</h1>
…
```

@section ***will match up with any*** *@RenderSection* ***with a matching name and replace it with its content.***

Views/Shared/_Layout.cshtml

```
…
<head>
    …
    @RenderSection("header", required: false)
</head>
…
```

The Result of Using Our Layout



Our ViewData has set the titles in both our browser and search results.

Search Results Example

```
Characters - ForgingAhead
localhost:5000/Character
Lists all characters in ForgingAhead.
```

Our header section set the text used to describe our page in search results.

Some Conventions to Note

There are several conventions you should be aware of when it comes to layouts.

- Layout naming convention is _PascalCase (note the underscore).
- All layouts typically go in the **Shared** folder.
- Section names are typically lowercase.
- If you have content that will only be displayed on some but not all pages using your layout, then that content shouldn't be in your layout.

Level 3 – Section 2

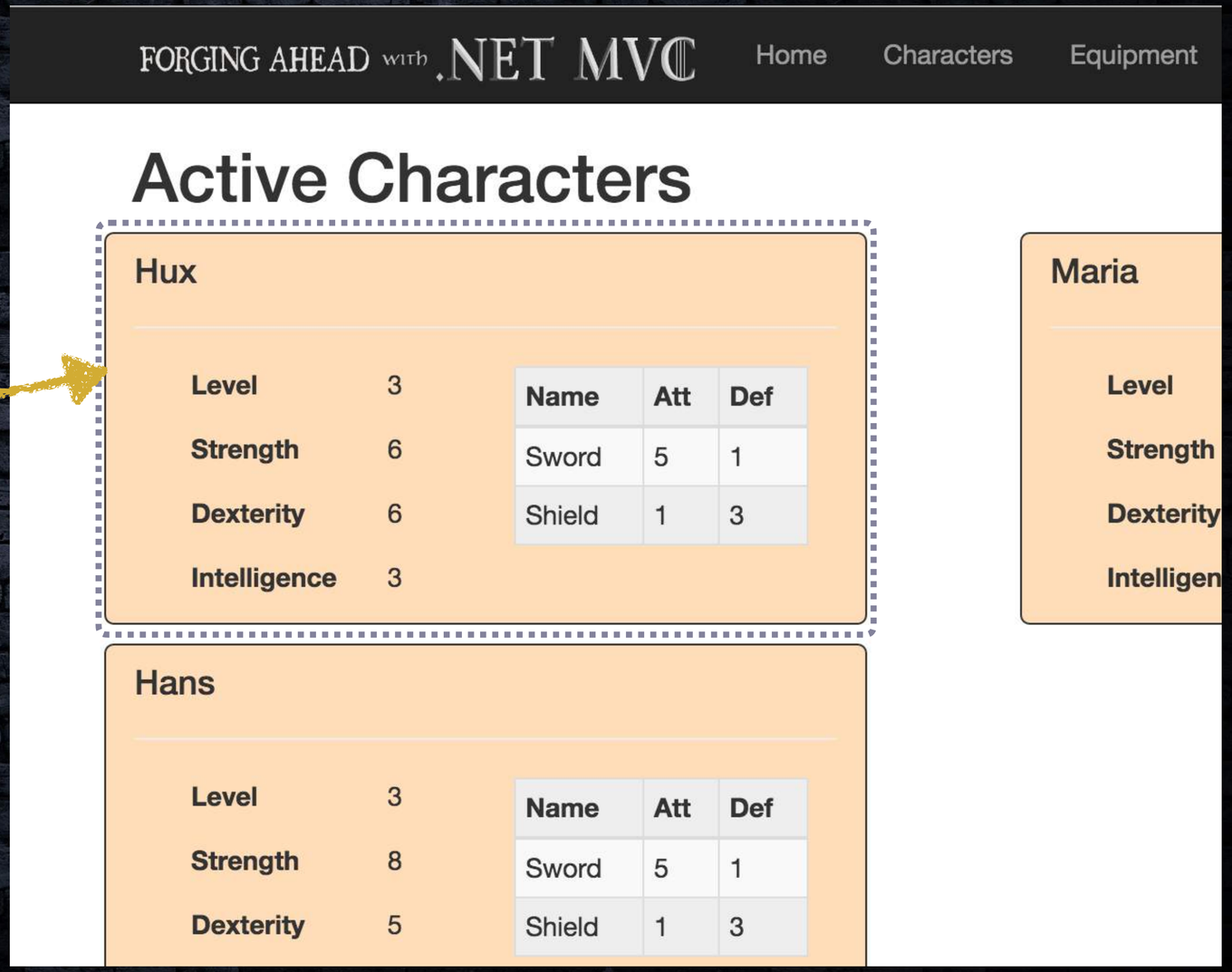
Making Our Presentation Smarter

Partials

Reusing Our Views With Partial Views

Our character details appear multiple times here and will be on our Details page as well.

We should make this into a partial view to avoid duplicating code.



Creating Our _CharacterSheet Partial View

- Views
  - Shared
    - _CharacterSheet.cshtml
    - _Layout.cshtml

Pascal case preceded by an underscore is the standard naming convention for partial views, editor views, and layouts.

Our _CharacterSheet Partial View

Controllers\Shared_CharacterSheet.cshtml

CSHTML

```
@model ForgingAhead.Models.Character

<div class="col-md-5 bordered-div">
  <h4>
    @Model.Name
  </h4>
  <hr/>
  <div class="col-md-6">
    <div class="form-group">
      <label asp-for="Level" class="control-label col-md-8"></label>
        <div class="col-sm-3">
          <p class="form-control-static">@Model.Level</p>
…
```

_CharacterSheet will just be a formatted view for all of our characters' properties.

Using Our Partial View in a View

Our Details view will just have its model declaration and the @Html.Partial helper.

Views/Character/Details.cshtml

CSHTML

```
@model ForgingAhead.Models.Character

@Html.Partial("_CharacterSheet",Model)
```

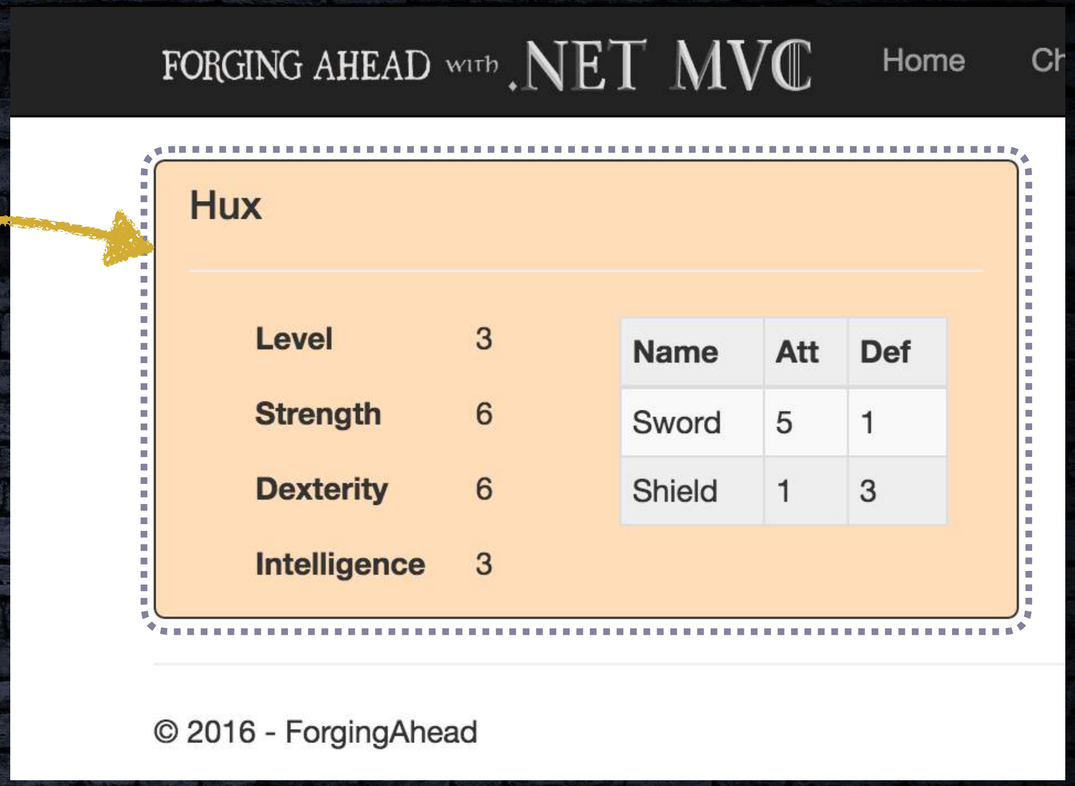
When this view loads, this line will be replaced with the contents of the _CharacterSheet partial.

We also need to pass our Character into the partial.

Results of Our Partial on Our Details Page

The end user viewing our Details page will have no way of telling we used a partial.

We can reuse this partial on the Index page and Character Details page.



Set Our Home/Index to Use _CharacterSheet

Now we have what we need to set up our Active Characters List on Home/Index.

Views/Home/Index.cshtml CSHTML

