



**Database Access Technology**

**ADO.NET**

# Lesson №6

## Getting acquainted with Entity Framework

### Contents

<b>ORM system concept</b> .....	4
New terms .....	8
<b>Entity Framework architecture</b> .....	9
Entity Data Model .....	9
Object Services Layer .....	10
Entity Client data provider .....	11
ADO.NET data provider .....	11
New terms .....	12
<b>Entity Data Model</b>	
<b>creation approaches</b> .....	13
Database first .....	13
Model first .....	13
Code First .....	14
New terms .....	14

<b>DbContext class</b> .....	<b>15</b>
New terms .....	19
<b>Database first</b> .....	<b>21</b>
Database creation in LocalDB .....	23
EDM creation for Database first .....	28
<b>LINQ to Entities (introduction)</b> .....	<b>36</b>
First() и FirstOrDefault() .....	36
Single() и SingleOrDefault() .....	37
ToList() .....	39
OrderBy() .....	40
Find() .....	41
<b>Filling the database</b> .....	<b>43</b>
<b>Navigation properties</b> .....	<b>46</b>
New terms .....	49
<b>Home task</b> .....	<b>50</b>

# ORM system concept

---

Today in most cases, storing information in relational databases is a standard technique. Databases proved their effectiveness and safety. You remember that in relational databases information is stored in related tables. Imagine that you write an application, which should work with a database. You are already familiar with the ADO.NET technology; therefore, you can estimate those tasks, which should be solved.

The application needs to get data out of database tables and place it in class objects. In the ADO.NET technology, the *DataTable* or *SqlDataReader* classes can be used. You remember, of course, that each DBMS works with its data types, which often don't have an exact match in the application development environment. How do you think, why does the *SqlDataReader* class contain the following methods: *GetInt16()*, *GetInt32*, *GetInt64()*, *GetFloat()*, *GetDouble()*, *GetDecimal()*, and several dozens of similar ones? In order to read database data as accurately as possible. But even having this set of tools, you can't always get an exact match. In many cases, it's necessary to resort to the last argument, the *Object* type. You remember that *Object* is basic for all other types, and therefore, we can convert any data type into *Object*. What does it lead to? It leads to the fact that in order to read data from each database table, it's necessary to write a "personal" code, configure it to a table structure, and monitor the compatibility of data types. There can be dozens of tables in a real database, and I'll tell you a secret, even hundreds. Of course, if you are quite persistent

and hard-working, you will write and debug this code! And when you will bring it to a customer, you suddenly will find out that some database tables were changed. And you will have to change your code, wherein you work with these tables. And of course, this code is related to other processing of your application, so you will have to make changes there as well. Of course, no one is sorry about your work and time spent for creating this data processing ☺.

Is it possible to optimize this process? Let's consider the following approach. In a database, there is a Book table, which you should work with. You create a class in the application and name it Book as well (although it's optional). In this class, you create properties corresponding to table fields and methods for reading data from the table into its class objects. In addition, you create a method, which allows you to record class objects into a database table. I think you remember what encapsulation is? Here, it is manifested in its pure form: you encapsulate all the required interactions with the Book table in this class. You create a "database table – application class" conformance. Herewith, each string, which is read from the Book table, will be transformed into a separate object of the Book class. Here, it's necessary to implement the required conversion of data types between the table fields and class properties. The same should be done for all database tables, which the application should work with. If you collect all this processing in one place, it will greatly reduce the dependence of the rest application code from the database structure.

What are the benefits of this approach? The database encapsulates in one place. If a table structure is changed, and

according to Murphy's Law, it will be definitely changed, you will have to make changes only in the class that corresponds to the changed table.

What are the drawbacks of this approach? Anyway, you should solve issues of the compatibility of types at the level of each class and each table. It's necessary to create many new application classes and think over their interaction with other parts of the application. It's necessary to create containers in order to store and process objects of these classes. In other words, this approach solves some problems but creates others. In addition, we assumed that one application class corresponds to one database table, but this assumption is far from the real life. Let's continue to talk.

