Entity Framework

N-Tier Application with EF

Using Entity Framework to create a Multi-tier Project

What is Entity Framework

- Entity Framework is a common ORM framework used for .NET applications
    - ORM - Object Relational Mapping
    - Allows you query and manipulate data in a database using object-oriented paradigms
    - Every table is represented by a DTO class in the code
    - Allows you to create and populate tables as well as add/update/delete data all from your .NET code
    - No advanced SQL skills are needed.
- There are several ORM frameworks out there but we will be using Entity Framework
    - With EF, a table is referred to as a SET
    - A single record for the table (a row of data) is the "entity".
    - If you were building a customer management application the Customer table would be your "set"
    - Each customer would be an entity that is stored in that set.

Code First vs Database First

- There are two main ways to utilize EF to coordinate your database with your code.
- Database First
  - means you create the database, tables, and relationships using SSMS, then use entity framework to generate the DTOs and the code to work with the data.
  - everytime you change the database, you have to re-generate the code
  - you need to have a decent SQL skillset to ensure you create everything correctly
- Code First
  - you write your DTO’s in .NET, then use entity framework to build the database and the tables
  - if you need to add a new column to a table you just add a new property to your DTO and EF will update the table
  - this is the way the majority of new development is doing it
  - in fact, doing "database first" with .NET Core 6 requires multiple steps to get it set up
- We will be doing Code First

New D&D Management Project

- We are going to build a small application with a database back end that will be used to manage information for someone’s Dungeon and Dragons project
- What we build here will morph and grow as we learn new concepts in the future.
- We will start with a console front end with a data layer to connect to the database
- Eventually we will update to an MVC front end but we want to use the same data layer and domain layer
- After that, our MVC application will morph into an API application that will be used by an Angular web app we will create
- All of this will unfold in several weeks, but every iteration will need to use our data and database so it's important we get it set up the right way

Repository Project

Create the Data Layer Project

- Since we want to make sure that all future iteration of our application can use the code we are about to write, we will be creating a class library that other application can reference.
- "Create a new project" → C# Class Library
    - name:DungeonMasterRepository_2022 (update year accordingly)
    - .NET 6.0
    - Change the name of the solution → DungeonMaster_2021
    - "Create"
- Nuget Packages (latest stable)
    - Microsoft.Extensions.Configuration
    - Microsoft.Extensions.Configuration.FileExtensions
    - Microsoft.Extensions.Configuration.Json
    - Microsoft.EntityFrameworkCore
    - Microsoft.EntityFrameworkCore.Design
    - Microsoft.EntityFrameworkCore.SqlServer
    - Microsoft.EntityFrameworkCore.Tools

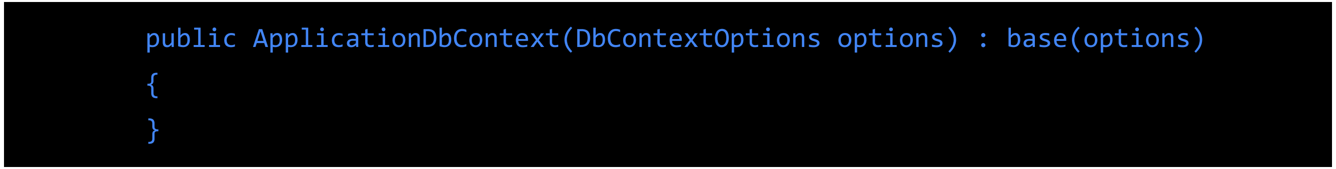
Create a DBContext

- To work with the database EF needs an object to act as the interpreter between the code and the database
  - This object is called the "DBContext"
    - defines the entity sets (tables)
    - allows the use of Fluent API to override the database schema
- Rename Class1.cs to "ApplicationDbContext.cs"
  - answer "yes" at the prompt
- Add a “using” statement for **Microsoft.EntityFrameworkCore**
- This class needs to inherit the **DBContext** class

Create a DBContext

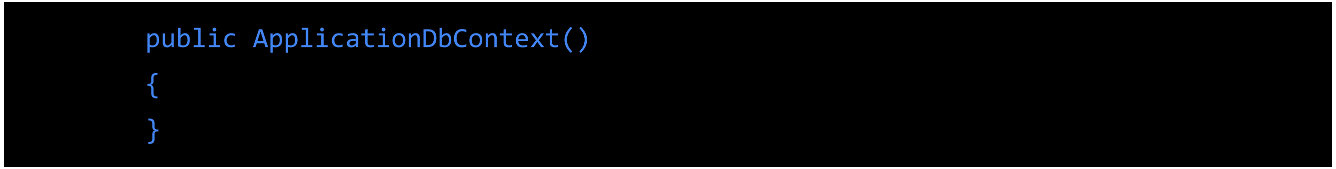
```
using Microsoft.EntityFrameworkCore;

namespace DungeonMasterRespository_2021
{
    public class ApplicationDbContext : DbContext
    {

    }
}
```

- DB Context needs an “options” object in order to work properly. We could spend a full day going down rabbit holes talking about the options builder, but right now we are just to utilize the boiler plate code that 95% of EF implementations utilize
- Our DB context needs a constructor that will pass in an options object



```
public ApplicationDbContext(DbContextOptions options) : base(options)
{
}
```

- If you remember from our Inheritance section this constructor will pass the options argument to the base class
- For database migrations to work, you also need the empty constructor



```
public ApplicationDbContext()
{
}
```

Connection String

- Now we need a configuration file to store the connection string to our database
- Right click the repository project → Add → New Item
- appsettings.json *(share the code over chat to ensure they have it right)*

```
{
  "ConnectionStrings": {
    "DungeonManager": "Data Source=DELLLAPTOP;Initial
Catalog=DungeonMaster_2022;Trusted_Connection=True;TrustServerCertificate=True;"
  }
}
```

- Go to the properties of the JSON file
    - Build Action = Content
    - Copy to Output = "Copy if Newer"

ConfigurationBuilderSingleton.cs

- Create the new class - ConfigurationBuilderSingleton
- using Microsoft.Extensions.Configuration;
- (share this code for them to paste)

```
public sealed class ConfigurationBuilderSingleton
  {
    private static ConfigurationBuilderSingleton _instance = null;
    private static readonly object instanceLock = new object();
    private static IConfigurationRoot _configuration;

    private ConfigurationBuilderSingleton()
    {
      var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);

      _configuration = builder.Build();
    }

    public static ConfigurationBuilderSingleton Instance
    {
      get
      {
        lock (instanceLock)
        {
          if (_instance == null)
          {
            _instance = new ConfigurationBuilderSingleton();
          }
          return _instance;
        }
      }
    }

    public static IConfigurationRoot ConfigurationRoot
    {
      get
      {
        if(_configuration == null)
        {
          var x = Instance;
        }
        return _configuration;
      }
    }
  }
```

