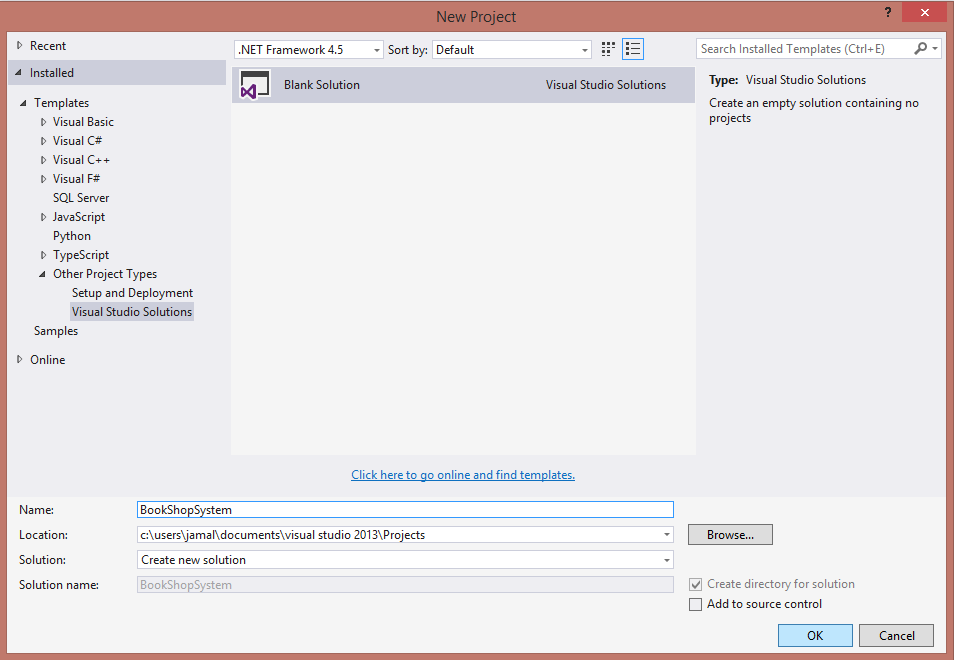
# Exercise: EF Code First - Book Shop

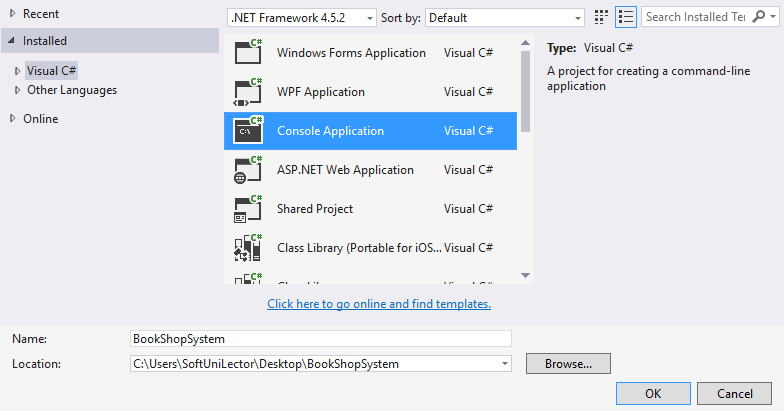
This document defines the **exercise assignments** for the ["Databases Advanced – Entity Framework" course @ Software University.](https://softuni.bg/courses/databases-advanced-entity-framework)

## Create a Database for Student System using Code First

First of all, create a **Blank Solution** and name it accordingly - in our case, **BookShopSystem** sounds like a nice name.