Let's introduce several terms which we will need later. You should remember that the information about a single object can be located not in a single database table but in several related database tables. For example, in order to describe a book, it is useful to store information about the publishing house that published a book. However, the table normalization rules require putting information about the publishing house into a separate table, for example, Publisher, and inserting the foreign key referred to the publishing description into the Book table. When using this approach, the information about a single book will be stored in two tables: Book and Publisher. Similarly, you can place information about the author of a book and maybe something else into a separate table. When transferring into the application, data on each book should be already stored in several classes that are related to each other.

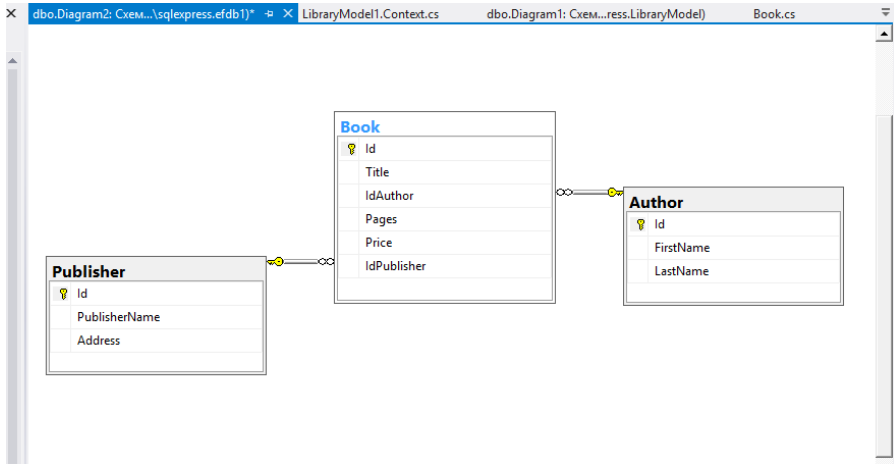


Fig. 1. Database structure

Keep in mind the first term: "object model". An object model is a group of application classes that are related to each other and used to store, process, and display database data. And now, it would be more correct to talk about the "database – object model" conformance instead of "database table – application class" conformance.

ORM systems are designed to implement this conformance. ORM stands for Object Relational Mapping.

Developers have faced the discussed problem long time ago. The first commercially successful ORM systems appeared in 1995. It was the TOPLink Framework created in Smalltalk. This framework was ported for earlier Java versions. A little bit later, the NeXT operating system presented the DbKit product, which solved the same task. Today, there are many ORM systems for different platforms and programming languages. To verify this, look at this review: [https://en.wikipedia.org/wiki/List\\_of\\_object-relational\\_mapping\\_software](https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software).

The subject of our study, Entity Framework, is an ORM system developed by Microsoft. Let's recall what "framework" means. Although, we understand the framework concept intuitively, I would like to explain it once again. For example, like this: it's a software or platform that provides a standardized way to solve the task of a system (or subsystem) design and development.

Entity Framework first appeared in .NET Framework 3.5 Service Pack 1 in 2008. The first version of the product caused a number of critical comments, but developers improved Entity Framework very substantially, and today, it's one of the leaders in this segment of tools. You will see very soon that the use of ADO.NET is not the only way to work with a database. But it doesn't mean that we don't need the ADO.NET technology any more. It is still an important and demandable tool. But now, you get a new tool, which allows you to work more effectively.

So, the Entity Framework task is to implement the conformance between objects, used in the application, and related database tables. Herewith, Entity Framework solves automatically many issues that arise when creating this conformance. We will consider all of this in our lessons.

### ***New terms***

- *Object model* is a group of application classes that are related to each other and used to store database data.
- *ORM* stands for Object Relational Mapping.



# Entity Framework architecture

---

## Entity Data Model

We have already found out that Entity Framework is a framework for ADO.NET that provides a developer with improved methods for storing and processing database information and displaying the database information within the application. Let's consider the Entity Framework internal structure. In order to do this, we should get acquainted with Entity Data Model (EDM) of this framework. EDM describes the interrelation between both application classes and database tables. EDM consists of three components:

1. **Conceptual model** describes application classes and relationships between them.
2. **Mapping** contains the correspondence schema between Conceptual model and Storage model, that is, between the application classes and database tables.

All this information about the database structure (Storage model), about the data model (Conceptual model), and about their mutual mapping is contained in an XML-file with the .edmx extension. In addition to EDM, Entity Framework contains a number of important components which are called layers.

3. **Storage model** describes related database tables.

