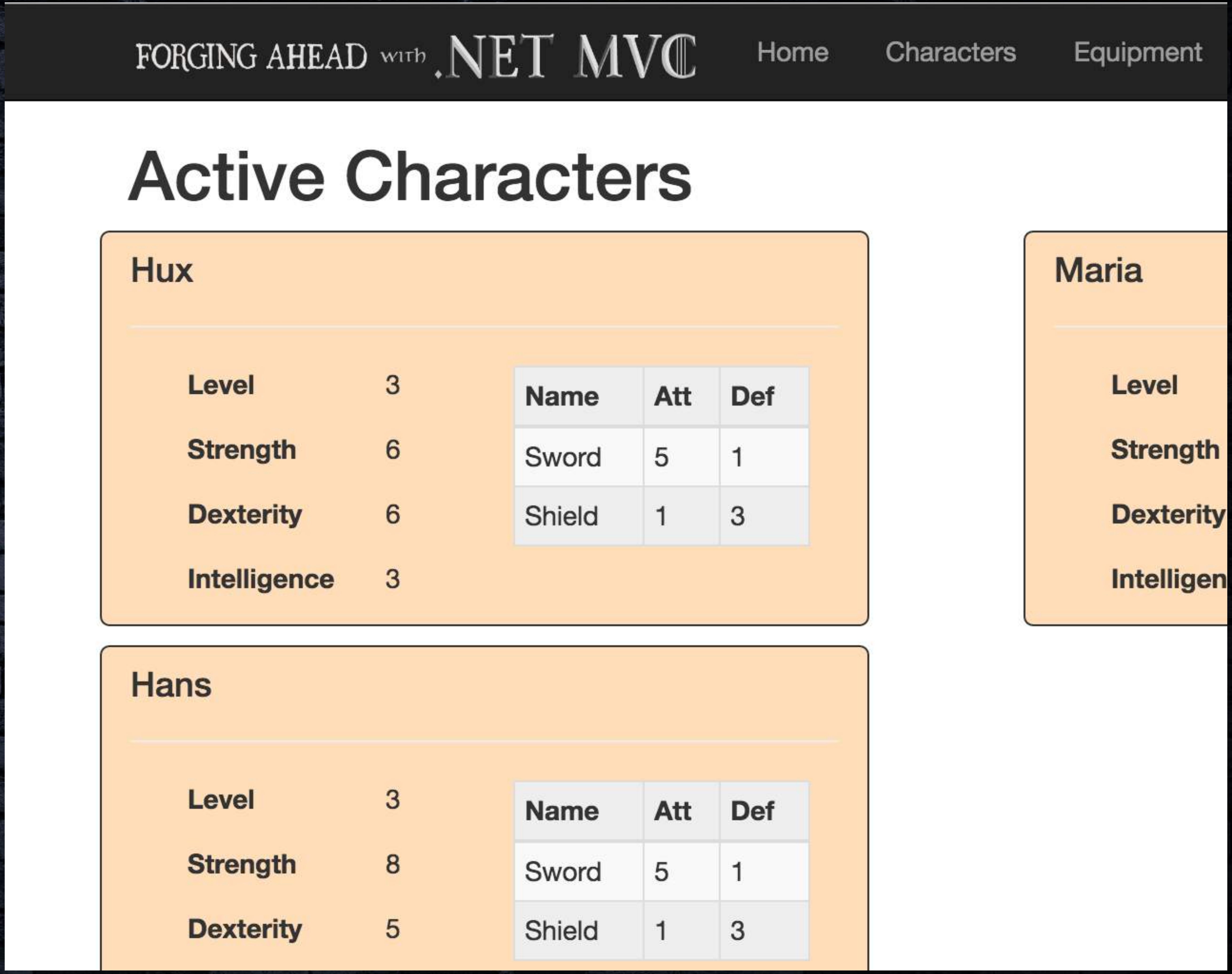
FORGING AHEAD with .NET MVC

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



Active Characters

Hux

Level	3
Strength	6
Dexterity	6
Intelligence	3

Name	Att	Def
Sword	5	1
Shield	1	3

Maria

Level

Strength

Dexterity

Intelligen

Hans

Level	3
Strength	8
Dexterity	5

Name	Att	Def
Sword	5	1
Shield	1	3