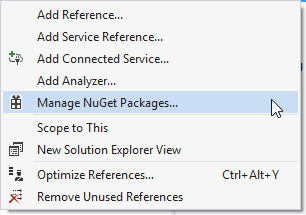


After that create simple console application:

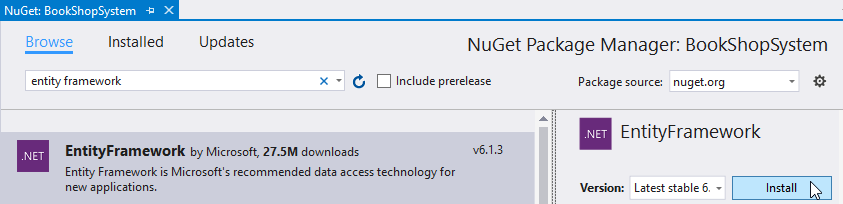


We will separate code into different folders - **Models** (holding the classes Entity Framework will use as a basis for creating our database), **Data** (holding Entity Framework specific logic) and **Client** (holding our renamed Program.cs file).

Before we continue make sure that we have installed Entity Framework (go to the project then right click on “**References**” -> “**Manage…**”):



Then in the new query window change section to “**Browse**” search for entity framework and install it.



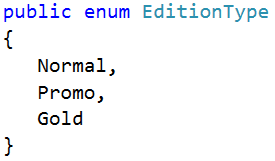
### Step 1 - Model the Database

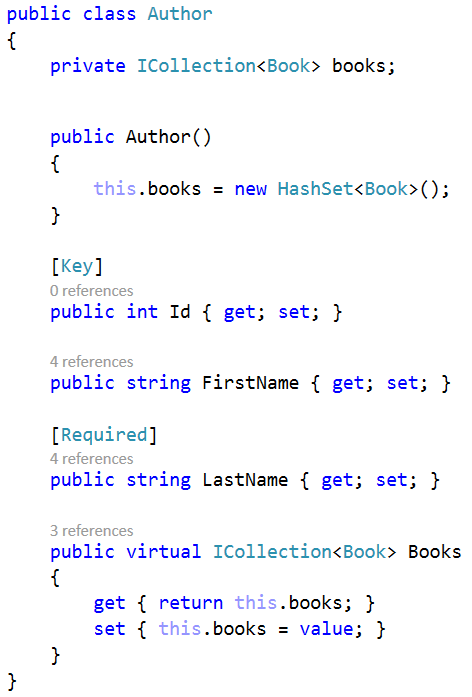
A book shop keeps **books**. A book can have one **author** and many **categories**. Let's create a class for each of the main tables.

* **Book** - id, title (between 1 ... 50 symbols - inclusive), description (optional, up to 1000 symbols), edition type   
  (**Normal**, **Promo** or **Gold**), price, copies (number), release date (optional)
* **Author** - id, first name (optional) and last name
* **Category** - id, name

Assume everything **not market optional** is mandatory. The **classes** should describe with **properties** each of the **table columns**.

For the book edition type you could use enumeration:





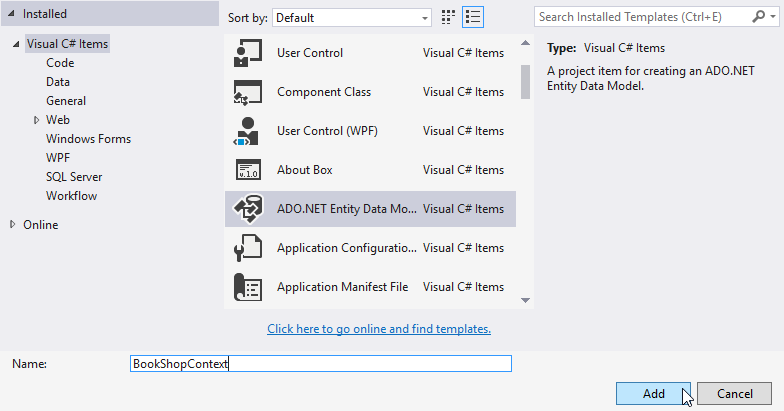
Add constraints as described above using attributes (e.g. **Required**, **MinLength**, **MaxLength**, etc). Do the same for the **Book** and **Category** models. Make sure you add **navigation properties** when there are relations.

**Important** **Note**: Always make **navigation properties** **virtual**, so Entity Framework can **override** them and apply **lazy loading**.