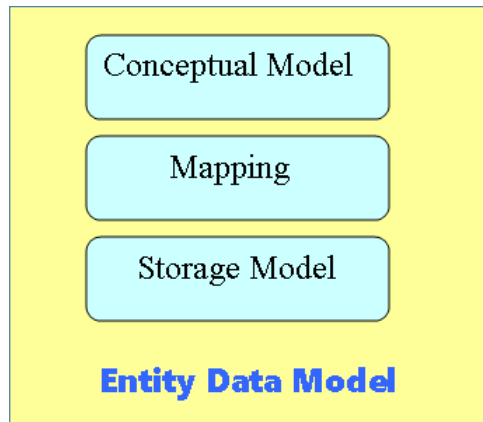


Fig. 2. Entity Framework architecture

### Object Services Layer

According to the fifth Codd's rule, "Any RDBMS should support at least one relational language". Entity Framework offers two ways to access the DBMS: LINQ to Entities and Entity SQL.

**LINQ to Entities** is a LINQ extension that creates queries to Conceptual model (that is, to application object classes) in C# or VB. Let's consider in more detail the use of LINQ to Entities.

**Entity SQL**, according to the MSDN, "is a storage-independent query language that is similar to SQL. Entity SQL allows you to execute queries to entity data represented either as objects or in a tabular form". This query is a bit more complicated than LINQ to Entities and requires special consideration.

So, **Object Services Layer** is the most important component of Entity Framework, which allows a user to use the programming language (**LINQ to Entities or Entity SQL**) to create

database queries. This layer works with application object classes by synchronizing them with data in database tables. In this layer, we perform such actions as: fixing a current state of objects and converting data, obtained from database tables as a query result, into application object classes.

### Entity Client data provider

After receiving a query to LINQ to Entities or Entity SQL, this layer converts it into SQL and transfers it to Entity Client data provider.

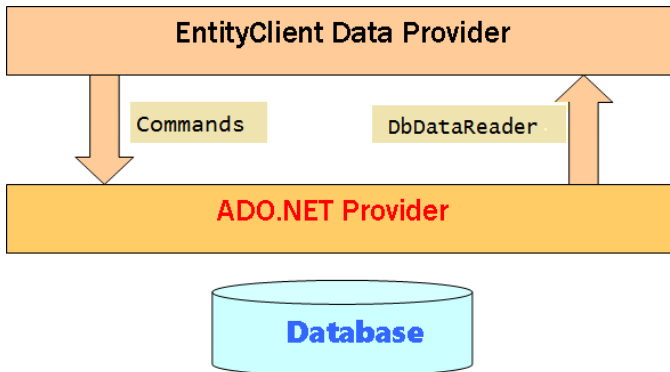


Fig. 3. Interaction of layers

### ADO.NET data provider

This layer is designed to refer directly to the DBMS using the ADO.NET technology. Until this point, all queries created in LINQ to Entities or Entity SQL should be converted into SQL queries.

### ***New terms***

- ***EDM (Entity Data Model)*** is a model, which describes, on the one hand, the interrelation of application object classes and on the other hand, related database tables.

# Entity Data Model creation approaches

---

We've established that Entity Data Model is the Entity Framework foundation. Developer has several ways to create Entity Data Model. Each of them leads to the same result actually, but has its own distinctive characteristics. Let's consider them. We will use Visual Studio 2015 and Entity Framework 6.1 suitable for writing applications for .NET 4.0 and .NET 4.5 versions. But before proceeding to the creation of examples, let's list and briefly describe all these approaches.

## **Database first**

In this case, a developer should firstly create EDM in Visual Studio. To do this, Visual Studio provides a special operation mode in the designer. We will consider how it's done. After that, we will create a database on the base of the created Data model. It's recommended to use this approach in the case, when you can imagine clearly the database structure or even better, when you know it.

## **Model first**

This way of working with Entity Framework appeared beginning with the version 4.1. It's particularly close to programmers because it requires from them to execute traditional actions, that is, to write a code. At this approach, a programmer creates the required code classes, on the base of which the Entity Framework creates EDM and a database.

## Code First

This way of working with Entity Framework appeared beginning with the version 4.1. It's particularly close to programmers because it requires from them to execute traditional actions, that is, to write a code. At this approach, a programmer creates the required code classes, on the base of which the Entity Framework creates EDM and a database.