```
@model List<ForgingAhead.Models.Character>
<h1>
  Active Characters
</h1>
<div class="row">
  @foreach(var item in Model)
  {
    @Html.Partial("_CharacterSheet",item)
  }
</div>
```

Instead of doing <Label>@item.Name</Label>, we'll pass item through our @Html.Partial.

Setting Up Home/Index to Get Active Characters

Controllers\HomeController.cs

CS

```
…
public class HomeController : Controller
{
    …

    public IActionResult Index()
    {
        var model = _context.Characters.Where(e => e.IsActive).ToList();
        return View(model);
    }
}
…
```

We can just set our model to all characters where IsActive is true, then pass that back to our view.

We're All Set Up on Partials!

Now we can update _CharacterSheet once instead of updating both Details and Index.

Details Page

FORGING AHEAD with .NET MVC Home Ch

Hux

Level	3
Strength	6
Dexterity	6
Intelligence	3

Name	Att	Def
Sword	5	1
Shield	1	3

© 2016 - ForgingAhead

Index Page

FORGING AHEAD with .NET MVC Home Charac

Active Characters

Hux

Level	3
Strength	6
Dexterity	6
Intelligence	3

Name	Att	Def
Sword	5	1
Shield	1	3

Hans

Level	3
Strength	8
Dexterity	5
Intelligence	2

Name	Att	Def
Sword	5	1
Shield	1	3

© 2016 - ForgingAhead

Some Conventions to Note

There are several conventions you should be aware of when it comes to partials.

- Partials naming convention is _PascalCase (note the underscore).
- All partials can be placed in the **Shared** folder *or* partials can be placed in folders based on what model they use.

Partials in Shared

- Views
  - Character
  - Shared
    - _CharacterPartial.cshtml
    - _NonspecificPartial.cshtml

OR

Partials Organized by Model

- Views
  - Character
    - _CharacterPartial.cshtml
  - Shared
    - _NonspecificPartial.cshtml

Level 3 – Section 3

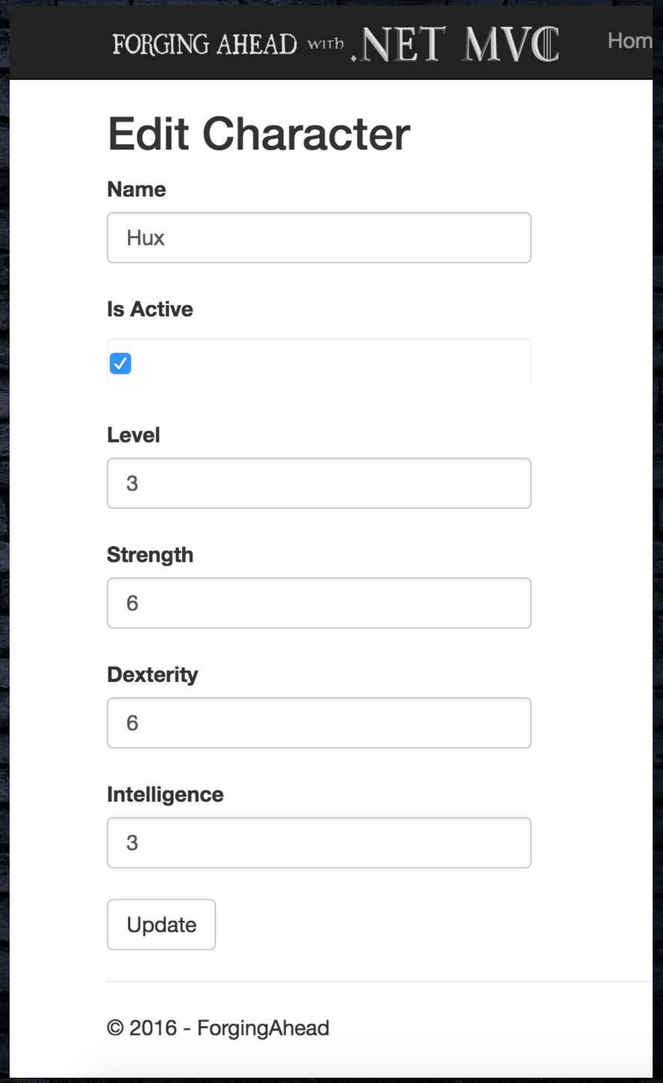
Making Our Presentation Smarter

Editor Templates

Updating Our Characters

Our players will need to occasionally update their characters.

We need to make a page like our Create page so we can easily update our characters.



Adding Edit Action to Controller

The only difference between our Edit action and Details action is the title and returned view.

Controllers/CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
  …
  public IActionResult Edit(string name)
  {
    ViewData["Title"] = "Edit " + name;

    var model = _context.Characters.FirstOrDefault(e => e.Name == name);
    return View(model);
  }
…
```

Change our Title to be "Edit Character.Name".

Using the EditorForModel Helper in the Edit View

We could make the edit view by writing all of the HTML by hand, but using editor helpers can simplify things for us.

Views/Character/Edit.cshtml

CSHTML

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

<form asp-action="Update" asp-controller="Character">
    @Html.EditorForModel()
    <input type="submit" value="Update" class="btn btn-default"/>
</form>
```

Set up our asp-action and asp-controller to work with Controller/Update.

@Html.EditorForModel automatically creates labels and input fields in this form based on properties in the model.

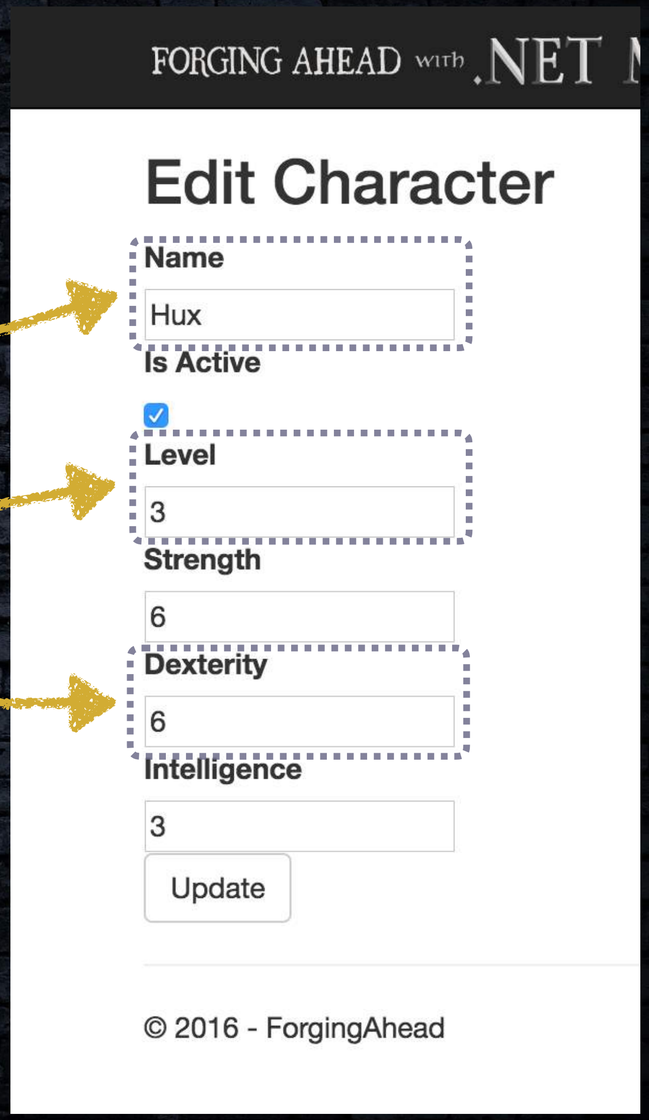
Automatically Generated Template

EditorForModel **will automatically generate a template using its properties.**

Models/Character.cs

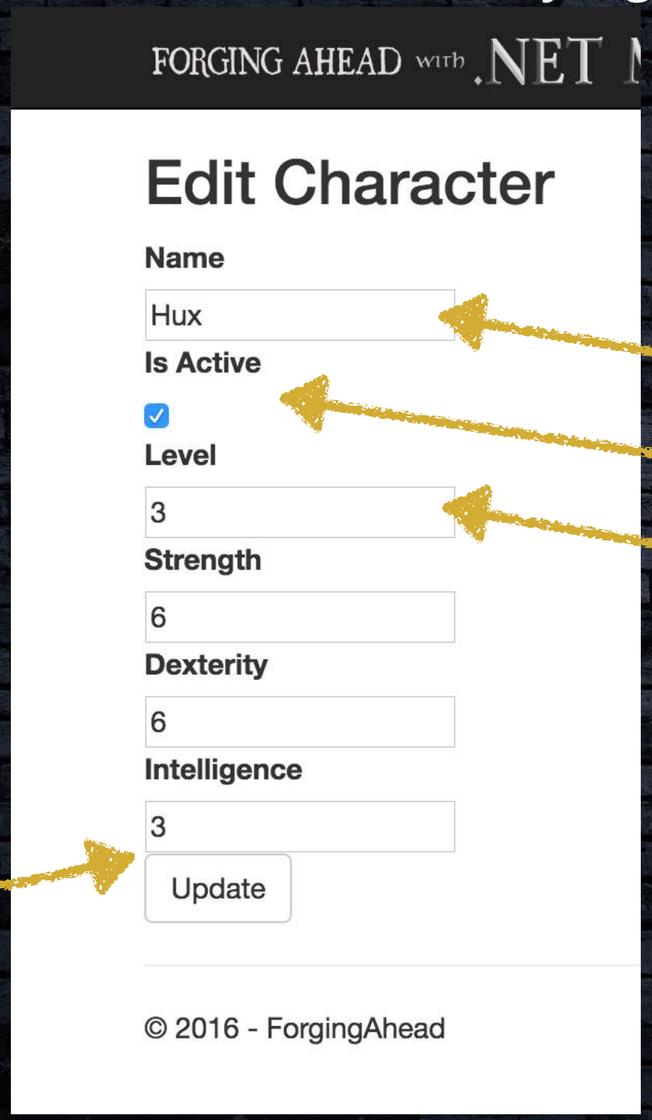
CS