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

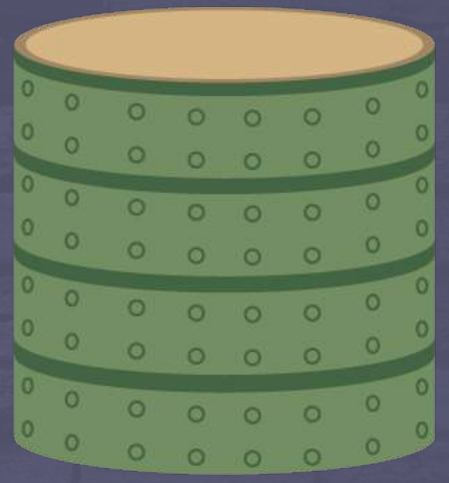
Options for Storing Data Long Term

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

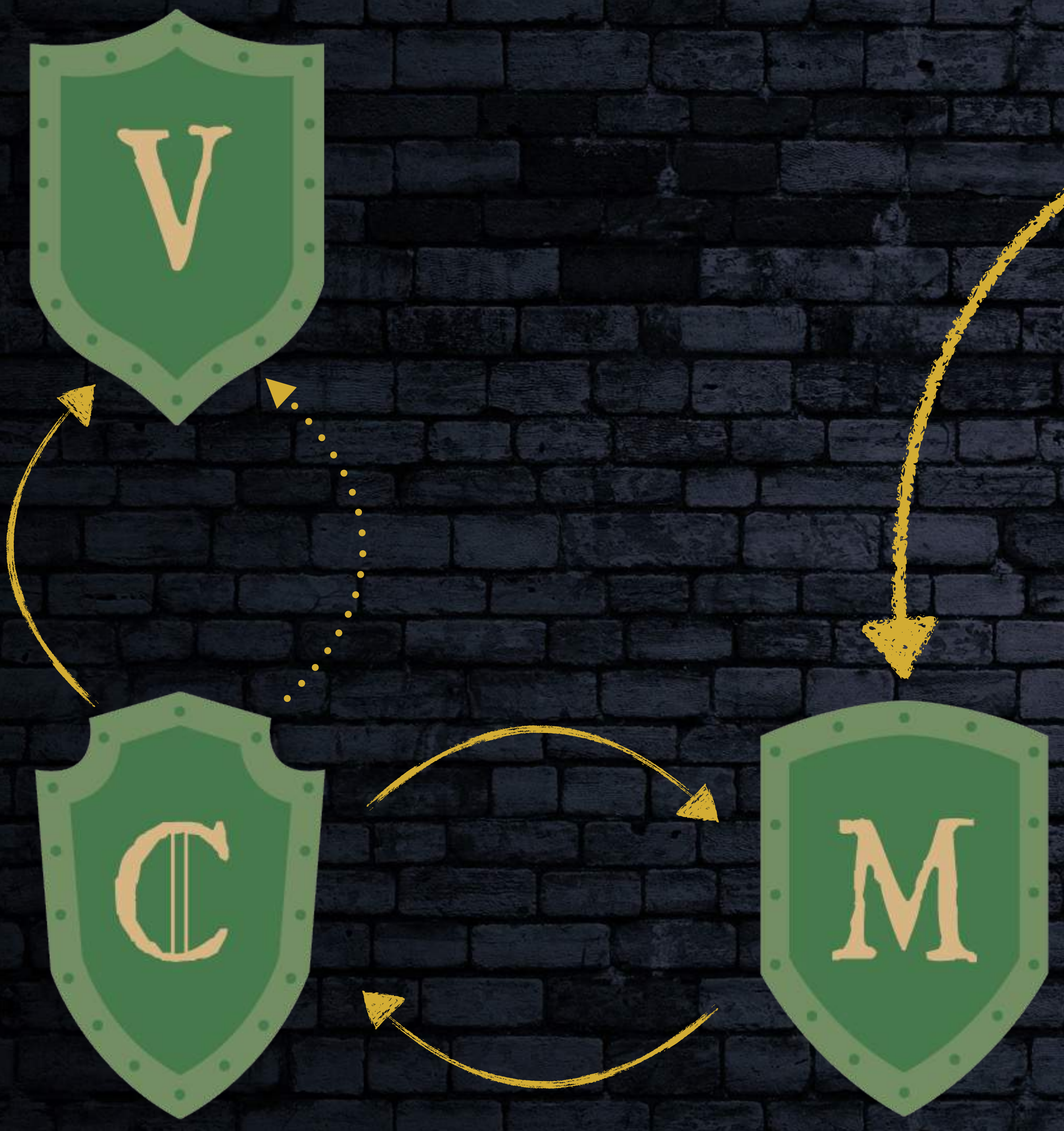


- *Optimized for relational data*
- *Less issues with multiple active users*
- *"Bad data" is easier to mitigate*

