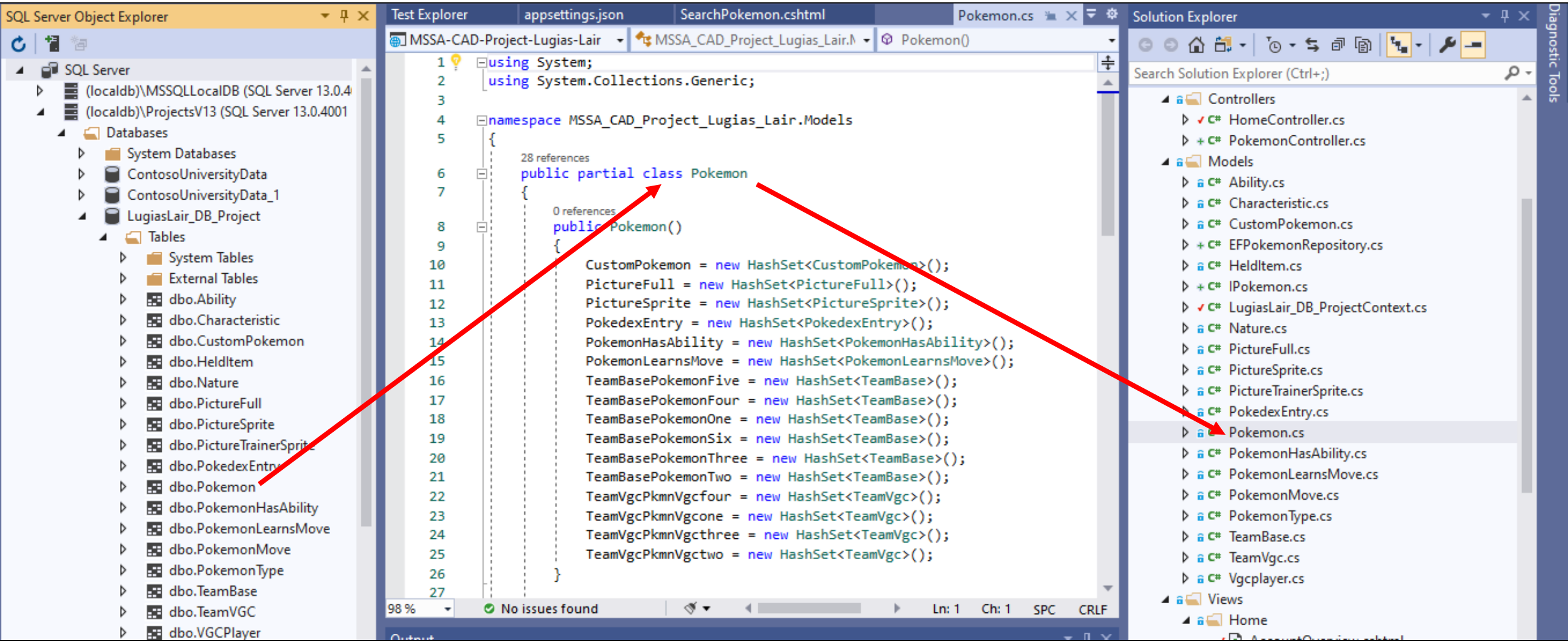


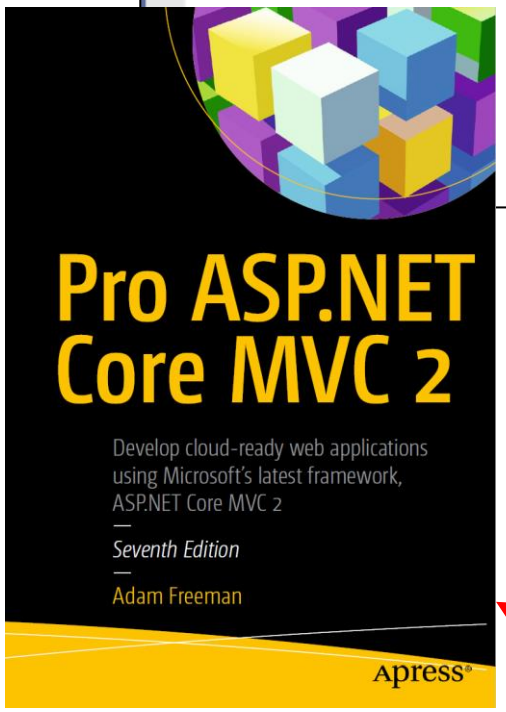
As a refresher, we used our SQL scripts to make a localDB within Visual Studio. Then we used Entity Framework to scaffold the model classes for the web application. But there are a lot of additional steps needed before you can display the data on a webpage.

```
MSSA-CAD-Project-Lugias-Lair  |  MSSA_CAD_Project_Lugias_Lair.Mo  |  Pokemon

1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Threading.Tasks;
5
6   namespace MSSA_CAD_Project_Lugias_Lair.Models
7   {
        6 references
8       public interface IPokemon
9       {
            3 references
10          IQueryable<Pokemon> Pokemon { get; }
11      }
12  }
13
```