```
public class Character
{
    public string Name { get; set; }
    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; }
}
```



Our Template Works but Could Be Better

This looks kind of like our Create view, but the styling is a bit underwhelming.



There are three different field types in this form.

String

Boolean

Int32 (Integer)

The spacing between our Update button and Intelligence field is especially bad.

Overriding Editor Templates

EditorFor **will use the templates in our** EditorTemplates **folder instead when available.**

- Views
  - Character
  - Home
  - Shared
    - EditorTemplates
      - Boolean.cshtml
      - Int32.cshtml
      - String.cshtml

First, create an** EditorTemplates **folder in our** Views/Shared **directory.

Then, add** Boolean, Int32, **and** String.cshtml **files to that new folder.

Editor templates can be applied to any datatype or object.

String.cshtml **Editor Template**

EditorFor **will use this template any time it needs to handle a** string.

Views/Shared/EditorTemplates/String.cshtml

CSHTML

This datatype would be int32 or boolean in the other templates.

```
@model string
<input class="form-control" asp-for="@Model" /> <br />
```

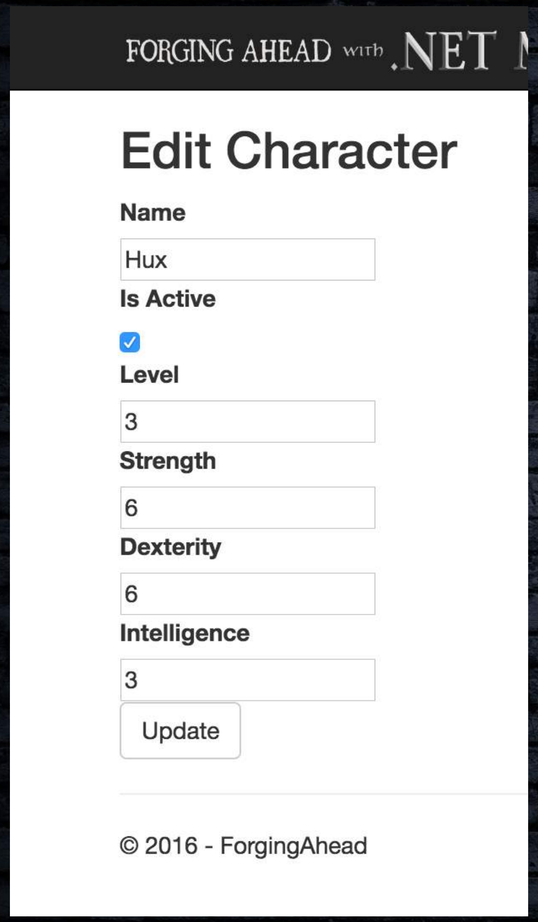
This is a style from the Bootstrap framework that we can use to style this input.

Our Int32.cshtml and Boolean.cshtml will be the same with their respective datatypes as their models.

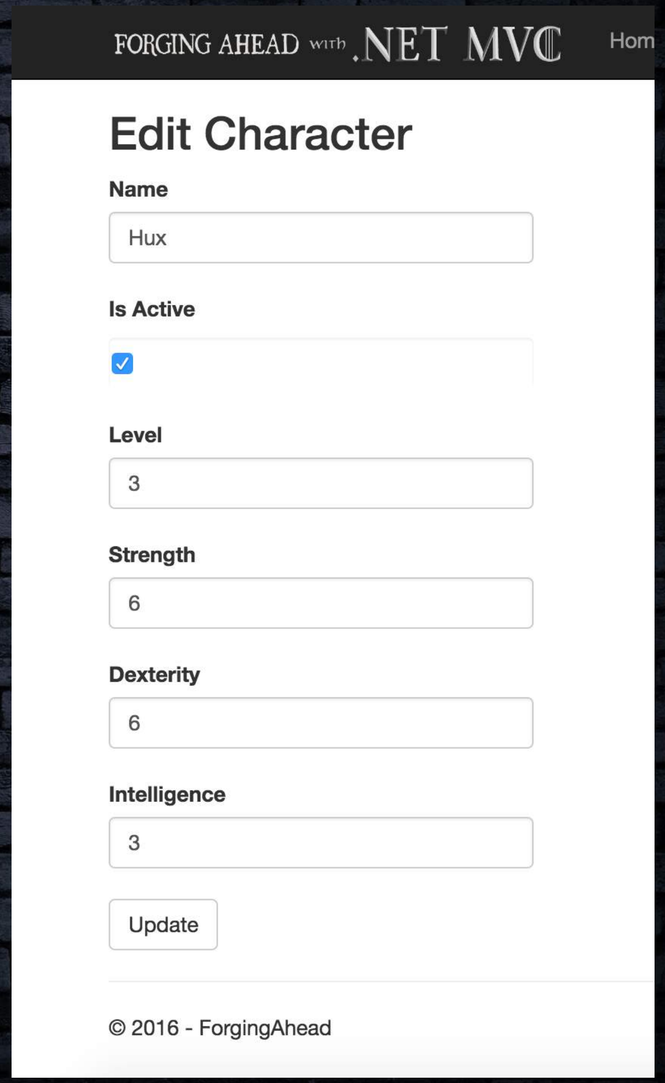
Now We Have a Proper Edit View

Our players can update their characters and, as an added bonus, the form looks decent.

before



after

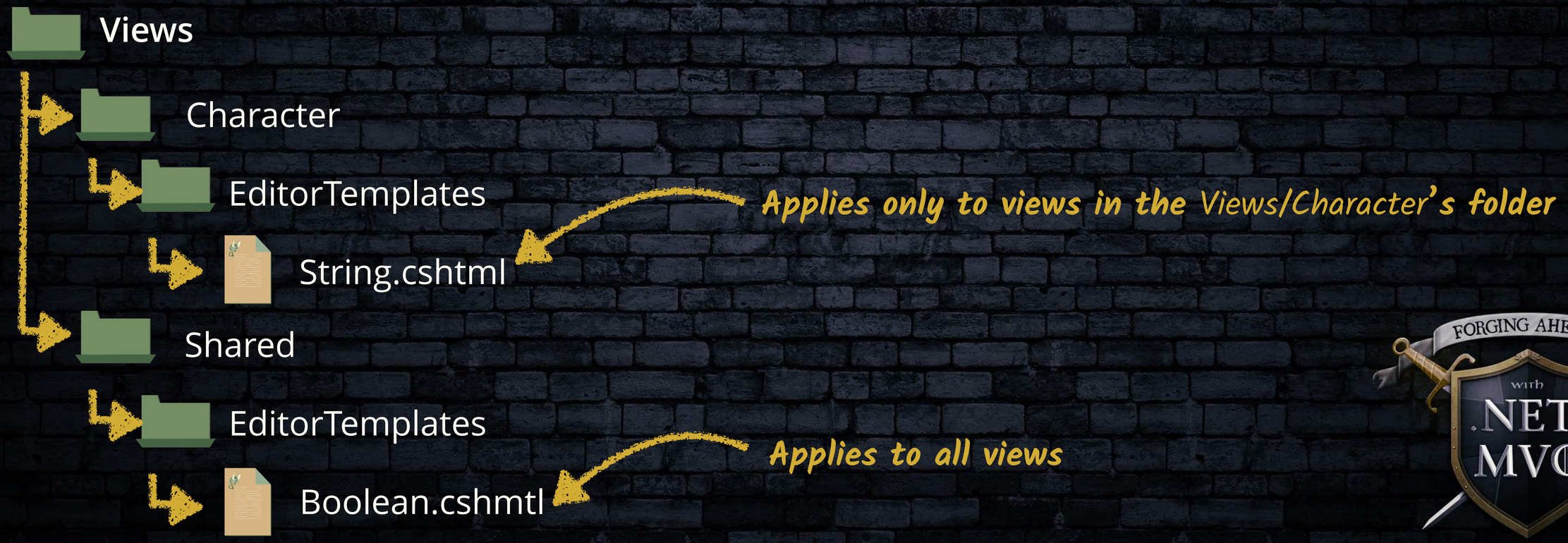


Editor templates are a way to set input styles globally throughout your application.

Some Conventions to Note

There are several conventions you should be aware of when it comes to editor templates.

- Editor templates naming convention is PascalCase.
- Editor templates go in an **EditorTemplates** subfolder, which can be located in different folders depending on the intended scope.



FORGING AHEAD
with
.NET
MVC

Level 4

Creating Logical URLs

Custom Routes

Making Our URLs More User Friendly

Some of our URLs are hard to remember and understand. We should simplify them.

Current URL to Our Details Page

localhost:5000/Character/Details?Name=Hux

Desired URL to Our Details Page

localhost:5000/Character/Hux/Details

- It's clearer where a URL will take us
- Search engines use URLs in their algorithms
- We should avoid query strings

Where Do We Want to Set Our Custom Route?

Application-wide routes are typically set in Startup.cs — otherwise, they can be set on the specific action affected.