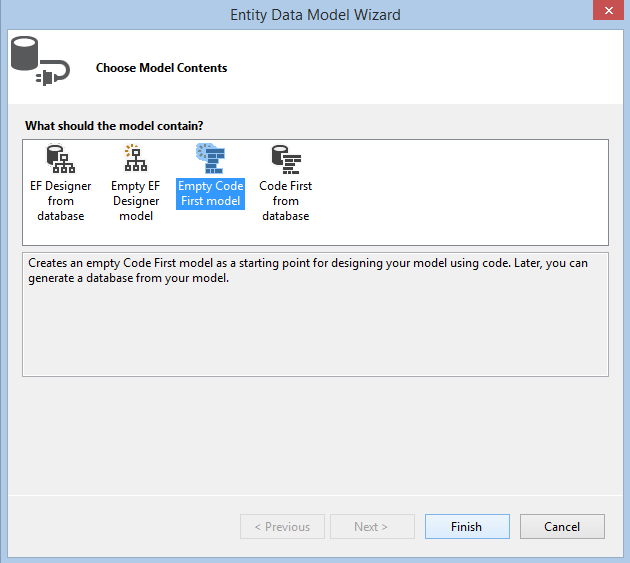
### Step 2 - Create the Data Folder

Once the models are done, our next step is to write the so-called **Data Layer** (Folder). Let's create it in a separate folder.

In the newly created folder, add a new ADO.NET Entity Data Model. Name it **BookShopContext**.

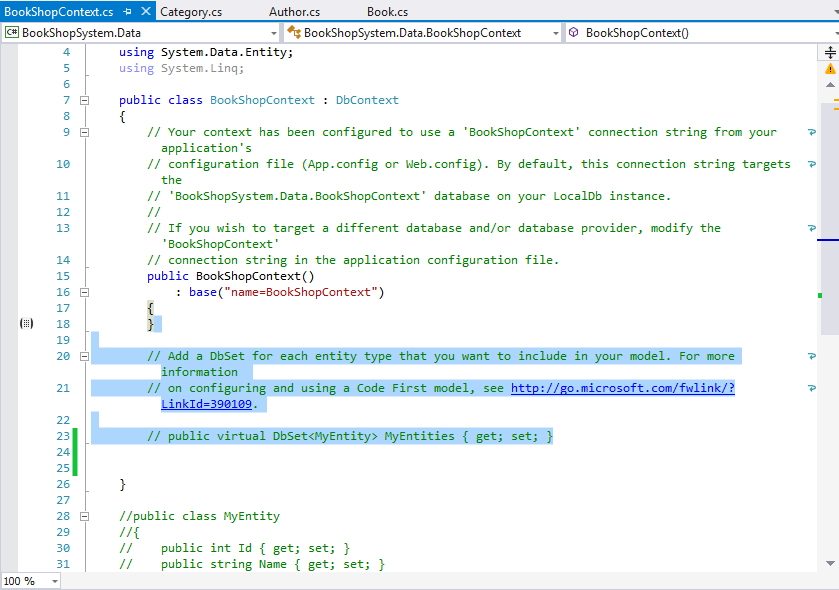


From the Model Wizard, select **Empty Code First Model** (meaning we will model our database from the code we write alone).

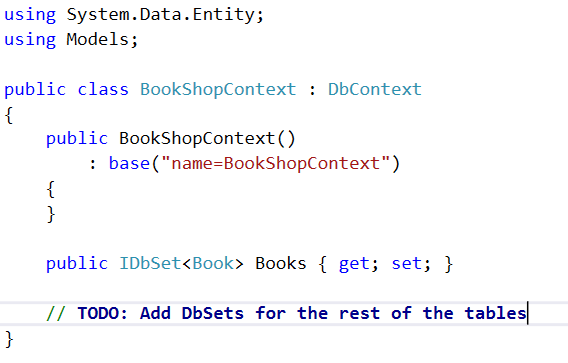


The Wizard should generate a **BookShopContext** class in our Data folder. The idea behind the context is that it **manages the connection to the database** for us and reveals all data as **DbSet<T>** - generic collections which represent the data stored in our database. Thanks to it, we can easily work on that data with standard CRUD operations - just like we work with any C# collections!