### *New terms*

- ***Database first, Model first, Code First*** are different ways of creating EDM when working with Entity Framework.

# DbContext class

We need a database in order to study Entity Framework. It will be an arbitrary database in the sake of simplicity, of course, and in order not to deflect our attention from an acquaintance with Entity Framework. Let's create a new database on the MS SQL Server, for example, Library and create Book, Author, and Publisher tables in it.

```
CREATE TABLE Author
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    FirstName VARCHAR(100) NOT NULL,
    LastName VARCHAR(100) NOT NULL
)

CREATE TABLE Publisher
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    PublisherName VARCHAR(100) NOT NULL,
    Address VARCHAR(100) NOT NULL
)

CREATE TABLE Book
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    Title VARCHAR(100) NOT NULL,
    IdAuthor INT NOT NULL FOREIGN KEY REFERENCES
    Author(Id),
    Pages INT,
    Price INT,
    IdPublisher INT NOT NULL FOREIGN KEY
    REFERENCES Publisher(Id)
)
```

You can already create databases, so we will not dwell on this stage. Scripts for creating tables are presented in order to ensure the correspondence of table names and table fields in your applications. If you have an already installed MS SQL Server, you can run Management Studio and create a database on this server. In this case, you should keep in mind the server address, where you created your database. You'll need this address soon. You can create your database in LocalDB, like Visual Studio 2015 prompts by default.

We have already found out that when creating EDM according to the "Database first" technology, Entity Framework adds classes corresponding to database tables to the current project. We will repeat once again that these classes are called entities in the Entity Framework terminology. For our database, we will obtain the Author, Publisher, and Book classes in our application. However, it turns out that Entity Framework adds not only these classes. In addition to classes corresponding to database tables, Entity Framework creates one more class that is called a database context. It's one of the main classes of Entity Framework. Let's get acquainted with it closer. For example, for our database, this class can look as follows:

```
public partial class LibraryEntities: DbContext
{
    public LibraryEntities()
        :base("name=LibraryEntities")
    {
    }
    protected override void
    OnModelCreating(DbModelBuilder modelBuilder)
    {
```



```
protected override void
OnModelCreating(DbModelBuilder modelBuilder)
{
    throw new UnintentionalCodeFirstException();
}

public DbSet<Author> Author { get; set; }
public DbSet<Book> Book { get; set; }
public DbSet<Publisher> Publisher { get; set; }
}
```

As you can see, this class is derived from the System.Data.Entity.DbContext class. LibraryEntities executes the following main tasks:

- Connects to a database;
- Executes our database queries by converting database table values into application object classes (entities) and vice versa;
- Tracks and saves changes of the state of application object classes (entities) after the query execution.

In order to do this, theObjectContext class was used in the Entity Framework in earlier versions. However, it was a bit more complicated to work with it. The DbContext class is a wrapper around ObjectContext and provides a developer with more convenient mechanism of working with a database. By the way, ObjectContext and DbContext are not abstract classes, although they are used as basic classes. Inheritance is necessary in order to add properties responsible for a data access to this database within the database context (in our case, within the LibraryEntities class). When using the DbContext class, these properties have a DbSet<T> type where the T

parameter presents the entity names of the model, that is, names of classes that correspond to database tables.

Pay attention to the constructor of this class. Since LibraryEntities should connect to a database, we should specify somehow what database should be connected. In order to do this, the overloaded constructor of the DbContext class is used. Look at the description of the DbContext class constructors in the MSDN by the following link:

[https://msdn.microsoft.com/ru-ru/library/system.data.entity.dbcontext\(v=vs.113\).aspx](https://msdn.microsoft.com/ru-ru/library/system.data.entity.dbcontext(v=vs.113).aspx)

I think that it will be useful to present these descriptions from the MSDN here.

<code>DbContext()</code>	Constructs a new context instance using conventions to create the name of the database to which a connection will be made. The by-convention name is the full name (namespace + class name) of the derived context class. See the class remarks for how this is used to create a connection.
<code>DbContext(String)</code>	Constructs a new context instance using conventions to create the name or connection string of the database to which a connection will be made. See the class remarks for how this is used to create a connection.
<code>DbContext(DbCompiledModel)</code>	Constructs a new context instance using conventions to create the name of the database to which a connection will be made, and initializes it from the given model. The by-convention name is the full name (namespace + class name) of the derived context class. See the class remarks for how this is used to create a connection.