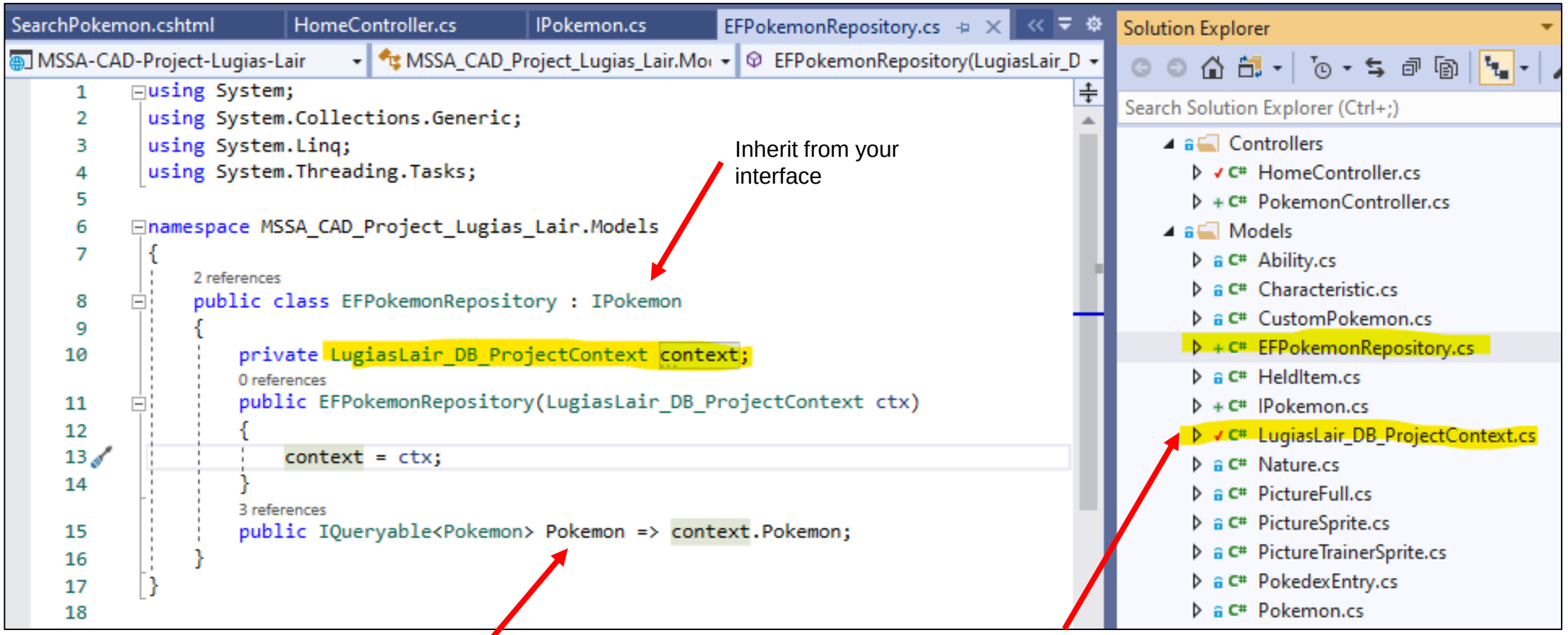
Search Solution Explorer (Ctrl+;)

- Controllers
  - HomeController.cs
  - PokemonController.cs
- Models
  - Ability.cs
  - Characteristic.cs
  - CustomPokemon.cs
  - EFPokemonRepository.cs
  - HeldItem.cs
  - IPokemon.cs
  - LugiasLair_DB_ProjectContext.cs
  - Nature.cs
  - PictureFull.cs
  - PictureSprite.cs
  - PictureTrainerSprite.cs
  - PokedexEntry.cs
  - Pokemon.cs



The first step after completing the Entity Framework scaffolding is to make an Interface that will let you receive a sequence of objects from the database. It is important that you specify that the property in the interface is “IQueryable” so that later on you can write queries to retrieve only the data you want to display/edit etc. See page 201 for more info.

This is the version of the book I am referring to, if you want to use the same page number references.