Delete the auto-generated comments and let's start writing.



In the **BookShopContext** class, write in all necessary tables (as shown below) as **IDbSet<T> properties**, where **T** is the respective table. Make those properties public so we can access them outside the current project.



Make them **virtual**.

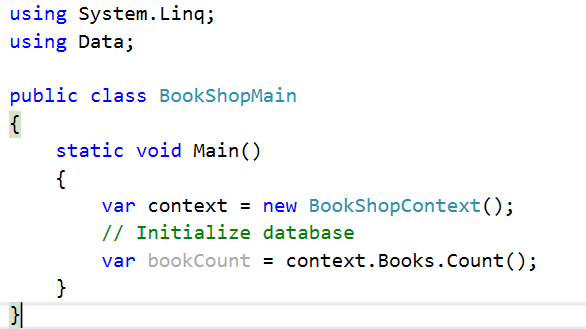
After that's done, there's one final thing left. In the **App.config** file there is a **<connectionStrings/>** tag which keeps information about the connection.

* The **name** attribute should correspond to the **connection string** in the **Context** **constructor**.
* Data source holds the target database (if you're using SQL Express it should be **.** or **localhost**. Otherwise, leave it **(LocalDb)\v11.0**.

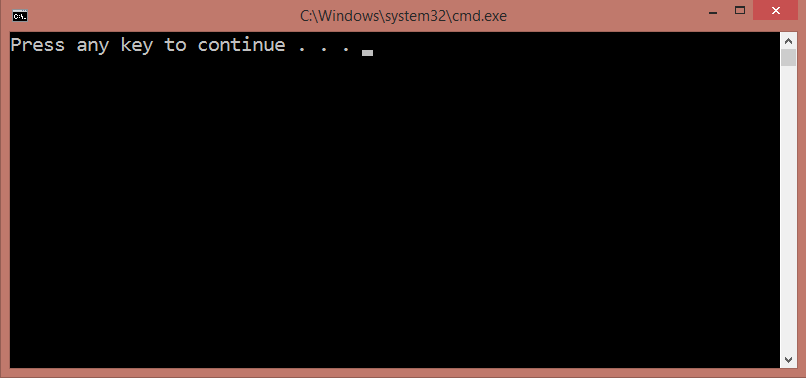
### Step 3 - Console Client

We now have our **models** (classes which will be mapped to database tables) and our **data layer.** It's time we write the console client and start Entity Framework for the first time.

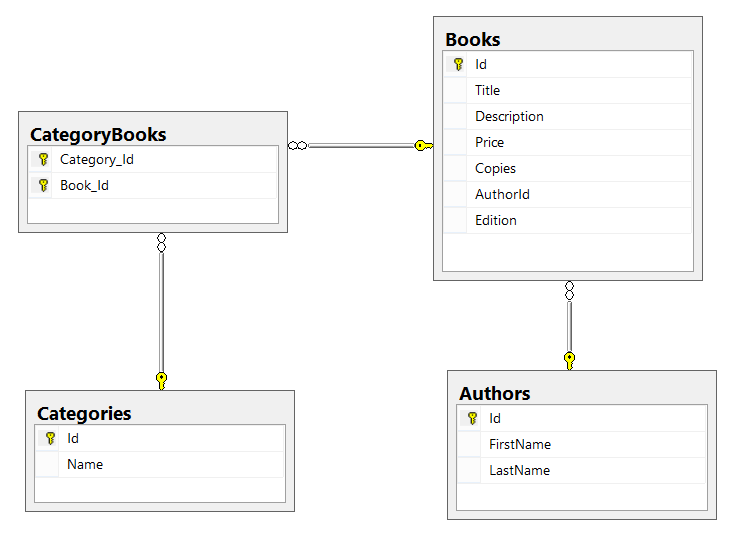
Finally, in order for Entity Framework to **create the database** for us, we need to execute any action through the **context**.



The application should finish without any exceptions.