Singleton Design Pattern

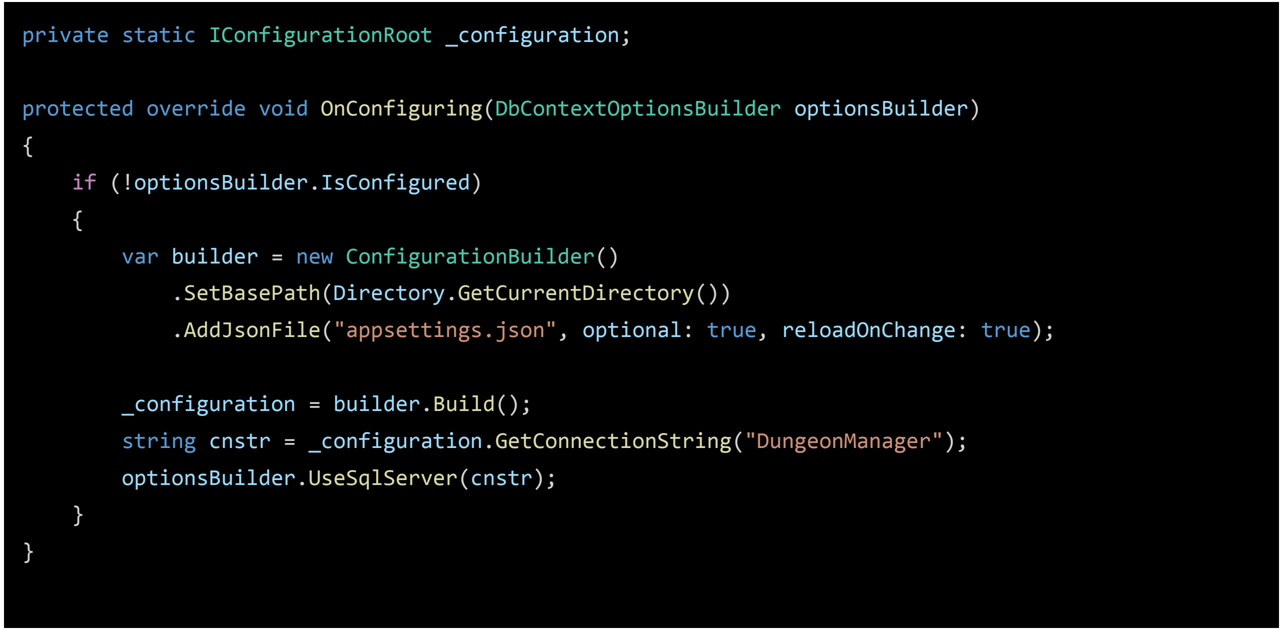
- Restrict the application to a "single" instance of a class
- This is very useful when exactly one object is needed to coordinate functionality across the application.
- A singleton will control it's own instantiations by
  - hide the constructor of the class
  - provide a public avenue to return the sole instance of the class
- You'll notice our class has a private constructor but two public read-only properties
- When application code tries to access the Instance property it will return the instance of the class, creating a new instance if one doesn't exist
- Singletons are very common in the case of Logging. You only one a single Logger instance to handle all of the logging in the application
- Many video games have a "game manager" singleton that will keep track of user settings and game details
- Our singleton is used to easily load our configuration file whenever we need to access information from it.

Enable Repository to use Code First

- In order to allow our repository to actually process the code and create the database objects we need to add some code to ApplicationDbContext
- Make sure you have these using statements
  - using Microsoft.EntityFrameworkCore;
  - using DungeonMasterDTO_2022;
  - using Microsoft.Extensions.Configuration;
  - using System.IO;

Enable Repository to use Code First

- Now add this code to ApplicationDbContext
  - *Paste the code to ensure everyone has it correct*

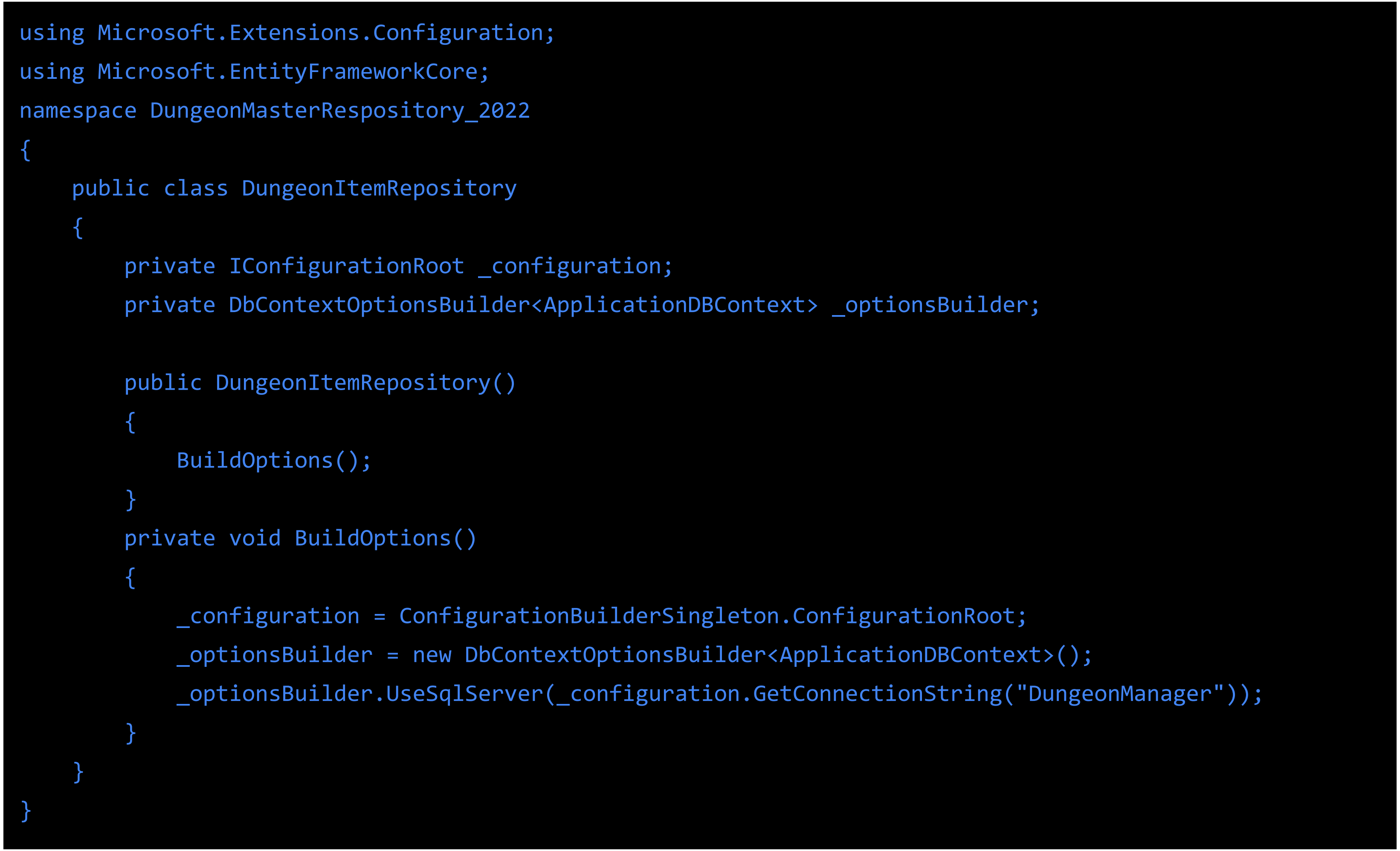


```
private static IConfigurationRoot _configuration;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);

        _configuration = builder.Build();
        string cnstr = _configuration.GetConnectionString("DungeonManager");
        optionsBuilder.UseSqlServer(cnstr);
    }
}
```

New Database

DungeonItemRepository.cs

- Create a new class DungeonItemRepository
- This class will be the Data Layer we use for Dungeon Items



```
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
namespace DungeonMasterRespository_2022
{
    public class DungeonItemRepository
    {
        private IConfigurationRoot _configuration;
        private DbContextOptionsBuilder<ApplicationDBContext> _optionsBuilder;

        public DungeonItemRepository()
        {
            BuildOptions();
        }
        private void BuildOptions()
        {
            _configuration = ConfigurationBuilderSingleton.ConfigurationRoot;
            _optionsBuilder = new DbContextOptionsBuilder<ApplicationDBContext>();
            _optionsBuilder.UseSqlServer(_configuration.GetConnectionString("DungeonManager"));
        }
    }
}
```