Our default route is application-wide, so it's set up in our Startup.cs file.

localhost:5000/Character/Details?Name=Hux

Our new routes will be "per action" and will live in our CharacterController.cs file.

localhost:5000/Character/Hux/Details

Adding a Custom Route

To create a custom route, we just add a route attribute with the route details to our action.

Controllers/CharacterController.cs

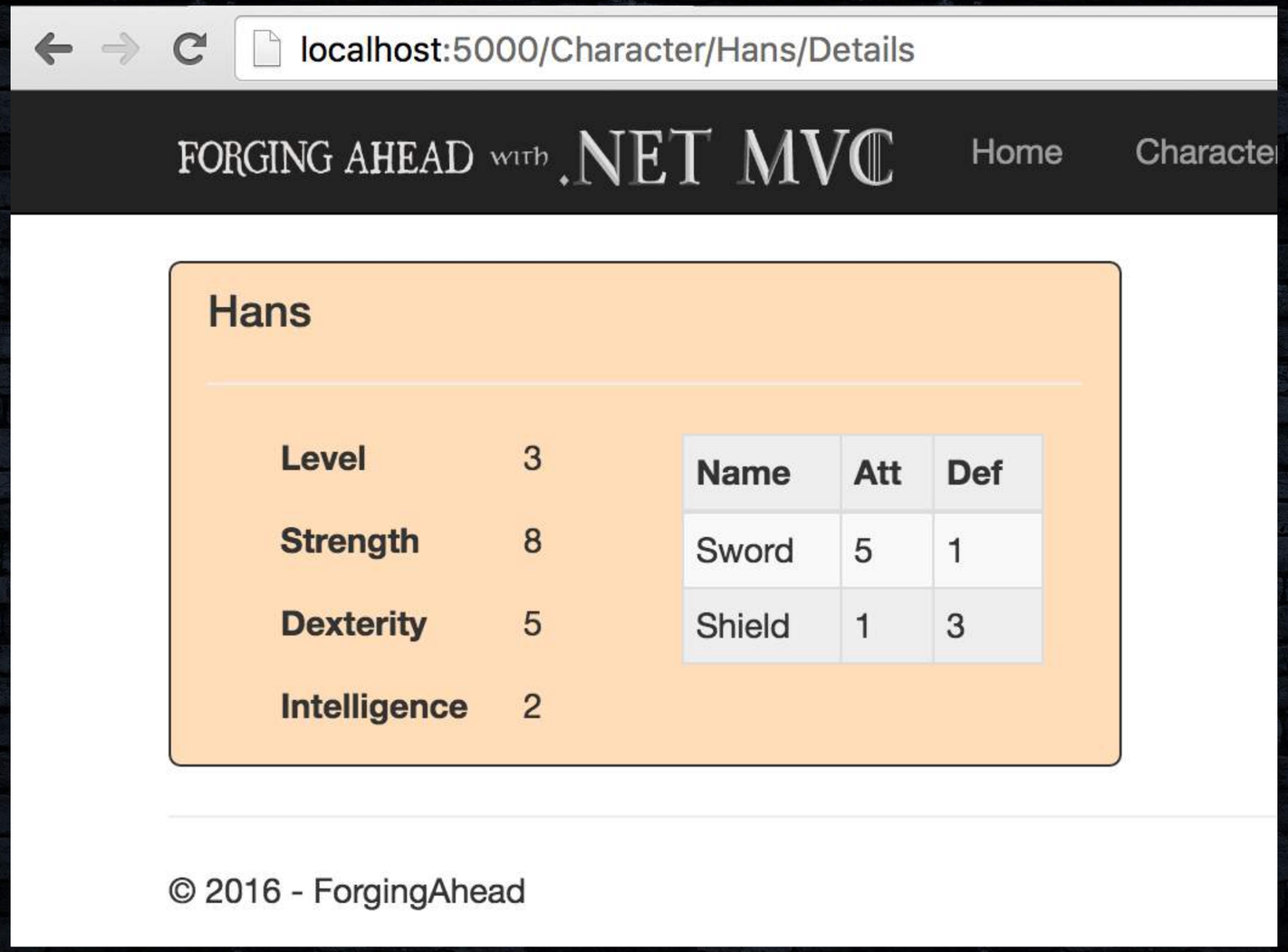
CS

```
…
public class CharacterController : Controller
{
…
  [Route("Character/{name}/Details")]
  public IActionResult Details(string name)
  {
    ViewData["Title"] = name;

    var model = _context.Characters.FirstOrDefault(e => e.Name == name);
    return View(model);
  }
…
}
```

Routes tied to specific actions should specify everything that isn't a parameter to reduce risk of conflict.

Now We Have Much Nicer URLs

This will help our users with our application, as well as help us show up on search results.



We're Seeing Weird Stuff With Create

Looking at how users are accessing our website, we're seeing something we didn't expect.

We expect our users to reach our Character Create page through:

localhost:5000/Character/Create

Then use the form we've made to create their character

But we're seeing users bypass our form to create characters using query strings...

localhost:5000/Character/Create?Name=Test&IsActive=True&Level=1&Strength=5&Dexterity=5&Intelligence=5

This can be a serious problem depending on what data is passed.

URLs Shouldn't Be Used to Submit Data

Users can directly create a character using query strings because we allow both GET and POST.

- URLs have a limited length.
- URLs can leak sensitive data.
- URLs posting data can accidentally be resubmitted.
- URLs use the verb GET, which is meant to request data, not submit it.

https://localhost:5000/Account/Login?UserName=Administrator&Password=InPlainText

Even with encryption, passing sensitive data via query strings is EXTREMELY dangerous! GET requests can leak sensitive information easily!

HttpVerbs **Help Indicate Data Interaction**

HTML forms and URLs only support GET **and** POST**, so we're only going to worry about them.**

- GET **- Requests data from the application**
- POST **- Submits data to the application**

Unless we specify a verb, all action methods will allow both *GET* ***and*** *POST* ***by default.***

Examples of GET

GET is all about making a request for data.

- Index - Requests a list of all characters
- Details - Requests a character's data
- Edit - Requests a character's data

Adding the HttpGet Attribute Allows GET

Specifying that Index is an HttpGet, it will only accept GET and no longer accept POST requests.

Controllers/CharacterController.cs

CS