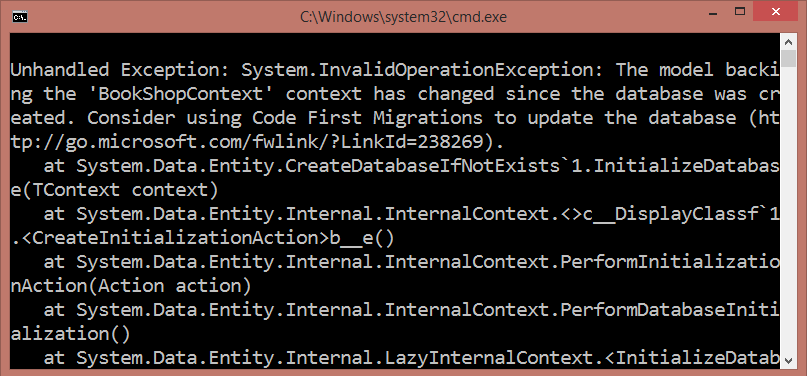


The generated relational **diagram** should be as follows:



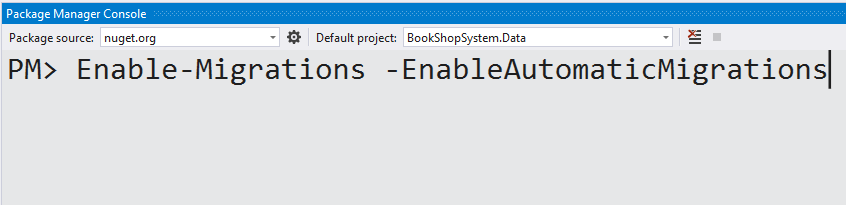
### Step 4 - Make Changes to the Local Model

Let's add a new property to the **Book** class - **AgeRestriction** (Minor, Teen or Adult). Run the program again. We expect EF to update the dabase according to the changes made to the models. However…

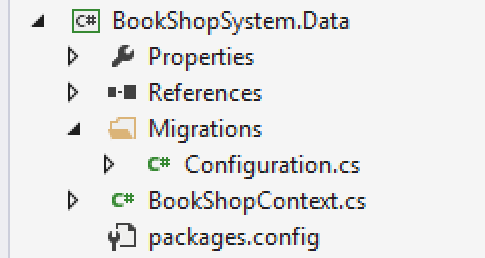


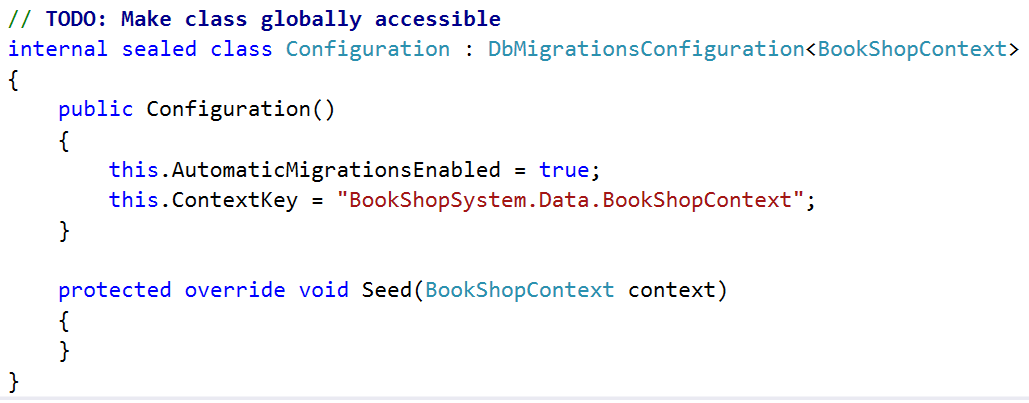
We get an exception and we are told to use **Code First Migrations**. **Migrations** update database according to the model by using a **strategy**. We need to configure Entity Framework to use a **migration strategy** (i.e. tell it what to do each time it **connects** to the database for the **first time**).

First, we need to **enable automatic migrations**. Open the Package Manager Console and write the following command:



This will create a **Configuration.cs** class which enables automatic migrations and has a **Seed()** method.