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

- ForgingAhead
  - Controllers
  - Models
    - Character.cs
  - Views

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

- **ForgingAhead**
  - Controllers
  - Models
    - ApplicationDbContext.cs
    - Character.cs
  - Views

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

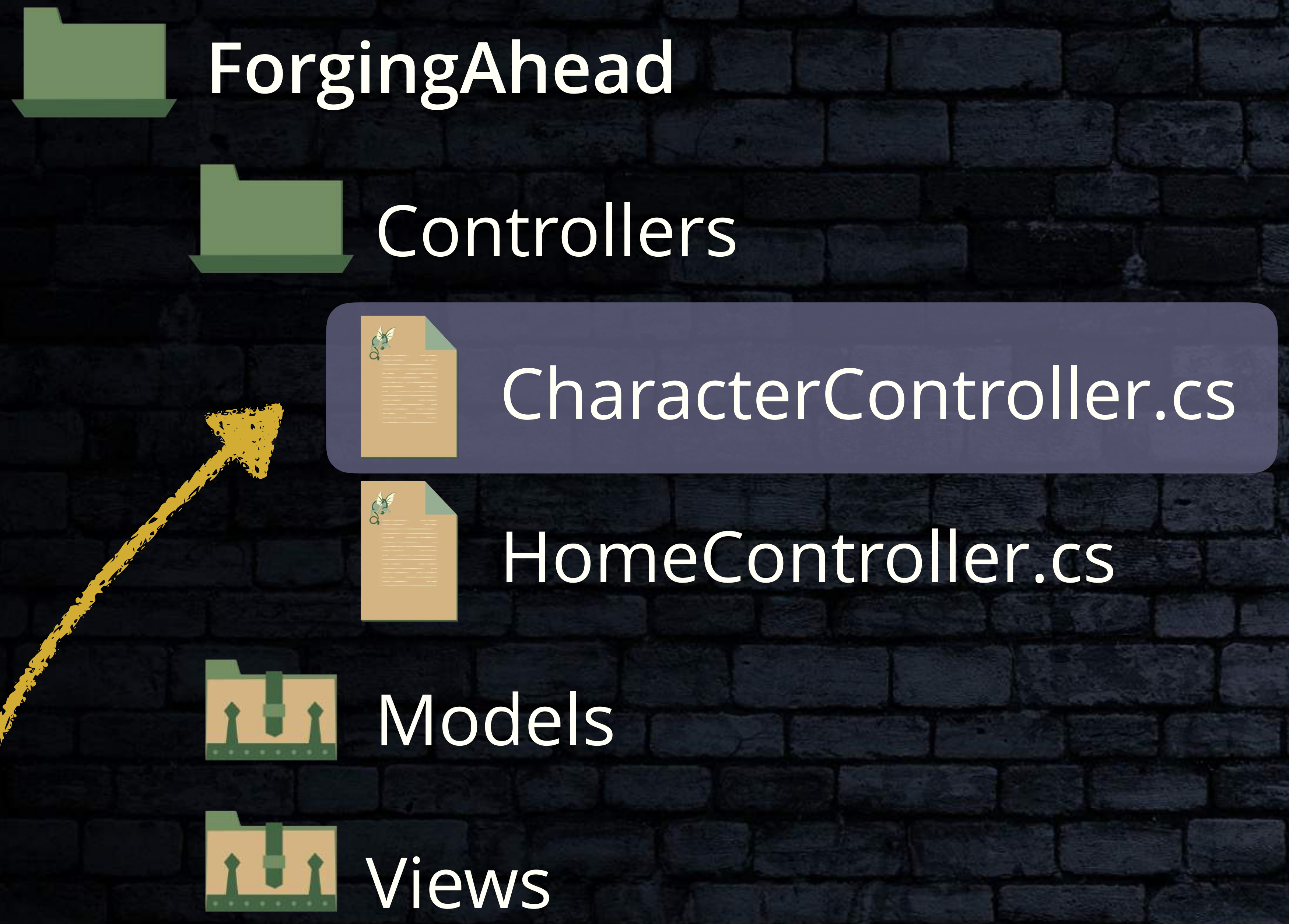
CS

```
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.



Create CharacterController.cs in our Controllers folder.

Controllers\CharacterController.cs

CS