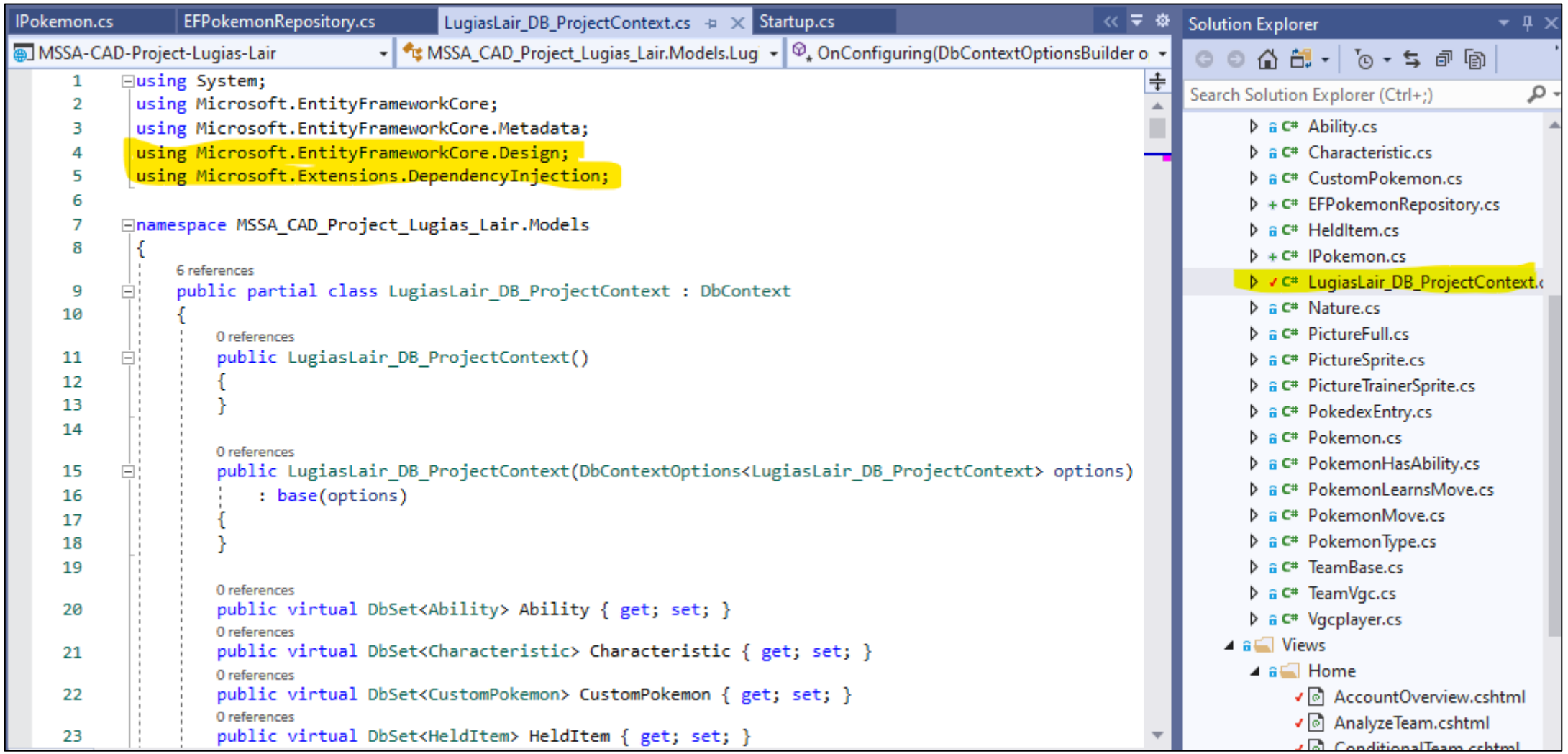


Iqueryable property implemented from the interface that will allow you to query the context data received from the database

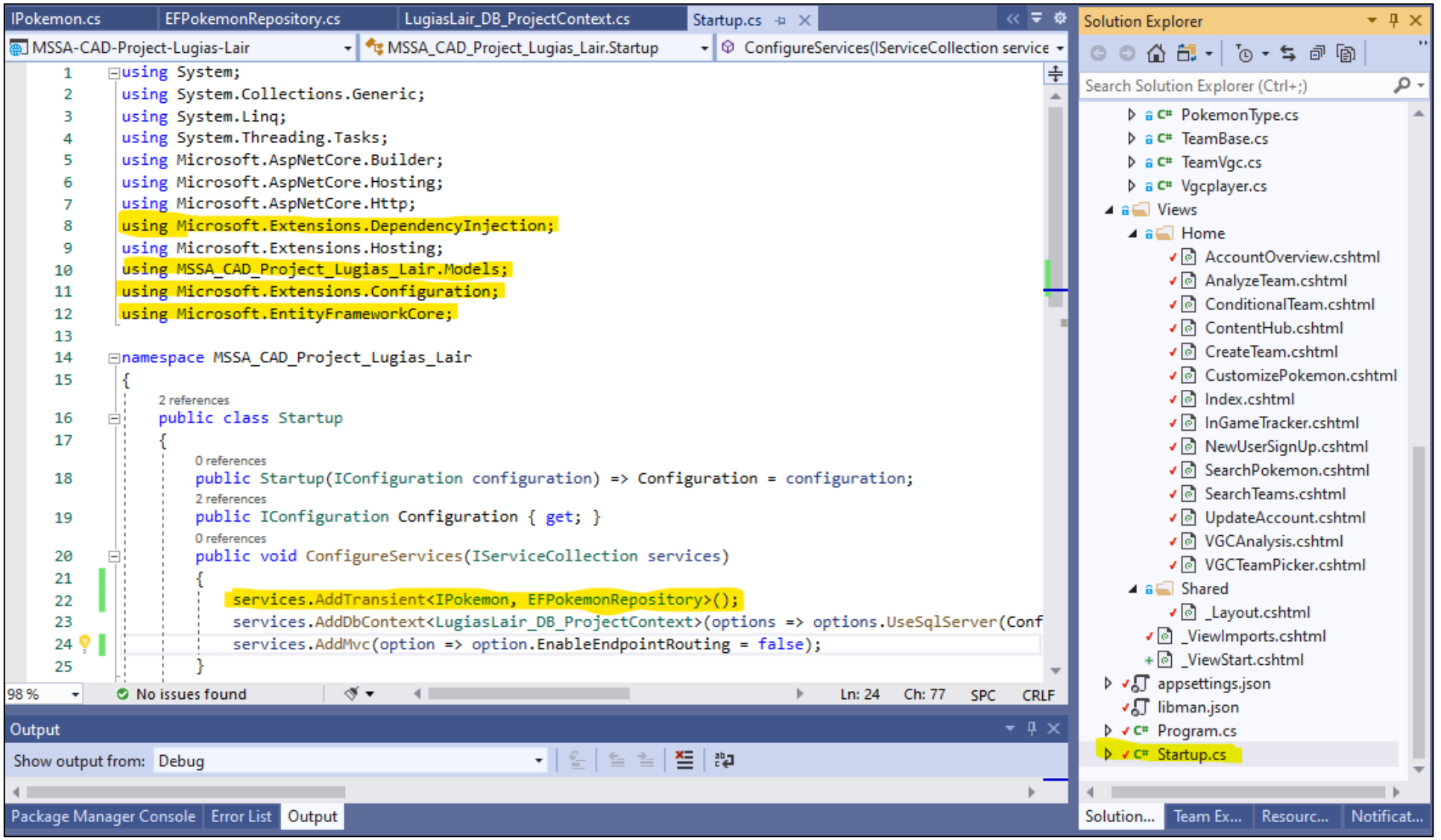
DB context class created by Entity Framework. Make sure to use whatever your file name is.