```
…
public class CharacterController : Controller
{
…
  [HttpGet]
  public IActionResult Index()
  {
    ViewData["Title"] = "Characters";

    var model = _context.Characters.ToList();
    return View(model);
  }
…
}
```

Any action that just returns data will typically use GET.

Examples of POST

POST is about making a request to submit or update data.

- Create - Requests to add new character data
- Update - Requests to update character data
- Delete - Requests to delete character data

Adding the HttpPost Attribute Allows POST

Specifying that Create is an HttpPost means Create will no longer accept query strings.

Controllers/CharacterController.cs

CS

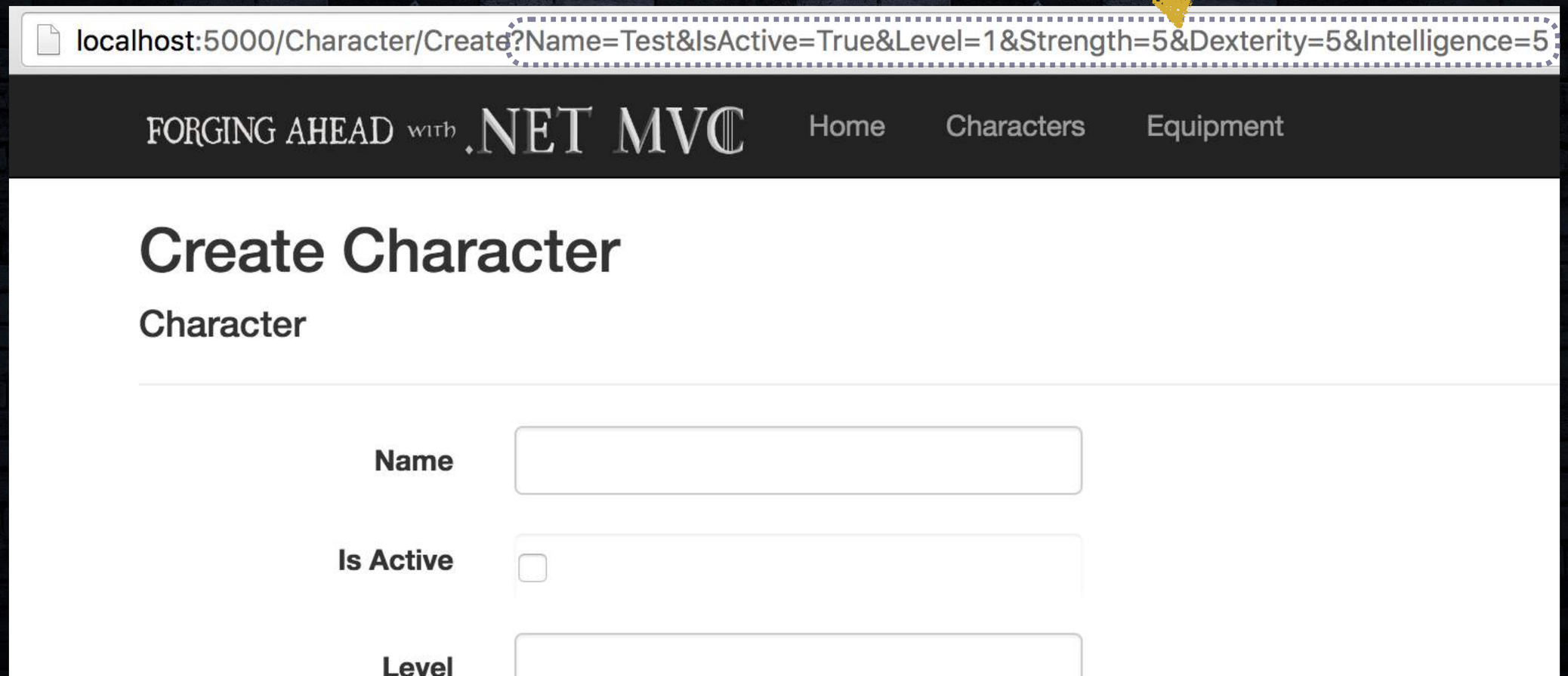
```
…
public class CharacterController : Controller
{
…
  [HttpPost]
  public IActionResult Create(Character character)
  {
    _context.Characters.Add(character);
    _context.SaveChanges();

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

Any action we submit data to will typically use POST.

Query Strings No Longer Work

If we try to access Create using query strings like before, we'll be taken to the Create view.

Now, no matter what query strings we put after Create in our URL, we'll be taken to the Create view.



What Verb Do We Use With Our Delete Action?

If we created an HttpRequest, we'd use Delete — but without it, we need to fallback to POST.

Controllers/CharacterController.cs

CS

```
…
  [HttpPost]
  [Route("Character/{name}/Delete")]
  public IActionResult Delete(string name)
  {
    var character = _context.Characters.FirstOrDefault(e => e.Name == name);
            _context.Characters.Remove(character);
            _context.SaveChanges();
    return RedirectToAction("Index");
  }
…
```

We actually have HttpDelete as an attribute, but unless we use JavaScript, we'll need to use HttpPost instead.

What Verb Do We Use With Our Update Action?

Updates are tricky, as PATCH, POST, or PUT could be used. In our case, we'll want POST.

Controllers/CharacterController.cs

CS

POST is the most appropriate for updating an entire record without creating one if it doesn't exist.

```
…
  [HttpPost]
  public IActionResult Update(Character character)
  {
    _context.Entry(character).State = EntityState.Modified;
    _context.SaveChanges();
    return RedirectToAction("Index");
  }
…
```

Now We're Using the Proper HttpVerbs

Using the proper verbs is good for a variety of reasons that will benefit our application:

- It helps keep sensitive information from leaking
- Prevents search engines from indexing actions for submitting data
- You'll need to know **HttpVerbs** if you host or consume a web service