Create the database

- Now it's time to create our database
- open package manager console
  - Tools → Nuget Package Manager → Package Manager Console
- Make sure the **Default Project** is your repository
- **add-migration initial_setup_create_items_table**
- *view migration snapshot*
- Change the migration to use smaller nvarchar values
- SAVE THE MIGRATION
- create the database
  - update-database
- verify database and table creation in SSMS

Domain Project

Create the Domain Layer

- Right click the solution → add → new project
- "Create a new project" → Class Library → DungeonMasterDomain_2022 (update year accordingly)
- Add a reference to the Repository project
- Rename Class1 to DungeonItemInteractor

Console Project

Create the Console Application

- Right click the solution → add → new project
- "Create a new project" → Console App → DungeonMasterConsole_2022 (update year accordingly)
- Add a reference to the Domain project
- Add the following package
  - Microsoft.EntityFrameworkCore.Design

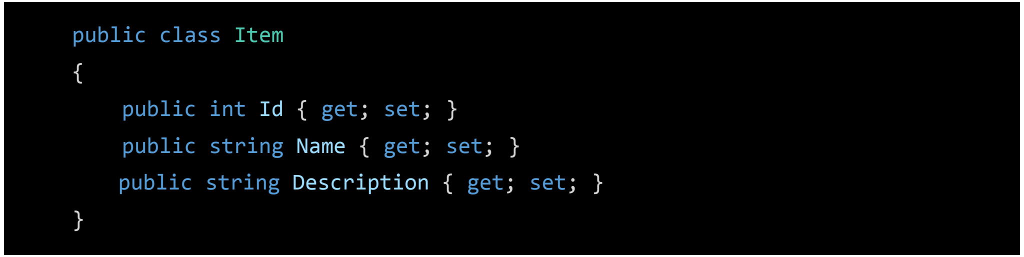
DTO Project

Create the Data Layer Project

- It is a good practice to store the data objects for your application in their own project.
- Allows reuse of the code in other projects
- Right click the solution → add → new project
- "Create a new project" → C# Class Library
    - name: DungeonMasterDTO_2022 (update year accordingly)
    - .NET 6.0
    - "Create"
- Have all other projects reference this one
- Rename Class1 to **Item**

Item Object

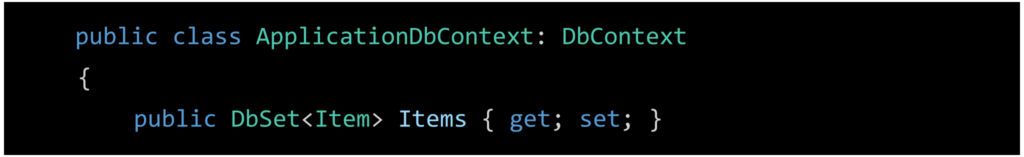
- Update Class1.cs to Item.cs



```
public class Item
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
```

Item DbSet

- Update your DB Context with a using statement to the DTO project
- If you want your EF solution to use a DTO as an entity and pair it with a table you need to use the **DbSet** property



```
public class ApplicationDbContext: DbContext
{
    public DbSet<Item> Items { get; set; }
```

  - This is telling EF that you need a table named Items in your database and that table will store multiple entities that match the Item DTO

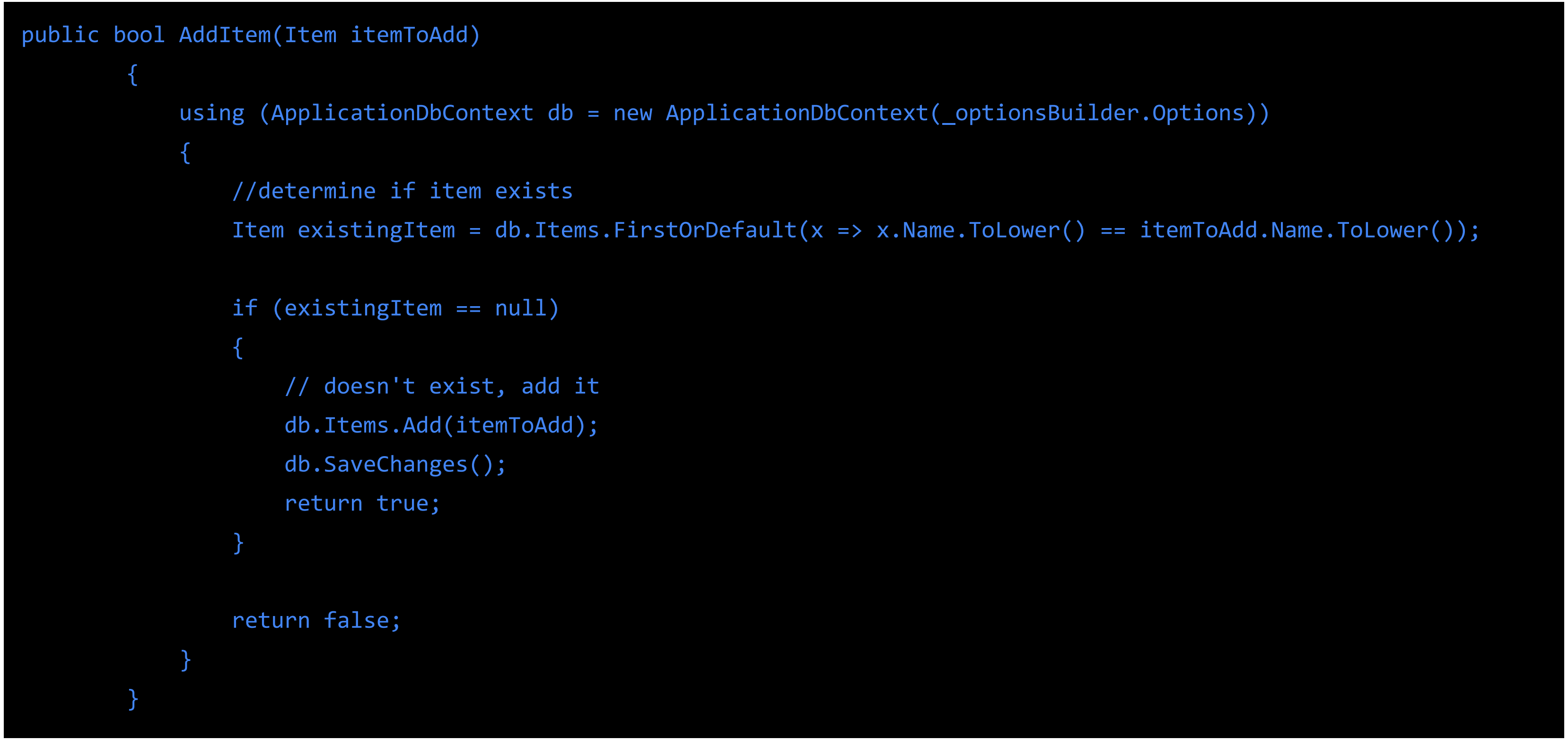
Adding Data

Repository Method

- We want our application to load some default data when it runs to insure we have information to work with.
- Our console application will initiate the process but proper design practice states that only the Repository should interact with the database
- To allow the addition of items we will add logic to all three of our layers
- We will start off by create an AddItem method in our **DungeonItemRepository**
- Make sure you have a using statement for your DTO project
  - using DungeonMasterDTO_2021;