Note: The DB context class may be underlined in red in the repository class until you register it in the startup file, which is coming up next

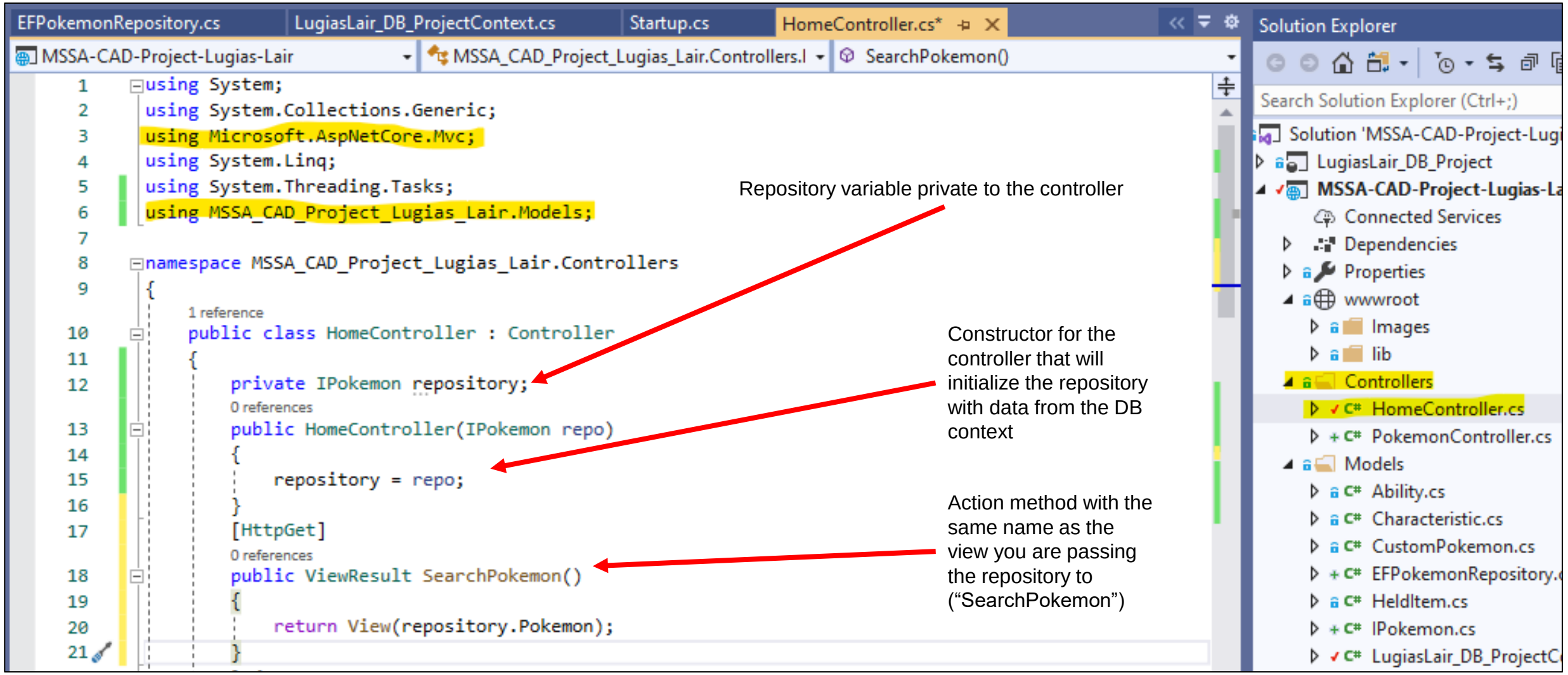
After making the interface, you need to make a repository. The repository needs to inherit from your interface so that it can query the objects it stores. The repository class have a private field that stores the context information from your database. When you performed Entity Framework scaffolding, the DB context class should have been created alongside the rest of your model classes. You then must write a constructor for the repository class that takes an instance of the DB context as an input parameter. See page 210 and 211 for more information (page 202 also demonstrates creating a repository class, but it is the “fake” repository that the author later removes when he executes the Entity Framework process, so I think it is best to refer to pages 210-211).



Ensure that you add the highlighted two using directives to your DB context class; they are not automatically included when you do Entity Framework scaffolding (see the top of page 210).



Now that you have the interface and repository created, you need to register them as transient services in the Startup file. This enables you to make changes to the repository without having to update other classes that reference the repository. Also make sure that you have all of the appropriate using directives (highlighted above). See pages 203, 212-213 for more information (as before, the example on pg. 202 deals with the fake repository, which is later replaced).



In order to display/manipulate the data that is stored in the repository, you need to use a controller to pass the data to a view. Within you chosen controller, make a private repository field that has the same type as the interface (MVC will supply whichever repository is indicated in the services part of the Startup file to satisfy this requirement). The write a constructor for the controller that takes an instance of the repository as an input parameter. Next, write an action method that will pass the data stored in the controller's repository to the view. The name of the action method should match the view to which you are passing the data. And because the repository can theoretically hold any data from your database, post-fix the repository variable with the name of the class you are interested in (in this case, the "Pokemon" class). See pages 204-205 for more details. *Note: Eventually you should have separate controllers for each major type of data, but for this example I am doing everything with the default Home controller.*