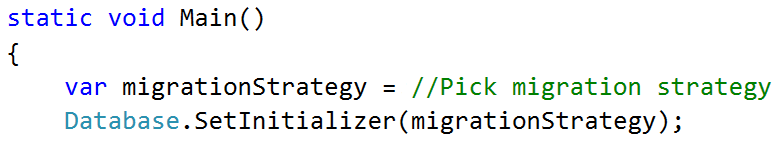




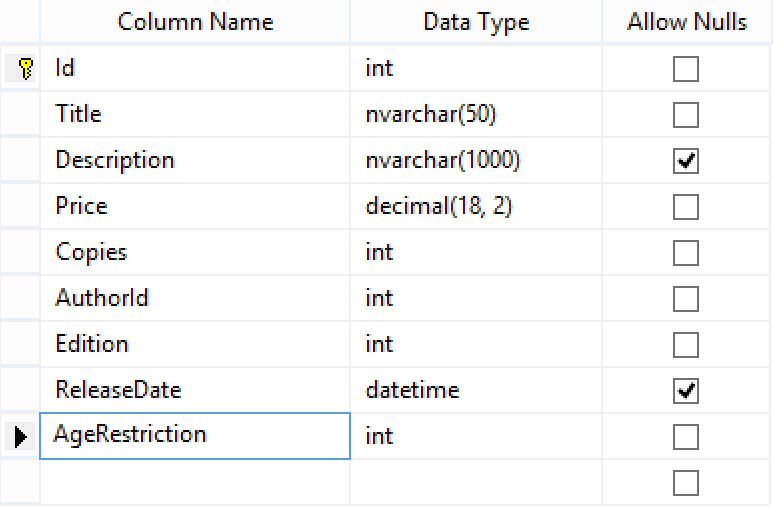
Finally, we need to the EF to apply a specific strategy to this configuration. The most used migrations are as follows:

* **DropCreateDatabaseIfModelChanges** - We lose all the data when the model changes
* **DropCreateDatabaseAlways** - Recreate the database each time we start the program
* **MigrateDatabaseToLatestVersion** - Updates the database in accordance to the changes we've made to the model

Select the last migration strategy and initialize it as shown below in the **Main()** method.



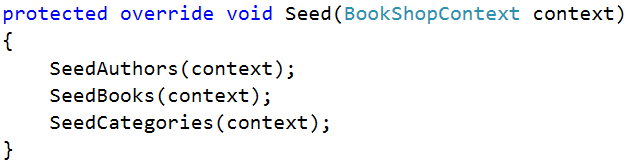
Once done, start the program again and no exception should occur. The design of the Books table should have a new column added.



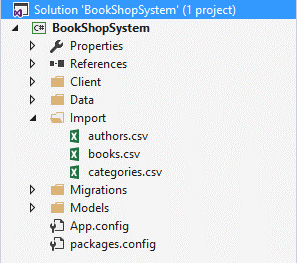
### Step 5 - Seed Data into the Database

We have our database up and running. However, there is no data to work with. Let's seed some!

Go to the **Seed()** method of the **Configuration.cs** class we just created. Add three methods as shown below.



Create new folder “**Import**” inside your project. In it put all **authors.csv**, **books.csv** and **categories.csv**:



Here is how SeedAuthors may look like:

|  |
| --- |
| **Importing Authors from File** |
| private static void SeedAuthors(BookShopContext context)  {  string[] authors = File.ReadAllLines("../../Import/authors.csv");  // Foreach all authors and add them to database.  // Skip first row because it is only describing column names.  for (int i = 1; i < authors.Length; i++)  {  string[] data = authors[i].Split(',');  string firstName = data[0].Replace("\"", string.Empty);  string lastName = data[1].Replace("\"", string.Empty);  Author author = new Author()  {  FirstName = firstName,  LastName = lastName  };  // When adding we may want to check if author with same first name and last name is existing.  context.Authors.AddOrUpdate(a => new { a.FirstName, a.LastName }, author);  }  } |

Useful links:

* What is [CSV](https://en.wikipedia.org/wiki/Comma-separated_values)?