Insert a list of items

- create the AddItem method



```
public bool AddItem(Item itemToAdd)
        {
            using (ApplicationDbContext db = new ApplicationDbContext(_optionsBuilder.Options))
            {
                //determine if item exists
                Item existingItem = db.Items.FirstOrDefault(x => x.Name.ToLower() == itemToAdd.Name.ToLower());

                if (existingItem == null)
                {
                    // doesn't exist, add it
                    db.Items.Add(itemToAdd);
                    db.SaveChanges();
                    return true;
                }

                return false;
            }
        }
```

Insert a list of items

- This code will check to see if an item with the same name exists
- If it does not, it will add it to the database
- Writing the code this way will allow you to run the application several times and never have to worry about creating duplicate records on start up

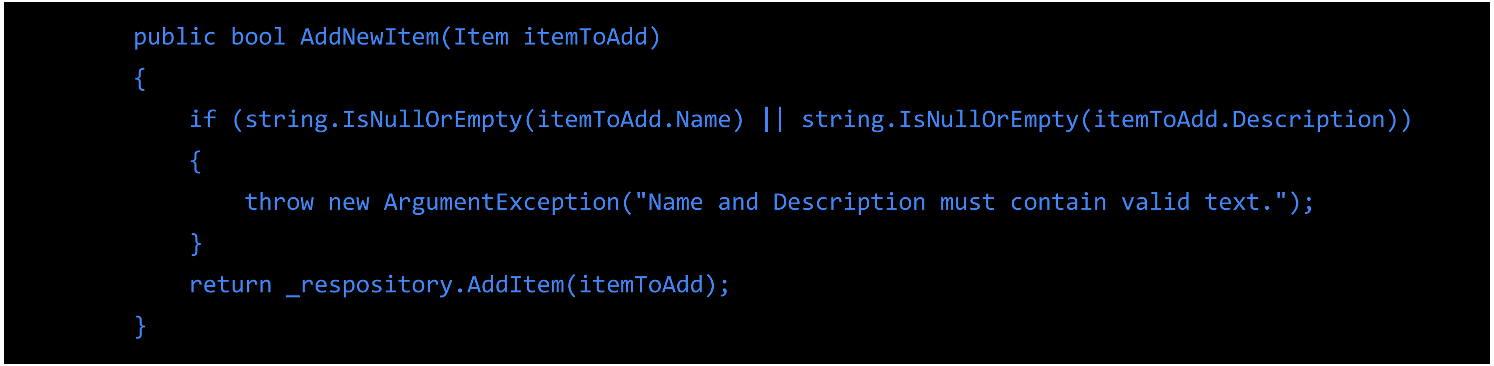
Interactor Method

- Following good architecture principles, we will have our interactor call the repository method. This way we can bake in any business rules or logic.
- In your DungeonItemInteractor class add as using statement for the DTO project
- Create a private repository object and have it instantiated in the constructor

```
private DungeonItemRepository _respository;
public DungeonItemInteractor()
{
    _respository = new DungeonItemRepository();
}
```

Interactor Method

- Create the **AddNewItem** method with some validation logic



```
public bool AddNewItem(Item itemToAdd)
{
    if (string.IsNullOrEmpty(itemToAdd.Name) || string.IsNullOrEmpty(itemToAdd.Description))
    {
        throw new ArgumentException("Name and Description must contain valid text.");
    }
    return _respository.AddItem(itemToAdd);
}
```

Console Method

- Now we can write the code in our Console project to utilize the interactor
- First create a method that builds a collection of Items

```
List<Item> BuildItemCollection()
{
    List<Item> initialItems = new List<Item>();
    initialItems.Add(new Item() { Name = "Common Arrow", Description = "A cheap wood arrow" });
    initialItems.Add(new Item() { Name = "Dull Sword", Description = "A very old sword" });
    initialItems.Add(new Item() { Name = "Ragged Tunic", Description = "It barely covers the important bits" });
    initialItems.Add(new Item() { Name = "Common Arrow", Description = "A cheap wood arrow" });
    initialItems.Add(new Item() { Name = "Dented Helm", Description = "What happened to the previous owner" });
    return initialItems;
}
```

Console Method

- Now create the method what will add each item

```
DungeonItemInteractor _dungeonItemInteractor = new DungeonItemInteractor();

void LoadStartUpData()
{
    foreach(Item item in BuildItemCollection())    {
        if (_dungeonItemInteractor.AddNewItem(item) == true)
        {
            Console.WriteLine($"{item.Name} was added to the database.");
        }
        else
        {
            Console.WriteLine($"{item.Name} was NOT added to the database.");
        }
    }
}
```

Console Method

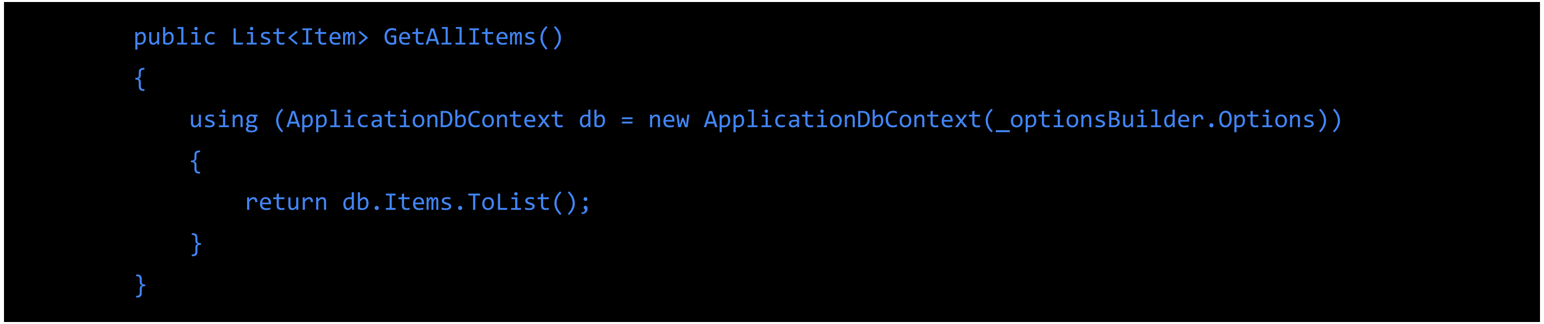
- Now add the code to call the **LoadStartupData** method

```
LoadStartUpData();

Console.WriteLine("Press any key to exit");
Console.ReadKey();
```

Retrieving Data

Repository Method

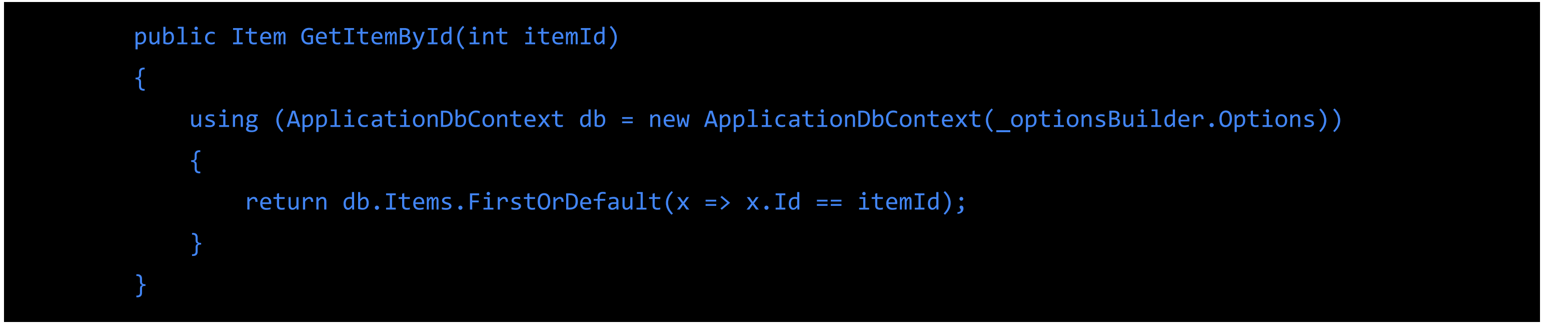
- Our repository should provide the ability to retrieve all the records in the table as well as filtering for a single record
- Start by creating the method in **DungeonItemRepository** that gets all records



```
public List<Item> GetAllItems()
{
    using (ApplicationDbContext db = new ApplicationDbContext(_optionsBuilder.Options))
    {
        return db.Items.ToList();
    }
}
```