```
using Microsoft.AspNetCore.Mvc;

namespace ForgingAhead.Controllers
{
    public class CharacterController : Controller
    {

    }
}
```

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

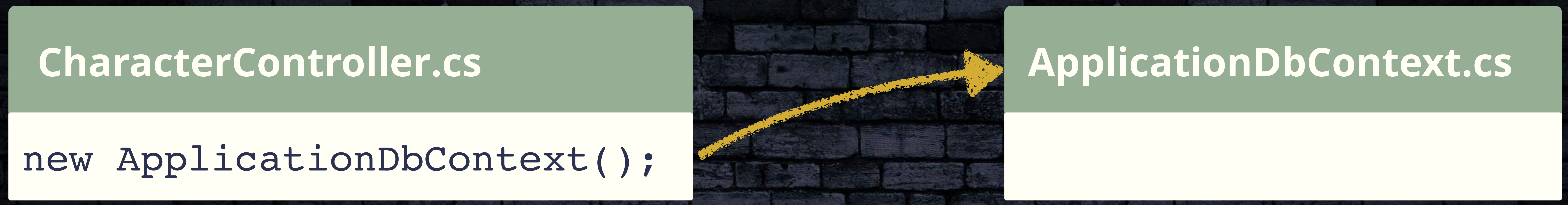
```
…
public class CharacterController : Controller
{
    private readonly ApplicationDbContext _context;

    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.

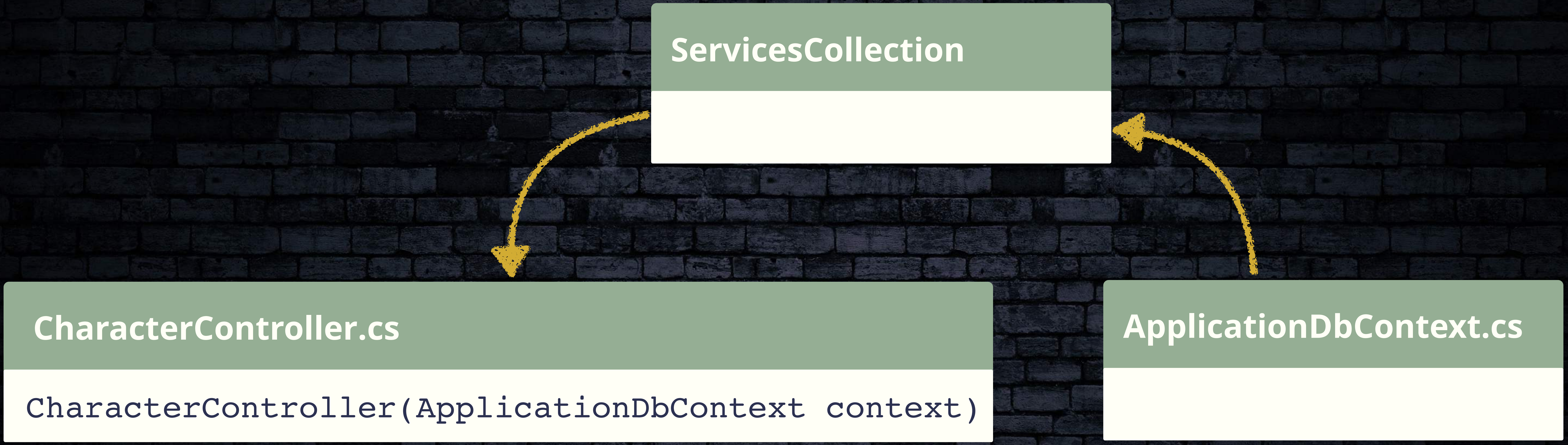


```
new ApplicationDbContext();
```

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.



```
CharacterController(ApplicationDbContext context)
```

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.

Collection using Where **method and lambda expression**

CS

```
…
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