Here is how SeedBooks may look like:

|  |
| --- |
| **Importing Books from File** |
| private static void SeedBooks(BookShopContext context)  {  int authorsCount = context.Authors.Local.Count;  string[] books = File.ReadAllLines("../../Import/books.csv");  for (int i = 1; i < books.Length; i++)  {  string[] data = books[i]  .Split(',')  .Select(arg => arg.Replace("\"", string.Empty))  .ToArray();  int authorIndex = i % authorsCount;  Author author = context.Authors.Local[authorIndex];  EditionType edition = (EditionType)int.Parse(data[0]);  DateTime releaseDate = DateTime.ParseExact(data[1], "d/M/yyyy",CultureInfo.InvariantCulture);  int copies = int.Parse(data[2]);  decimal price = decimal.Parse(data[3]);  AgeRestriction ageRestriction = (AgeRestriction)int.Parse(data[4]);  string title = data[5];  Book book = new Book  {  Author = author,  AuthorId = author.Id,  Edition = edition,  ReleaseDate = releaseDate,  Copies = copies,  Price = price,  AgeRestriction = ageRestriction,  Title = title  };  context.Books.AddOrUpdate(b => new { b.Title, b.AuthorId }, book);  }  } |

Useful links:

* Working with [enumerations](https://www.dotnetperls.com/enum).
* Working with context [local](http://www.entityframeworktutorial.net/EntityFramework4.3/local-data.aspx) data.

You should implement SeedCategories like this:

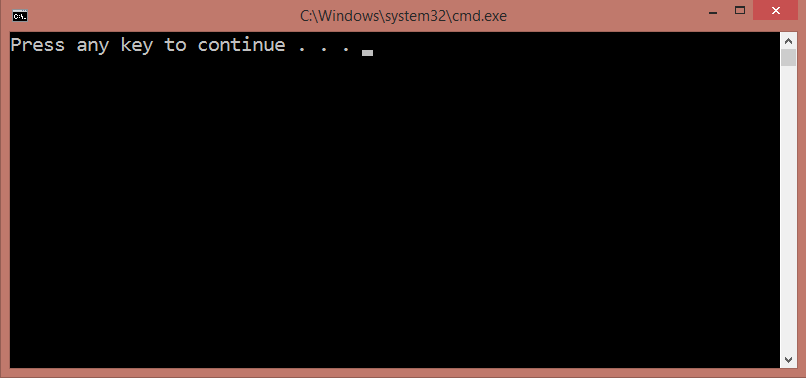
|  |
| --- |
| **Importing Categories from File** |
| private static void SeedCategories(BookShopContext context)  {  int booksCount = context.Books.Local.Count;  string[] categories = File.ReadAllLines("../../Import/categories.csv");  for (int i = 1; i < categories.Length; i++)  {  string[] data = categories[i]  .Split(',')  .Select(arg => arg.Replace("\"", string.Empty))  .ToArray();  string categoryName = // TODO: Get name from data.  Category category = new Category() { Name = categoryName };  int bookIndex = (i \* 30) % booksCount;  for (int j = 0; j < bookIndex; j++)  {  Book book = context.Books.Local[j];  // TODO: Add book to current category  }  context.Categories.AddOrUpdate(c => c.Name, category);  }  } |

### Step 6 - Related Books

Let's say at one point we decide that **books** should have **related books** - i.e. a book has many related books and each related book has related books as well.

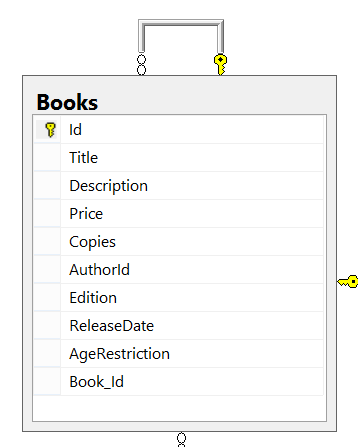
Go to the **Book** class and add a **navigational property RelatedBooks**(collection of releted books). Make sure you **instantiate** it in the constructor.

Start the program again.



No errors. The migration has most likely been successful and we should see a many-to-many self-reference in the database schema.