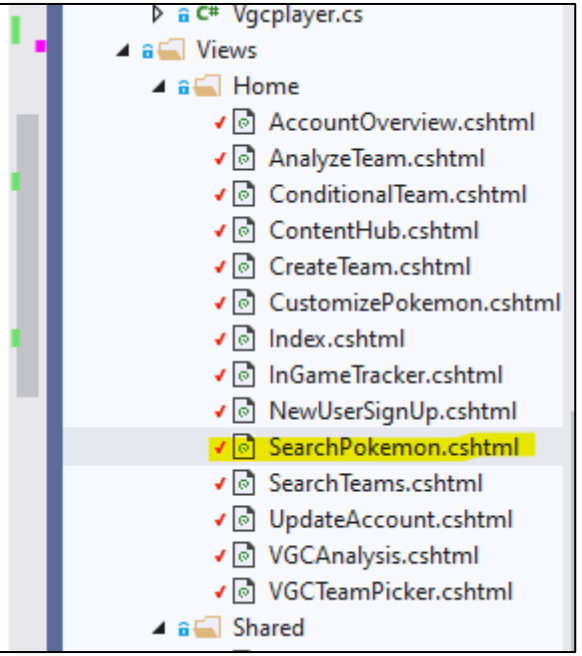
SearchPokemon.cshtml | LugiasLair_DB_ProjectContext.cs | Startup.cs | HomeController.cs*

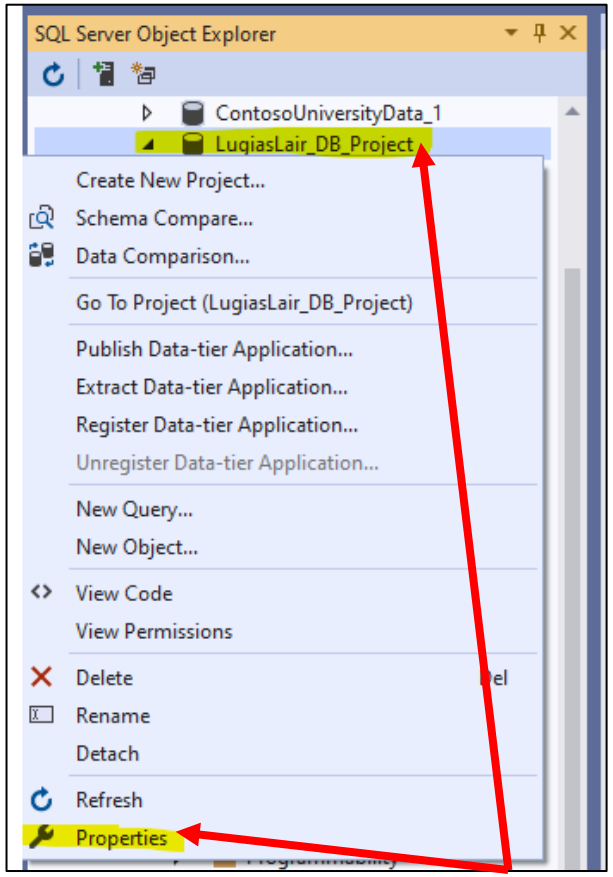
```
1   @model IEnumerable<Pokemon>
2
3   <header>
4       <h1>Pokemon Lookup</h1>
5       <p>This will be the page where users look-up a single Pokemon</p>
6   </header>
```

In the view file that is going to receive the data from the controller, add a model binding statement so that the view knows what kind of data it is receiving. In this case, the controller is passing a collection of IEnumerable Pokemon (since the repository uses IQueryable, the data is automatically able to be enumerated). See page 206 for more info. *Note: I know it should be possible to use a class name as the model for a view, but I have not figured that out yet)*.