Repository Method

- Now create the method to get a single Item



```
public Item GetItemById(int itemId)
{
    using (ApplicationDbContext db = new ApplicationDbContext(_optionsBuilder.Options))
    {
        return db.Items.FirstOrDefault(x => x.Id == itemId);
    }
}
```

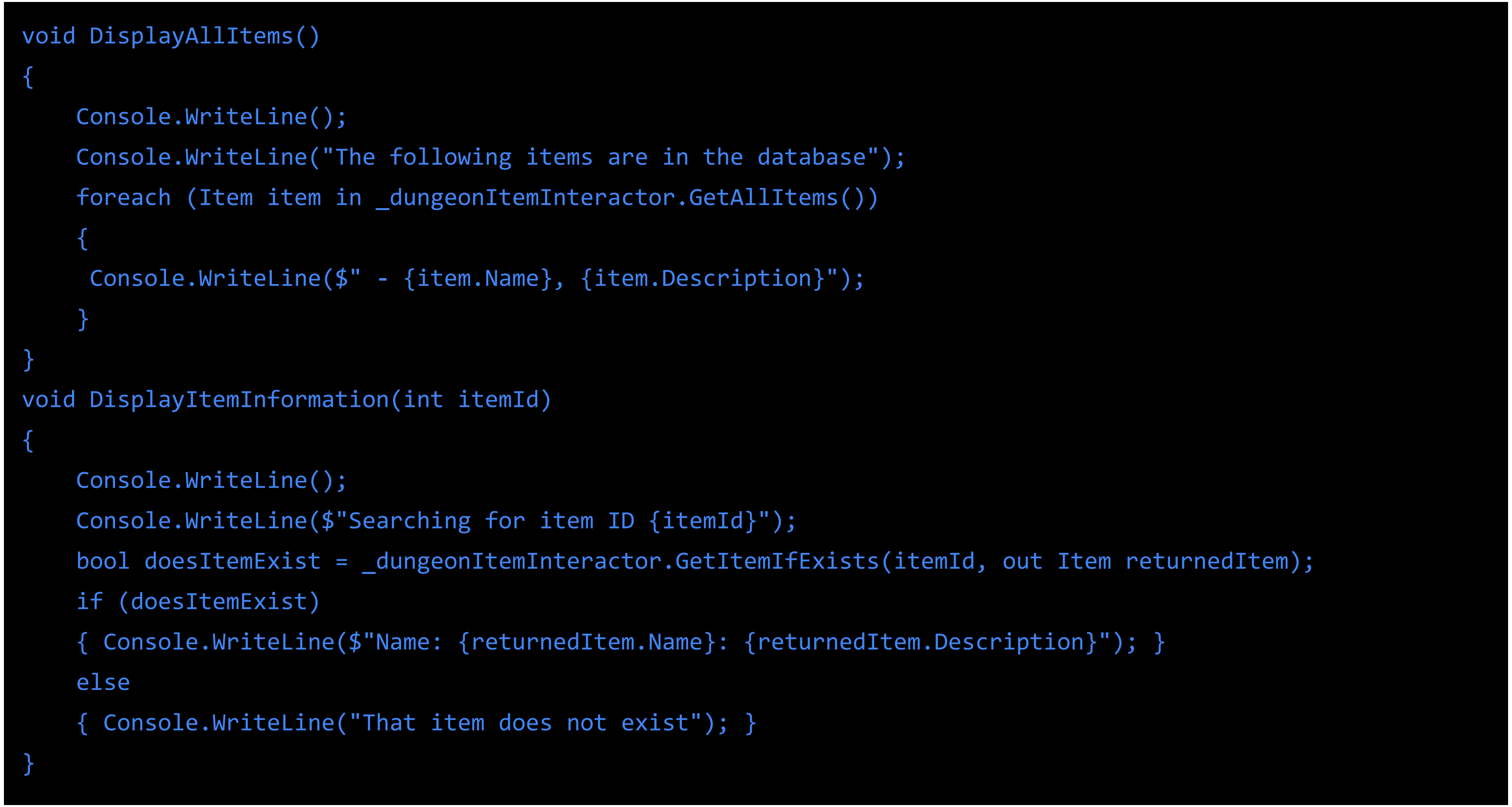
Interactor Method

- Now create the methods in **DungeonItemInteractor**

```
public List<Item> GetAllItems()
{
   return _respository.GetAllItems();
}

public bool GetItemIfExists(int itemId, out Item itemToReturn)
{
    Item item = _respository.GetItemById(itemId);
    itemToReturn = item;
    return itemToReturn != null;
}
```

Console Method

- Now create the methods in **Program.cs**



```
void DisplayAllItems()
{
    Console.WriteLine();
    Console.WriteLine("The following items are in the database");
    foreach (Item item in _dungeonItemInteractor.GetAllItems())
    {
     Console.WriteLine($" - {item.Name}, {item.Description}");
    }
}
void DisplayItemInformation(int itemId)
{
    Console.WriteLine();
    Console.WriteLine($"Searching for item ID {itemId}");
    bool doesItemExist = _dungeonItemInteractor.GetItemIfExists(itemId, out Item returnedItem);
    if (doesItemExist)
    { Console.WriteLine($"Name: {returnedItem.Name}: {returnedItem.Description}"); }
    else
    { Console.WriteLine("That item does not exist"); }
}
```

Console Method

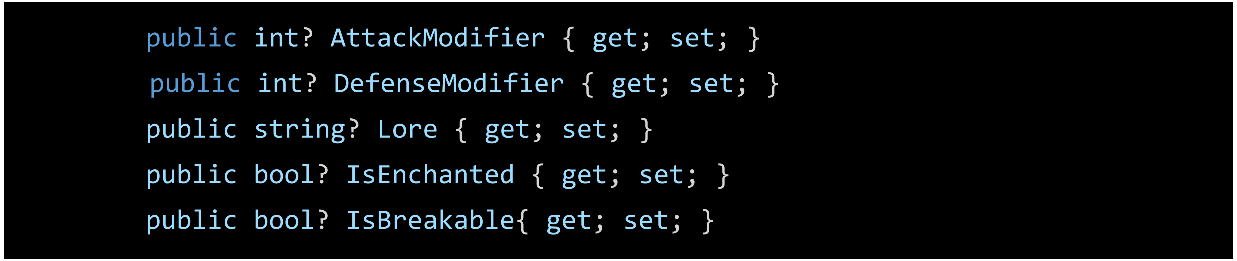
- Update **Program.cs** to call the new methods

```
DungeonItemInteractor _dungeonItemInteractor = new DungeonItemInteractor();
LoadStartUpData();
DisplayAllItems();
DisplayItemInformation(1);
DisplayItemInformation(10);
Console.WriteLine();
Console.WriteLine("Press any key to exit");
Console.ReadKey();
```