FORGING AHEAD
with
.NET
MVC

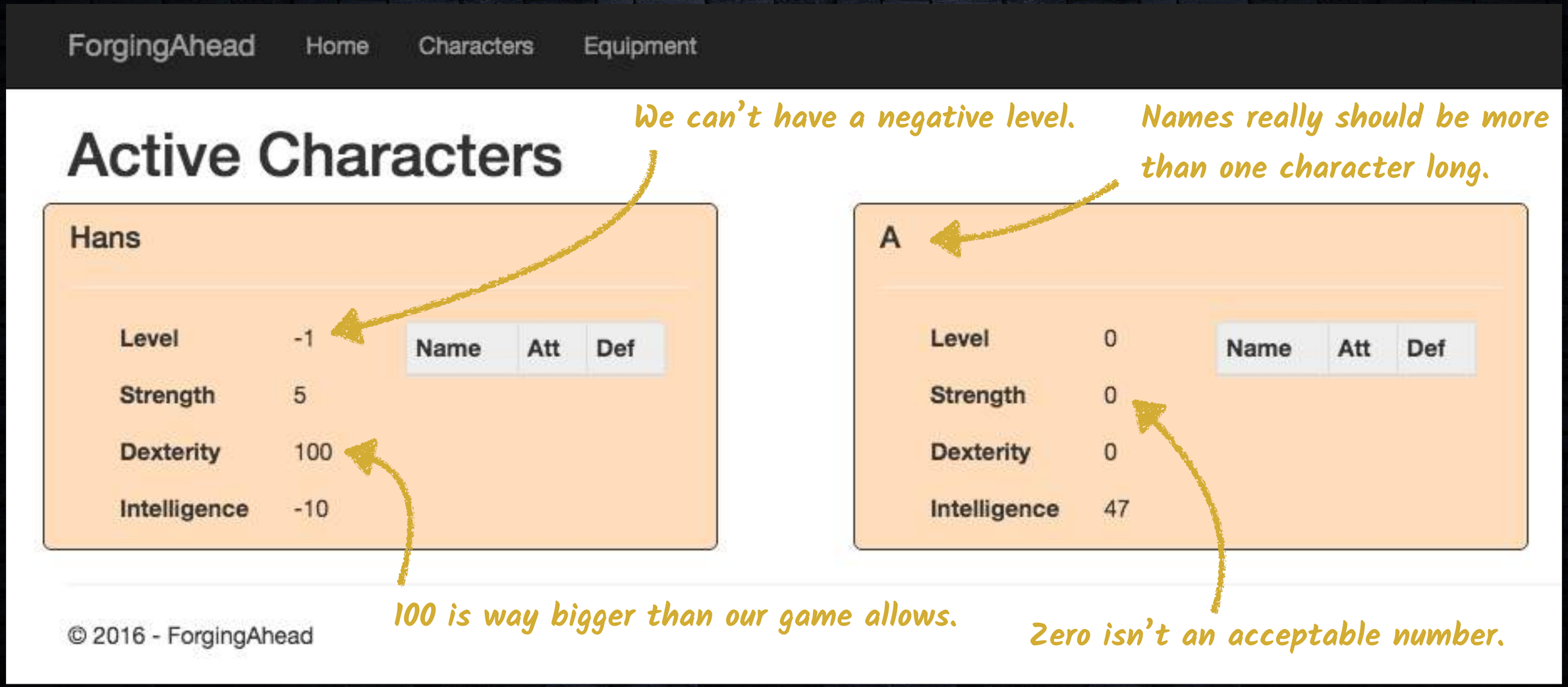
Level 5 – Section 1

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

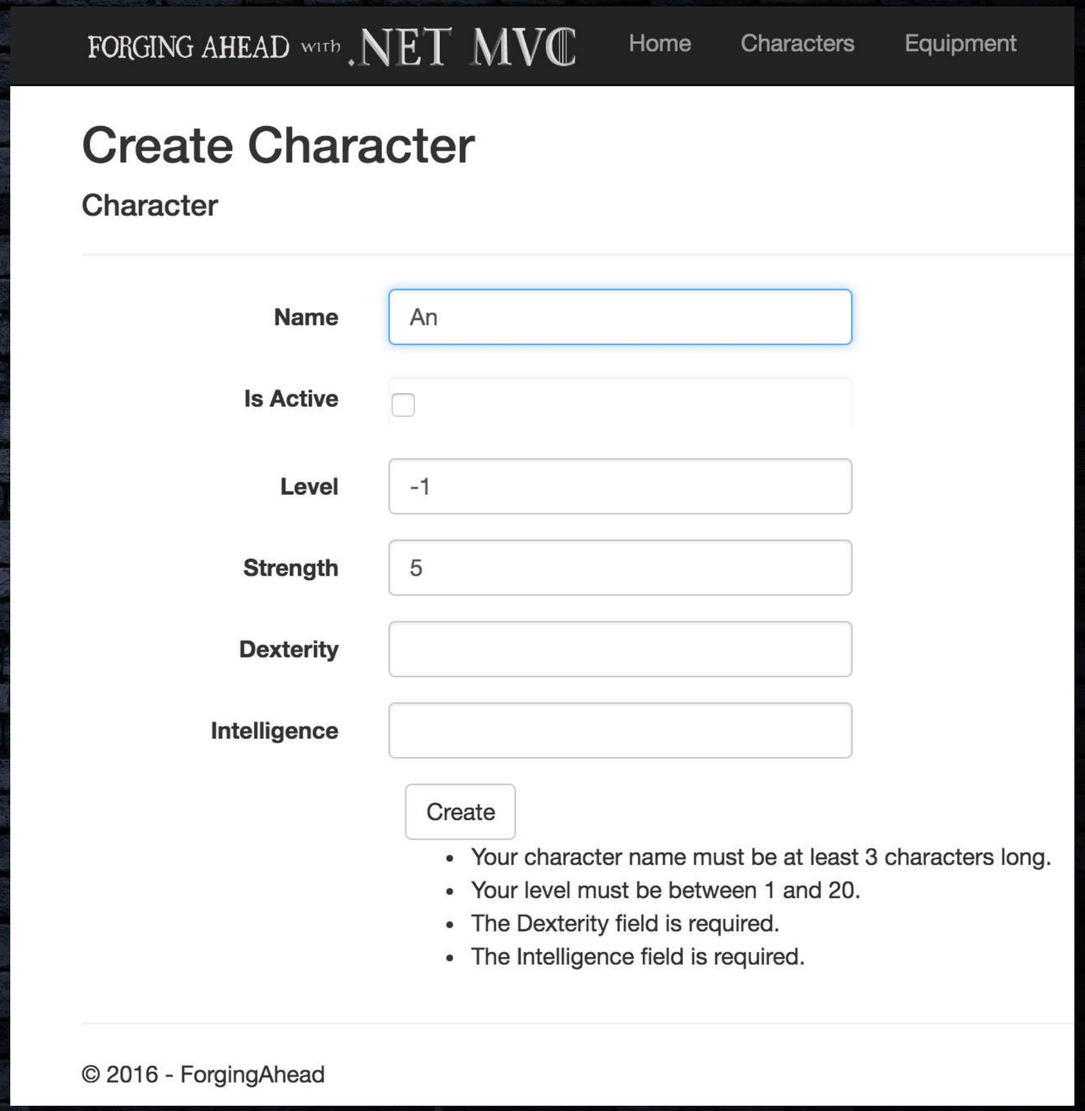
Validating Our Input

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

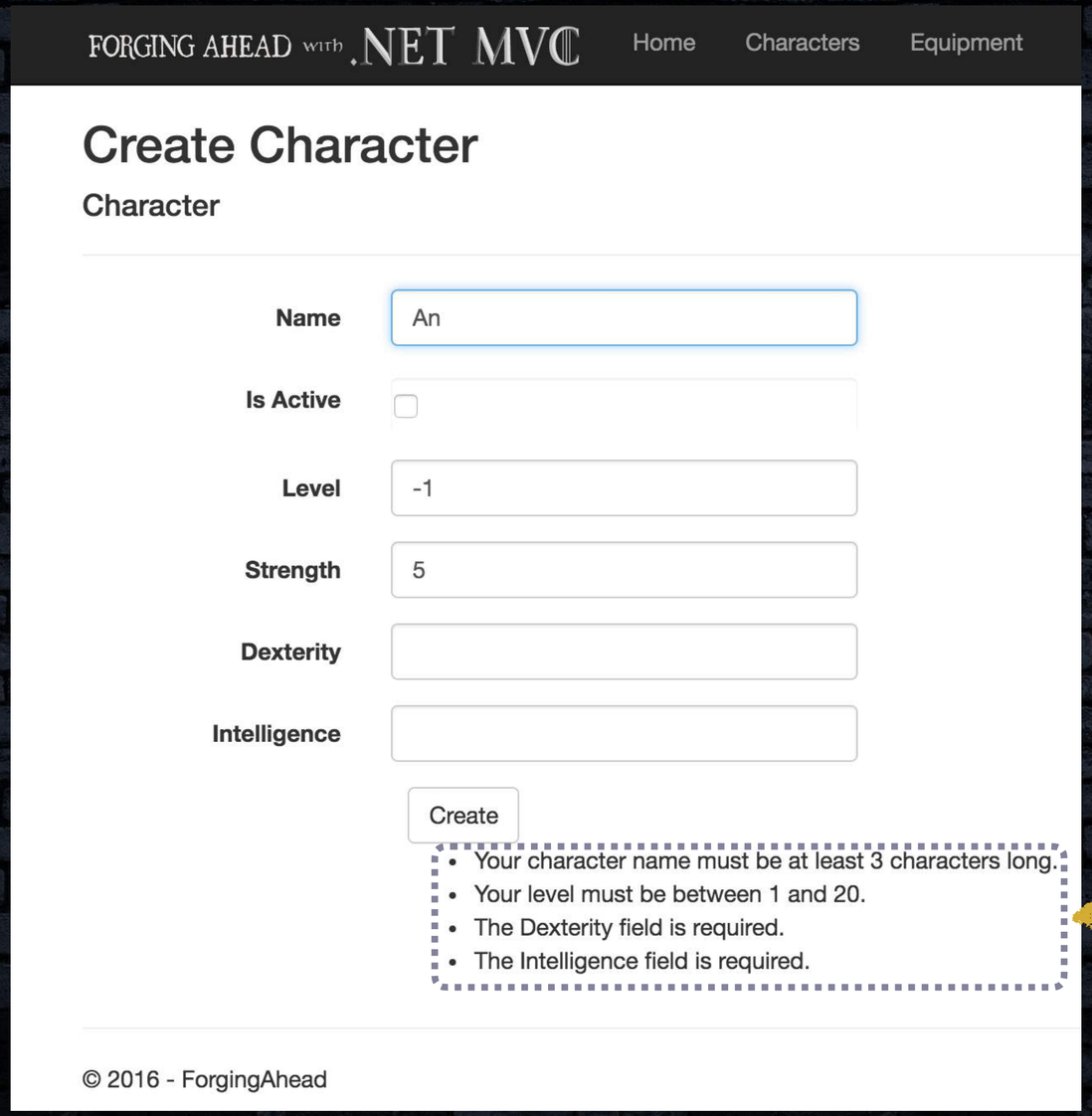
CSHTML

```
@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

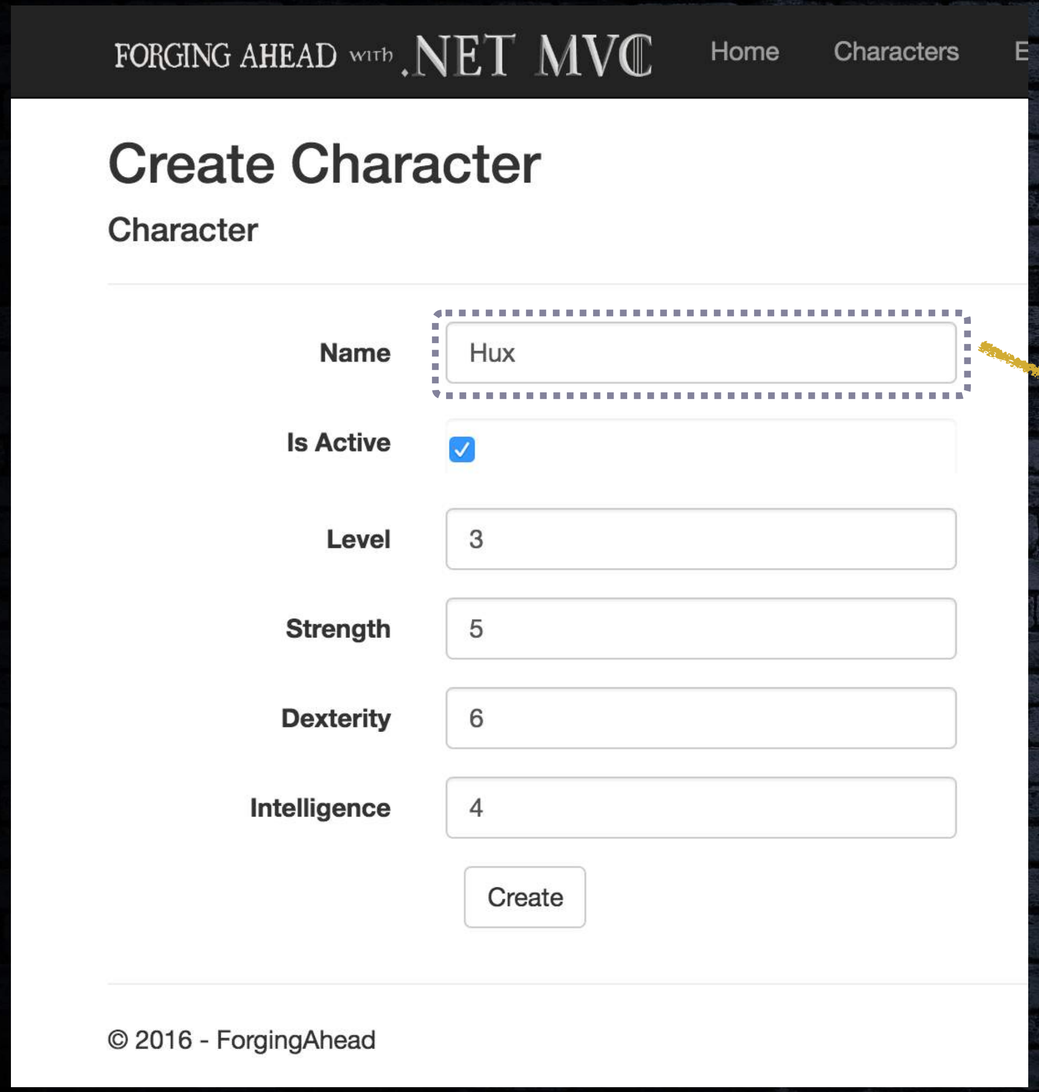
CSHTML

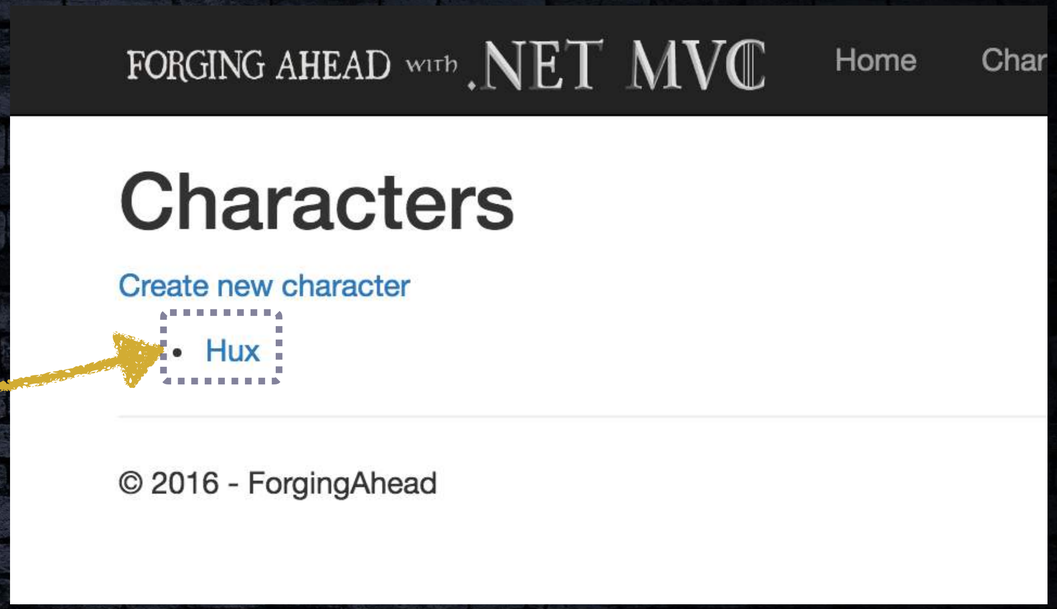