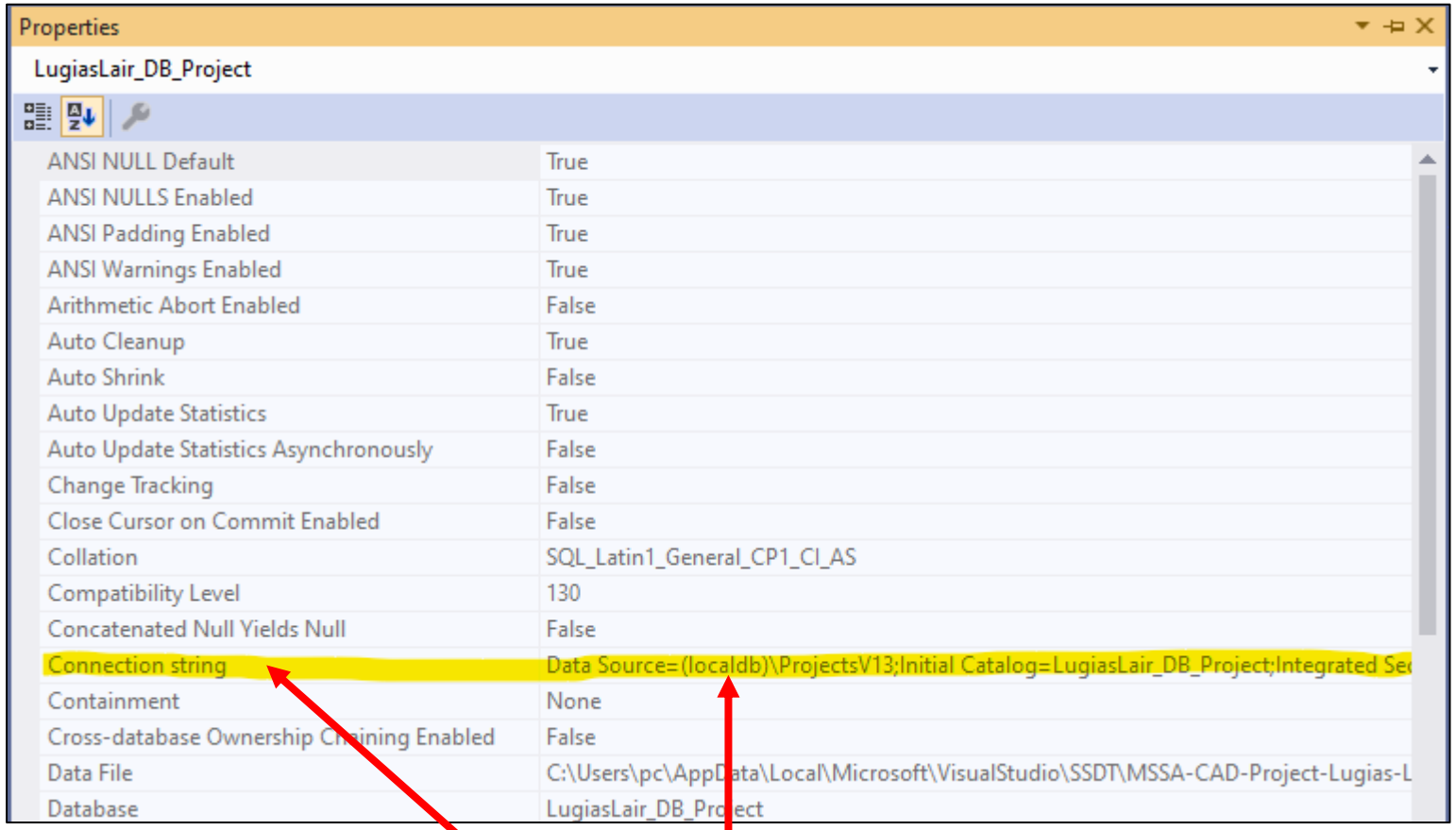
Now that the view knows what type of data it will receive, you can manipulate the display of that data by using razor expressions and simply referencing the collection by using the keyword “Model”. Note: if you have a class name as the model reference at the top of the view page, the way you address the data will look different (i.e. you’ll have to use “dot” notation such as “Model.Pokemon”).

```
@foreach (var p in Model)
{
    <tr>
        <td>@p.PokemonId</td>
        <td>@p.PokemonNumber</td>
        <td>@p.PokemonName</td>
        <td>@p.PokemonTypeOne</td>
        <td>@p.PokemonTypeTwo</td>
        <td>@p.PokemonStatTotal</td>
        <td>@p.PokemonHitPoints</td>
        <td>@p.PokemonAttack</td>
        <td>@p.PokemonDefense</td>
        <td>@p.PokemonSpecialAttack</td>
        <td>@p.PokemonSpecialDefense</td>
        <td>@p.PokemonSpeed</td>
    </tr>
}
</tbody>
</table>
```