However, that's not the case - the migration is successful but a book can have only 1 related book.

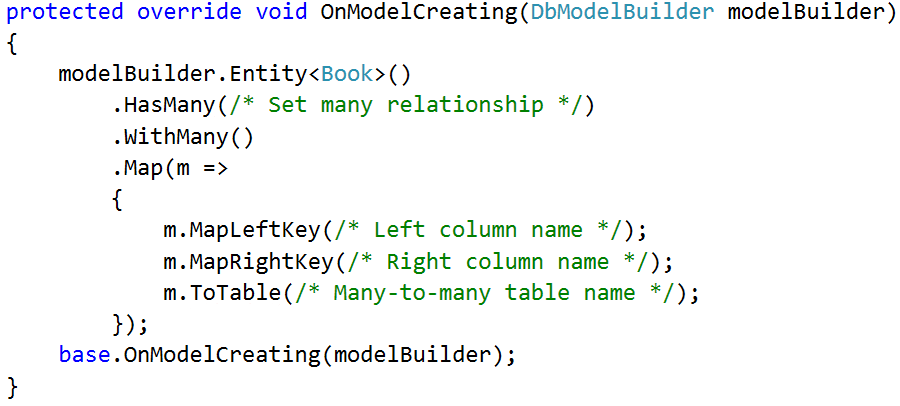


This is one of those cases where Entity Framework does not correctly update the DB schema after changes have been made to the model.

In such events, we need to manually configure the relationship with the so-called **Model Builder**. Go to the **BookShopContext** class and override the **OnModelCreating()** method.

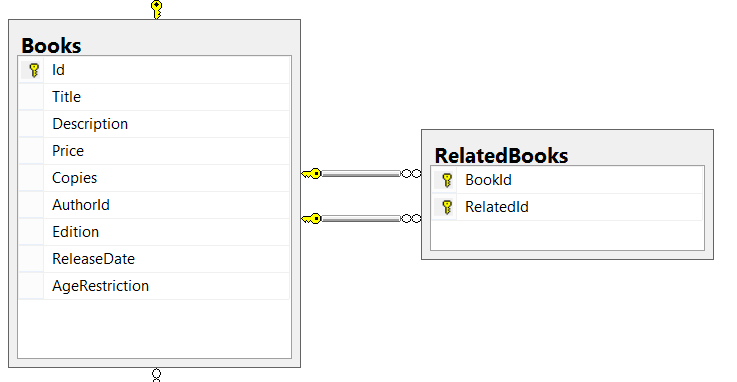
This method is called only once when entity framework starts for the first time (like **Seed()**). It receives a **DbModelBuilder** as argument. We will use that model builder to configure the desired relationship.

The Model Builder (also **fluent API**) will allow us to tell EF that a **Book** has **many books** (the related books) and those books have many as well. Then we will map that relationship to a **junction** (**many-to-many**) table where we manually set the **left key** (book id) and **right key** (related book id). Finally, we **name the table**.



At the end, we call the **base** implementation of the method (it contains native EF configurations, so it's best we call it at the end).

Restart the program and if no exception is thrown, the new DB schema should now look as follows:



Keep in mind that your column and table names may differ from the ones in the above picture.

And finally, let's test the **RelatedBooks** functionality it. **Query 3 books** from the database and set them as **related**.

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| var books = context.Books  .Take(3)  .ToList();  books[0].RelatedBooks.Add(books[1]);  books[1].RelatedBooks.Add(books[0]);  books[0].RelatedBooks.Add(books[2]);  books[2].RelatedBooks.Add(books[0]);  context.SaveChanges();  // **TODO:**  // **Query the first three books to get their names**  // **and their related book names**  foreach (var book in booksFromQuery)  {  Console.WriteLine("--{0}", book.Title);  foreach (var relatedBook in book.RelatedBooks)  {  Console.WriteLine(relatedBook);  }  } | --Absalom  A che punto A" la notte  After Many a Summer Dies the Swan  --A che punto A" la notte  Absalom  --After Many a Summer Dies the Swan  Absalom |