```
…
public class CharacterController : Controller
{
…
  public IActionResult Create(Character character)
  {
      if(!ModelState.IsValid)
        return View(character);
      _context.Characters.Add(character);
      _context.SaveChanges();

    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

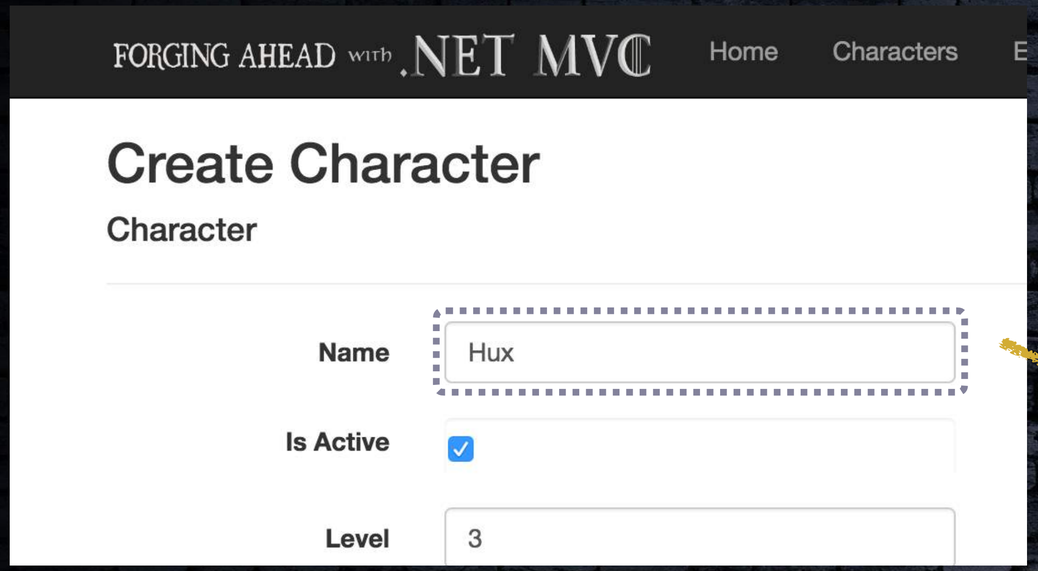
Entering a character with no validation errors works fine and adds the character to our list.