Modifying A DTO/Table

Add Properties to Item Entity

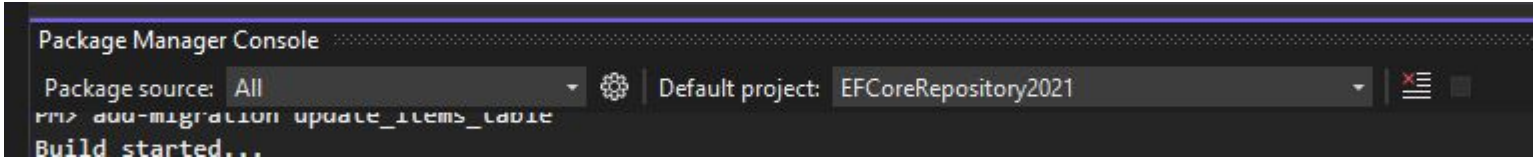
- Right now our Item DTOi s pretty scarce. Rarely will your tables be so small.
- Let's add a few properties to the Item to allow us to store more data
- Since we are added new columns to a table that already has data, we need to make sure our new properties allow null value. Otherwise the migration will fail
- For that we will use the nullable operator (?)



```
        public int? AttackModifier { get; set; }
        public int? DefenseModifier { get; set; }
        public string? Lore { get; set; }
        public bool? IsEnchanted { get; set; }
        public bool? IsBreakable{ get; set; }
```

Add Properties to Item Table

- Package Manager Console
  - -Make sure Repository project is selected as default project



Package source: All

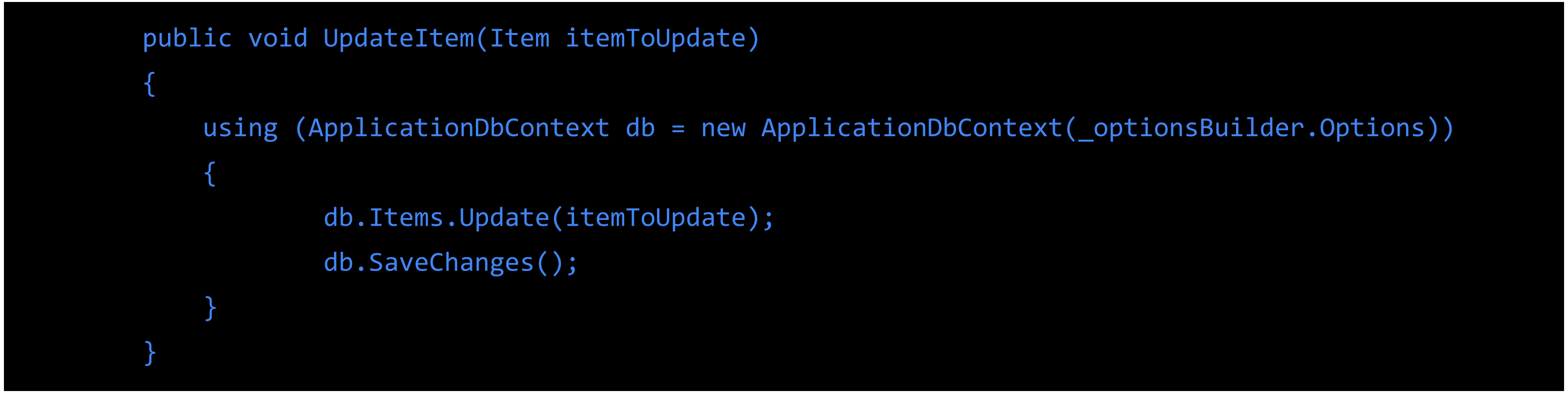
Default project: EFCoreRepository2021

```
PM> add-migration update_items_table
Build started...
```

  - **add-migration update_items_table**
  - Open migration to view the upcoming changes
  - **Since the console app it now our startup application it needs a copy of appsettings.json**
  - Drag the file to the console project
  - **update-database**
  - *verify in SSMS*

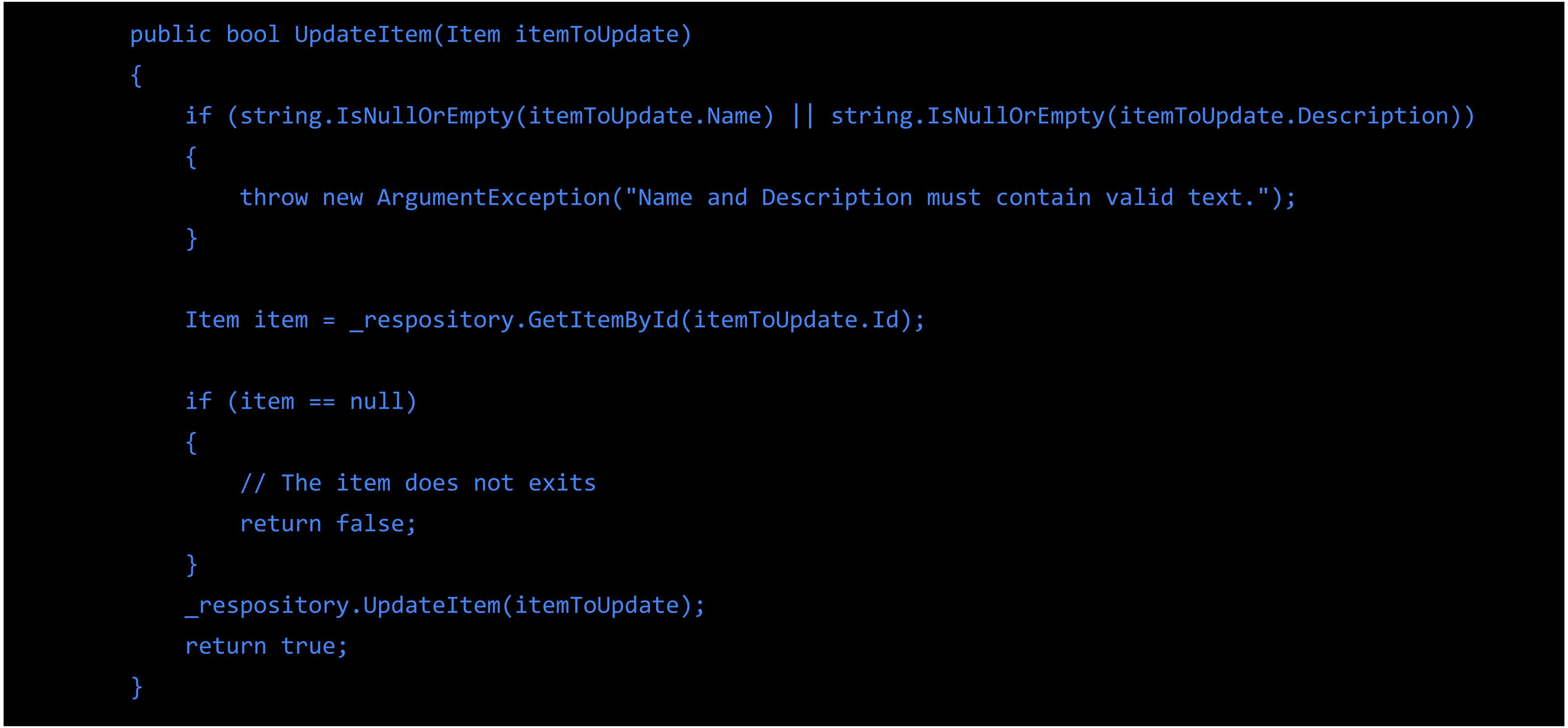
Update - Repository

- Now to write an UPDATE method that lets us update a record in our database



```
public void UpdateItem(Item itemToUpdate)
{
    using (ApplicationDbContext db = new ApplicationDbContext(_optionsBuilder.Options))
    {
            db.Items.Update(itemToUpdate);
            db.SaveChanges();
    }
}
```