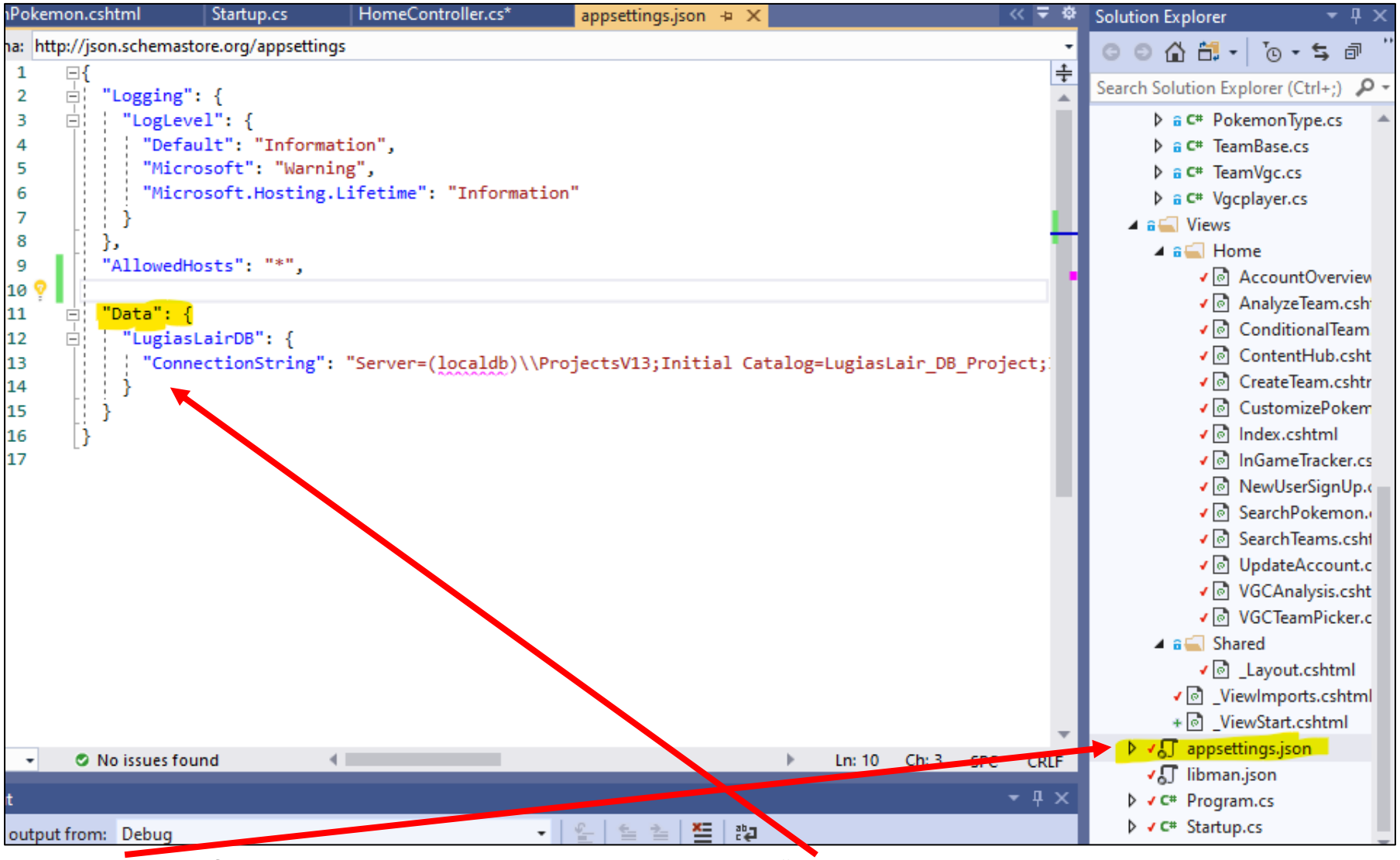




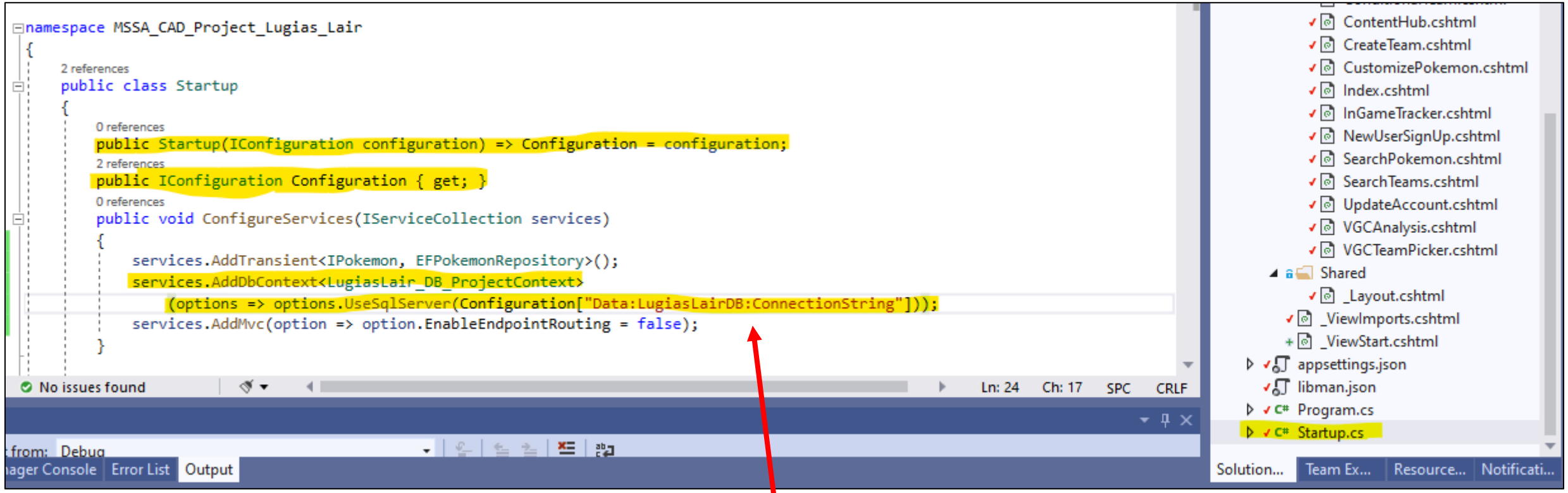
Now you need to actually connect to the database. Even though you conducted Entity Framework Scaffolding, the MVC application is not automatically connected to your lacalDB. Go to SQL Server Object Explorer and right-click on the database, then select “Properties.”



In the Properties window, copy the entire connection string. It will not fit in the view of the window, so make sure you highlight the entire thing.



Now open your appsettings.json file to add the connection string. Add a “Data” section, and then give the give the connection string a name based on your database (mine is “LugiasLairDB”). The ‘ConnectionString” is the actual value that the JSON file will associate with the key of the named connection string. This is where you copy + paste the entire connection string you got from the previous step. See page 211 for more details. *Note: when you copy the connection string from SQL Server Object Explorer, it will start with “Data Source”. I changed that to say “Server” in order to match the book; I am not sure if that has any effect, but it worked in my application.*