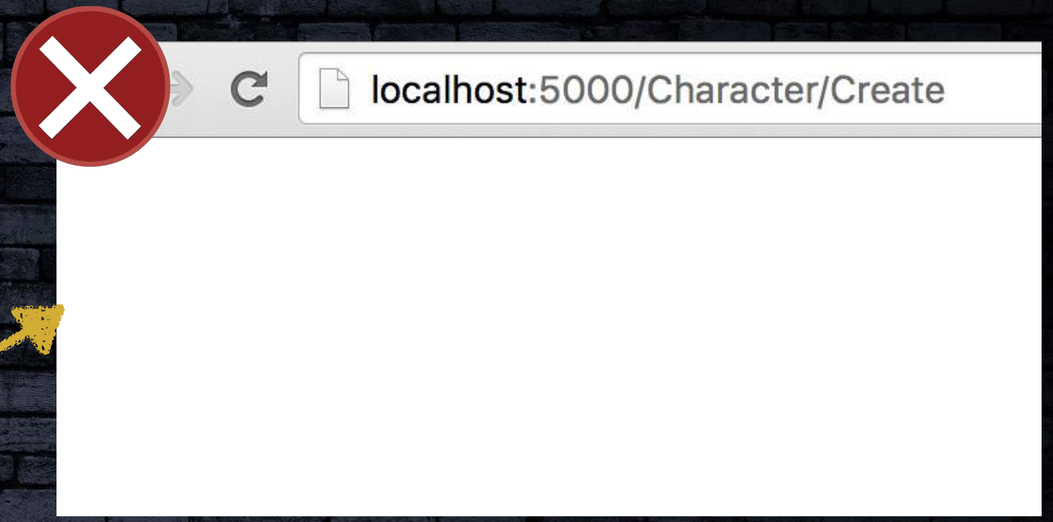




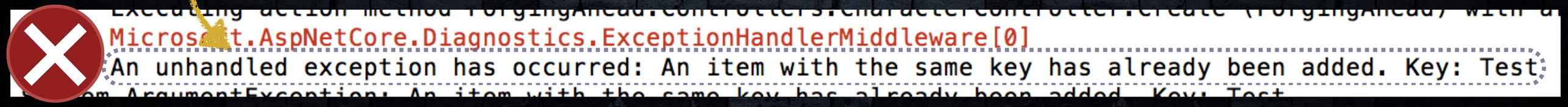
Using the Same Name Twice Is Broken

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");
  }
…
```

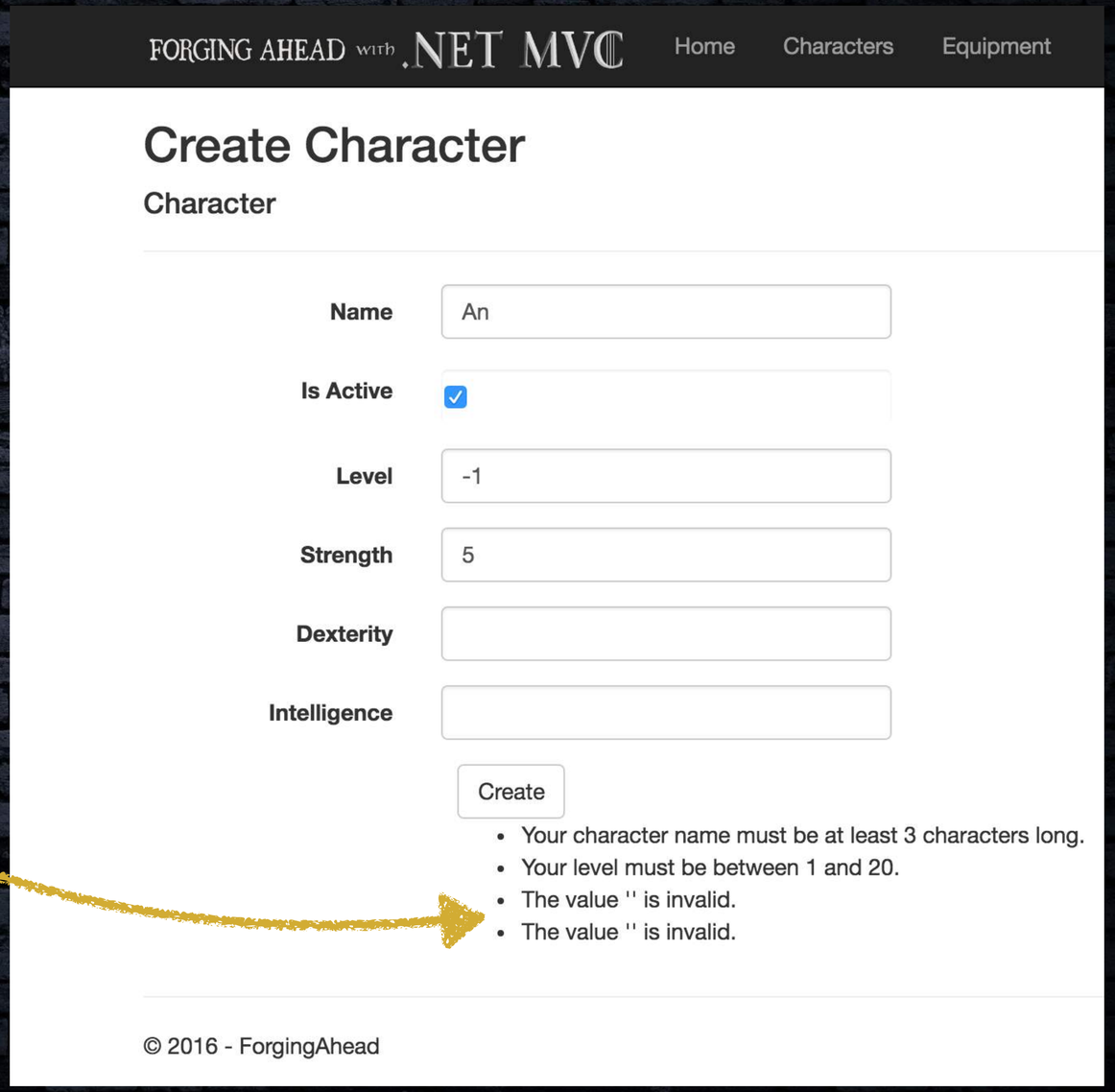
Field name

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

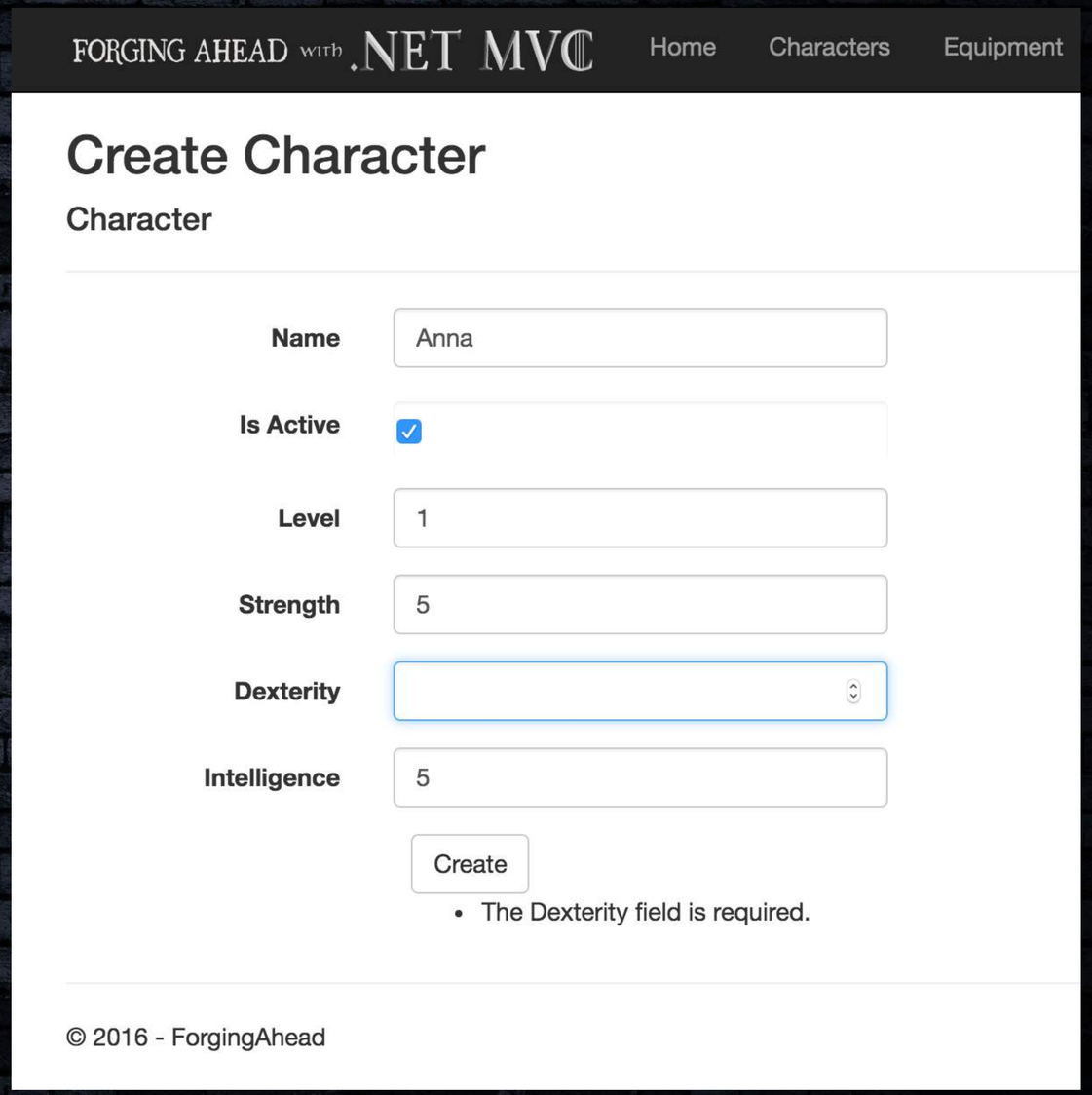
```
…
public class Character
{
  …
  [Required]
  [Range(1, 20)]
  public int Level { get; set; }
  [Required]
  [Range(1, 18)]
  public int Strength { get; set; }
  [Required(ErrorMessage = "The Dexterity field is required.")]
  [Range(1, 18)]
  public int Dexterity { get; set; }
  …
}
```

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