Update - Interactor



```
public bool UpdateItem(Item itemToUpdate)
{
    if (string.IsNullOrEmpty(itemToUpdate.Name) || string.IsNullOrEmpty(itemToUpdate.Description))
    {
        throw new ArgumentException("Name and Description must contain valid text.");
    }

    Item item = _respository.GetItemById(itemToUpdate.Id);

    if (item == null)
    {
        // The item does not exits
        return false;
    }
    _respository.UpdateItem(itemToUpdate);
    return true;
}
```

Update - Console

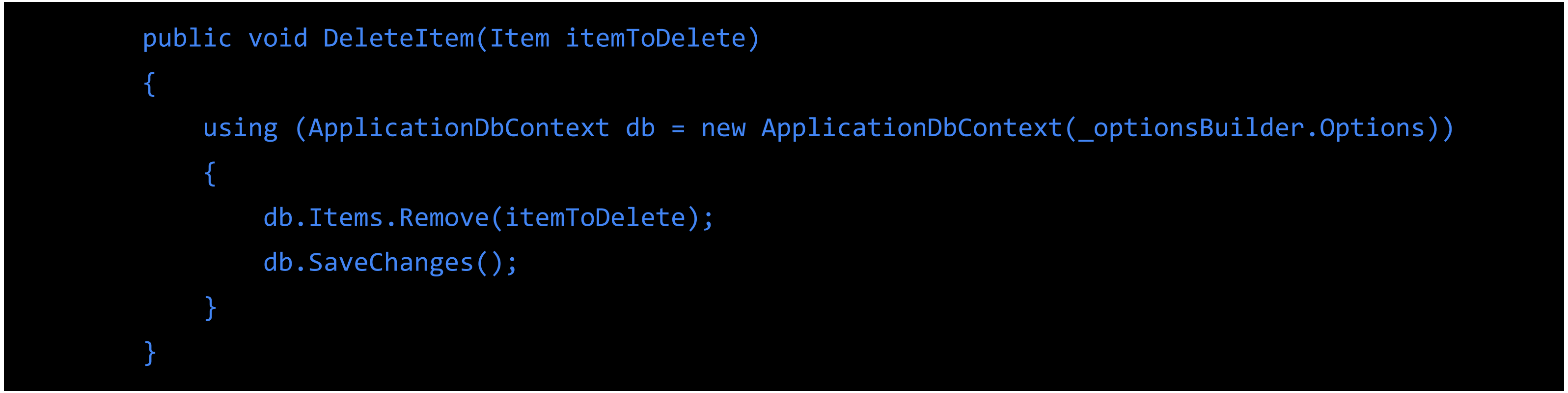
```
void UpdateItemModifiers(int itemId, int attackModifier, int defenseModifier )
{
    // Get the item to update
    if (_dungeonItemInteractor.GetItemIfExists(itemId, out Item returnedItem))
    {
        returnedItem.AttackModifier = attackModifier;
        returnedItem.DefenseModifier = defenseModifier;
        _dungeonItemInteractor.UpdateItem(returnedItem);
        Console.WriteLine($"{returnedItem.Name} was successfully updated to the database.");
    }
    else
    {
        Console.WriteLine($"There was a problem updating the record for Id {itemId}.");
    }
}
```

Update - Console

```
DungeonItemInteractor _dungeonItemInteractor = new DungeonItemInteractor();
LoadStartUpData();
DisplayAllItems();
DisplayItemInformation(1);
DisplayItemInformation(10);
Console.WriteLine();
UpdateItemModifiers(1,0,5);
UpdateItemModifiers(2,10,-1);
UpdateItemModifiers( 20, 10,10);

Console.WriteLine();
Console.WriteLine("Press any key to exit");
Console.ReadKey();
```

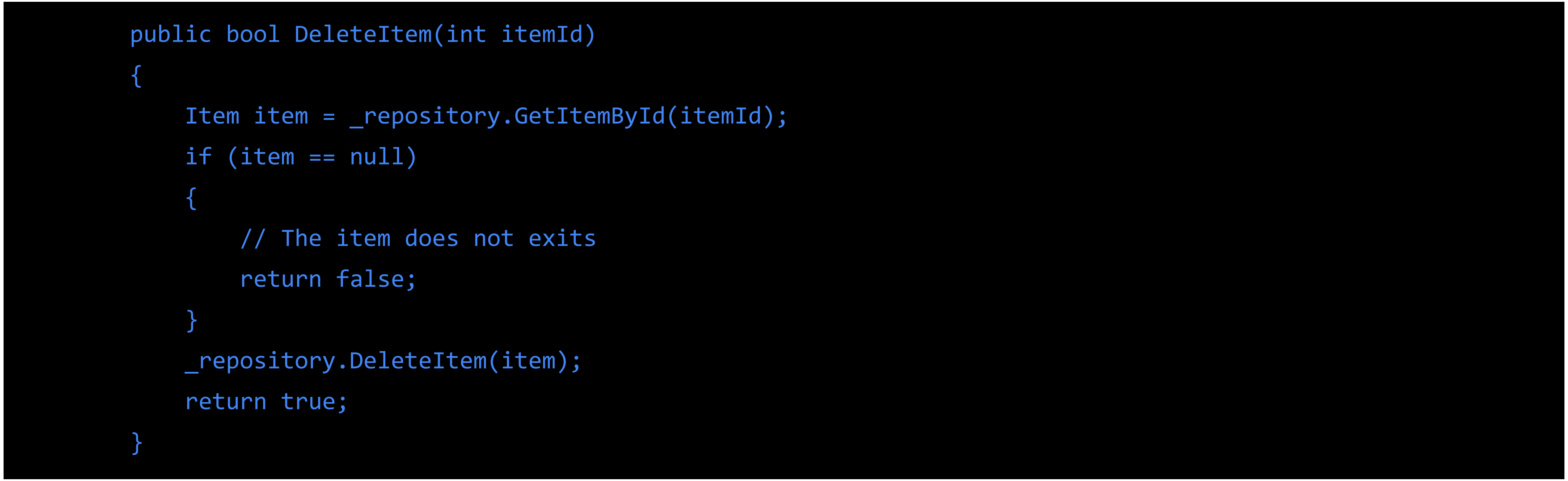
Delete - Repository

- Now to write an DELETE method that lets us delete a record in our database



```
public void DeleteItem(Item itemToDelete)
{
    using (ApplicationDbContext db = new ApplicationDbContext(_optionsBuilder.Options))
    {
        db.Items.Remove(itemToDelete);
        db.SaveChanges();
    }
}
```

Update - Interactor



```
public bool DeleteItem(int itemId)
{
    Item item = _repository.GetItemById(itemId);
    if (item == null)
    {
        // The item does not exits
        return false;
    }
    _repository.DeleteItem(item);
    return true;
}
```

Update - Console

```
void DeleteItem(int itemId)
{
    // Get the item to update
    if (_dungeonItemInteractor.DeleteItem(itemId))
    {
        Console.WriteLine($"Item ID {itemId} was successfully deleted from the database.");
    }
    else
    {
        Console.WriteLine($"There was a problem deleting the record for Id {itemId}.");
    }
}
```

Update - Console

```
DungeonItemInteractor _dungeonItemInteractor = new DungeonItemInteractor();
LoadStartUpData();
DisplayAllItems();
DisplayItemInformation(1);
DisplayItemInformation(10);
Console.WriteLine();
UpdateItemModifiers(1,0,5);
UpdateItemModifiers(2,10,-1);
UpdateItemModifiers(itemId: 20, 10,10);
Console.WriteLine();
DeleteItem(1);
DeleteItem(10);
Console.WriteLine();
Console.WriteLine("Press any key to exit");
Console.ReadKey();
```