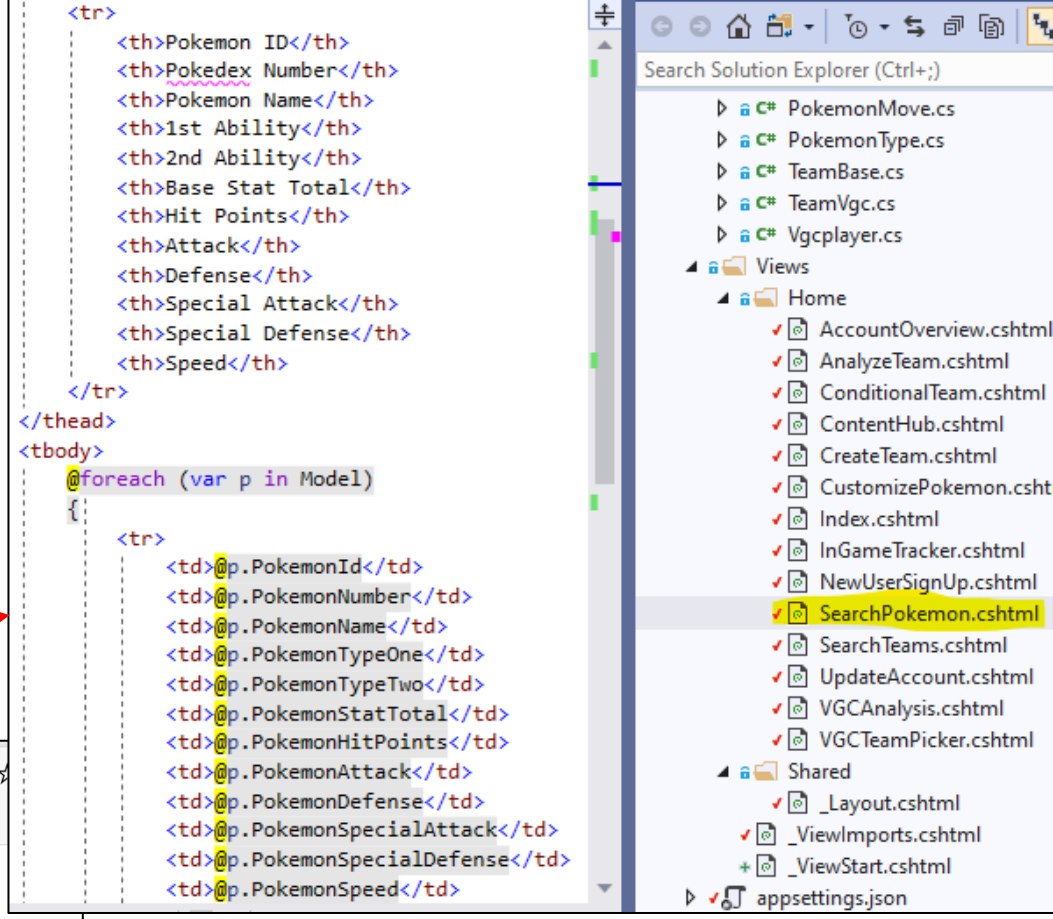


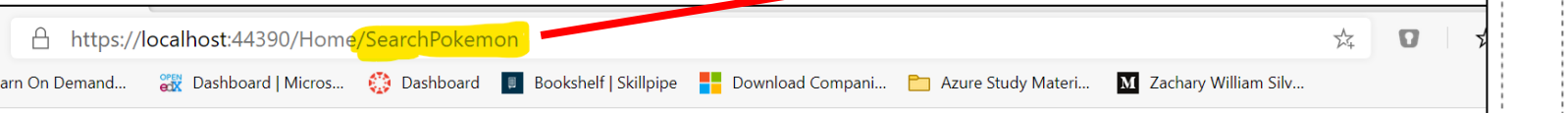
The final major change if to adjust the Startup file so that it uses the data connection configuration. (Make sure that you already have the Microsoft.Extensions. Configuration and Microsoft.EntityFrameworkCore using directives applied to the Startup file, which were highlighted on an earlier slide). You first add a Startup constructor that takes an IConfiguration object as an input parameter. Then add a public IConfiguration property. Finally, in the ConfigureServices method, add a service that adds a DB context and your local DB to provide that contect. This is where you enter the connection string info from the appsettings.json file (but just the name of the connection string, not the entire string itself). See page 212 – 213 for more details.

Note: There is a section on disabling scope verification in the Program file that starts at the bottom of 213 and ends on 214. When I tried to add that to my project the application would not launch. At this time, I do not think it is necessary.

Now the data can flow properly from the database to the view. Note that I am not actually searching/querying the data in this screenshot, just displaying all of the Pokemon in the database. I have not yet implemented true search or modification functionality.

```
    <tr>
        <th>Pokemon ID</th>
        <th>Pokedex Number</th>
        <th>Pokemon Name</th>
        <th>1st Ability</th>
        <th>2nd Ability</th>
        <th>Base Stat Total</th>
        <th>Hit Points</th>
        <th>Attack</th>
        <th>Defense</th>
        <th>Special Attack</th>
        <th>Special Defense</th>
        <th>Speed</th>
    </tr>
</thead>
<tbody>
    @foreach (var p in Model)
    {
        <tr>
            <td>@p.PokemonId</td>
            <td>@p.PokemonNumber</td>
            <td>@p.PokemonName</td>
            <td>@p.PokemonTypeOne</td>
            <td>@p.PokemonTypeTwo</td>
            <td>@p.PokemonStatTotal</td>
            <td>@p.PokemonHitPoints</td>
            <td>@p.PokemonAttack</td>
            <td>@p.PokemonDefense</td>
            <td>@p.PokemonSpecialAttack</td>
            <td>@p.PokemonSpecialDefense</td>
            <td>@p.PokemonSpeed</td>
```





Search Results

Pokemon ID	Pokedex Number	Pokemon Name	1st Ability	2nd Ability	Base Stat Total	Hit Points	Attack	Defense	Special Attack	Special Defense	Speed
1	1	Bulbasaur	5	8	318	45	49	49	65	65	45
2	2	Ivysaur	5	8	405	60	62	63	80	80	60
3	3	Venusaur	5	8	525	80	82	83	100	100	80
4	3	Mega Venusaur	5	8	625	80	100	123	122	120	80
5	4	Charmander	2		309	39	52	43	60	50	65
6	5	Charmeleon	2		405	58	64	58	80	65	80
7	6	Charizard	2	10	534	78	84	78	109	85	100
8	6	Mega Charizard X	2	15	634	78	130	111	130	85	100
9	6	Mega Charizard Y	2	10	634	78	104	78	159	115	100
10	7	Squirtle	3		314	44	48	65	50	64	43
11	8	Wartortle	3		405	59	63	80	65	80	58
12	9	Blastoise	3		530	79	83	100	85	105	78
13	9	Mega Blastoise	3		630	79	103	120	135	115	78