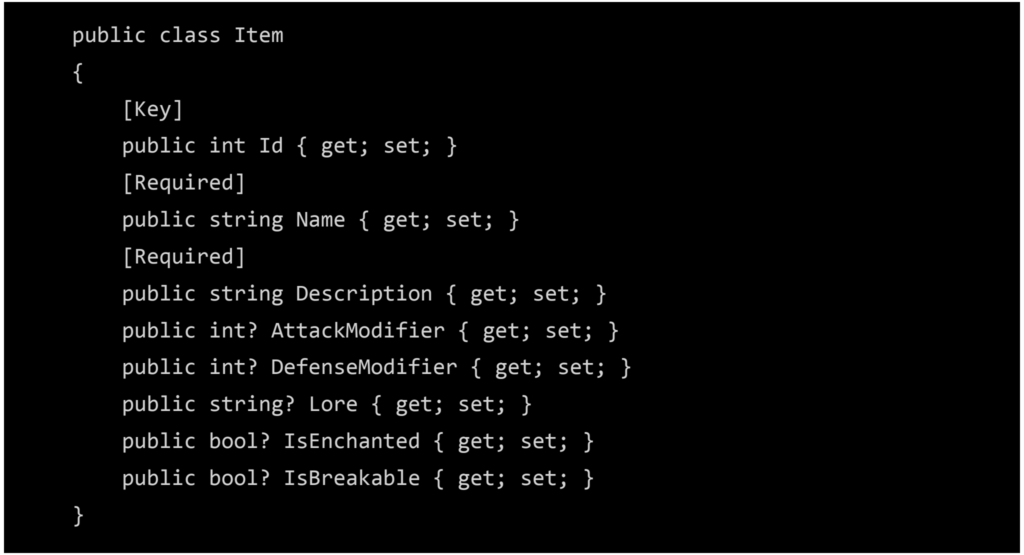
Data Annotations

- So far we have created our table with default functionality
- When using EF you have the ability to to add data annotation to your DTO to provide additional rules regarding your fields, these rules are generally called “contraints’
- To use data annotations you need the **using System.ComponentModel.DataAnnotations** statement
- VALUE CONSTRAINTS
  - [Required] should be used for any fields that will not accept null values.
  - [Range] can be used to create a min and max value for a field.
- DEFAULT VALUE
  - [DefaultValue] can be used to provide a default value for any missing field.
- Needs the using **System.ComponentModel**; statement
- In order to properly create relationships you need to make sure the Primary and Foriegn keys are set up properly in your DTO objects
- The [KEY]

Data Annotations

- So far we have created our table with default functionality
- When using EF you have the ability to to add data annotation to your DTO to provide additional rules regarding your fields, these rules are generally called “contraints’
- To use data annotations you need the **using System.ComponentModel.DataAnnotations** statement
- VALUE CONSTRAINTS
    - [Required] should be used for any fields that will not accept null values.
- KEY
    - [KEY] is used to guarantee that that your desired property is used as the primary key

Update Item Object

- 

```
public class Item
{
    [Key]
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Description { get; set; }
    public int? AttackModifier { get; set; }
    public int? DefenseModifier { get; set; }
    public string? Lore { get; set; }
    public bool? IsEnchanted { get; set; }
    public bool? IsBreakable { get; set; }
}
```

- Create migration: **add-migration update_items_contraints**
- Update database: **update-database**

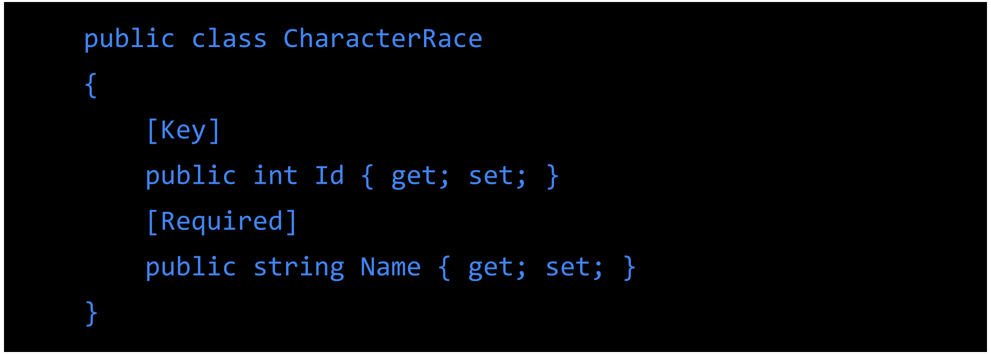
Peer Programming

- Use Entity Framework to create a Character
- Create the necessary, Repository, Interactor, and Console methods

Relationship Table

One to Many - Table

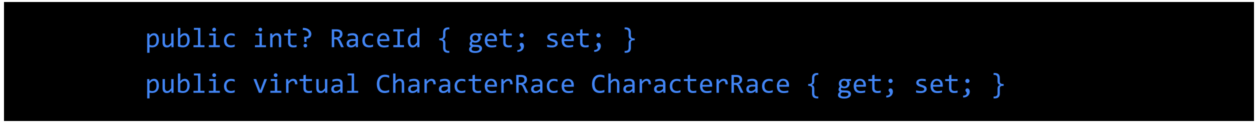
- A DnD character usually has a race as one of its attributes. Since many characters can have the same race we don’t want to store the value of the race in the character table
- A better option would be to have a character race table and relate it to the character table
- To do this, create the DTO for your CharacterRace table make sure it has the following using statement **System.ComponentModel.DataAnnotations**;



```
public class CharacterRace
{
    [Key]
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
}
```

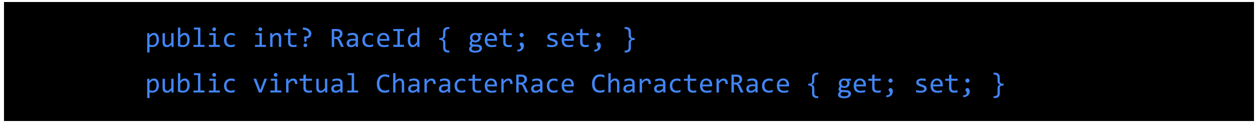
One to Many - Table

- Now update the Character DTO with the properties needed for the Foriegn Key
- You will need to make it nullable since there already data in the table



```
public int? RaceId { get; set; }
public virtual CharacterRace CharacterRace { get; set; }
```

- Update the Race DTO for the other side of the relationship



```
public int? RaceId { get; set; }
public virtual CharacterRace CharacterRace { get; set; }
```

- Create Migration: add-migration add_characterrace_table
- Update Database: update_database

Many To Many - Table

- Now that we have a the Character and the Inventory tables we will create a table to store the “Many to Many” relationships between the to
- This means that a single character can have many items and a single item can be in the inventory of many characters
- This table will only have three fields, RelationshipId, CharacterId, and ItemId.
- Although it it only has three properties the DTO needs a little more to create the foreign key relationships to the other tables
-

DTO CharacterItemRelationship

```
public class CharacterItemRelationship
{
    [Key]
    public int RelationshipId { get; set; }
    [Required]
    public int CharacterId { get; set; }
    public virtual Character Character { get; set; }
    [Required]
    public int ItemId { get; set; }
    public virtual Item Item { get; set; }
}
```

Update other DTO’s

- Character DTO

```
 public virtual List<CharacterItemRelationship> Relationships { get; set; } = new
List<CharacterItemRelationship>();
```

- Item DTO

```
 public virtual List<CharacterItemRelationship> Relationships { get; set; } = new
List<CharacterItemRelationship>();
```

- Update the DB Context - **public DbSet<CharacterRace> CharacterRaces { get; set; }**
- Create Migration: add-migration add_characterrelation_table
- Update Database: update_database