DbContext (DbConnection, Boolean)	Constructs a new context instance using the existing connection to connect to a database. The connection will not be disposed when the context is disposed if <i>contextOwnsConnection</i> is false.
DbContext (String, DbCompiledModel)	Constructs a new context instance using the given string as the name or connection string for the database to which a connection will be made, and initializes it from the given model. See the class remarks for how this is used to create a connection.
DbContext (ObjectContext, Boolean)	Constructs a new context instance on the base of the existing ObjectContext object.
DbContext (DbConnection, DbCompiledModel, Boolean)	Constructs a new context instance using the existing database connection, and initializes it from the given model. The connection will not be disposed when the context is disposed if <i>contextOwnsConnection</i> is false.

In the above example, we used the second constructor to which the connection string name to the required database is transferred.

Now, when we reviewed how Entity Framework works, we can proceed to specific examples.

### New terms

- **Database context** is a special class derived from the DbContext system class, and designed to connect to a database and execute database queries.

- ***Entity*** is a class corresponding to a database table created by Entity Framework automatically. Properties of this class correspond to table fields.

# Database first

---

Run Visual Studio 2015, and create a new project. Let it be a console application in C# named LibraryDbFirst, but it's optional. It's just the simplest kind of application, and we will not distract from the issues of using Entity Framework.

For the existing database, we will create Entity Data Model in this project, that is, according to the "Database first" technology. Before proceeding further, it's useful to make sure that the required Entity Framework version is installed in Visual Studio. It will be useful to make a small digression here in order to talk about the NuGet package. If you know what NuGet is, simply miss the next three paragraphs.

***NuGet** is a free Visual Studio extension designed for adding third-party libraries to your application. NuGet allows you to update and remove these libraries as well. The added library is expanded in a project as a package. NuGet-package is a set of files packaged into one file with the .nupkg extension in the format of Open Packaging Conventions (OPC). In its turn, OPC is simply a zip-file with some metadata. Chances are, NuGet is already installed in your Visual Studio. You can check this in the Project menu.*

*Look whether there is the Manage NuGet packages command there. If yes, enable it, and you can work with NuGet. If no, you should navigate to the Tools – NuGet packageManager menu. After enabling this menu point, you will see a Manage NuGet Packages for Solution command. Enable this command and thus, you will be able to work with NuGet. This package will be*

*useful for you, not only when working with Entity Framework but in other situations as well.*

*It's very simple and convenient to work with this package. Recall how you usually installed different extensions and plugins. At first, you should find a link to a necessary distribution, make sure that this distribution is workable, download it, and install. For NuGet, you should specify only the name of the required extension. NuGet will find the distribution by itself and install it. Of course, you should be connected to Internet. After NuGet is enabled, you will see the following window:*

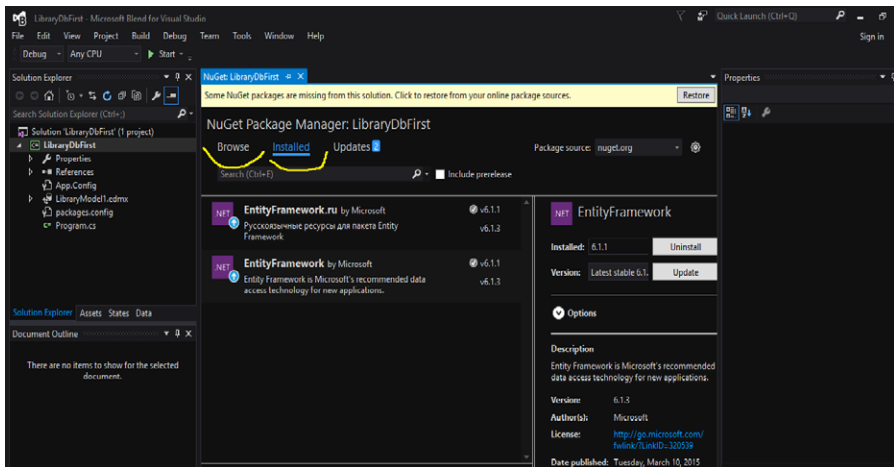


Fig. 4. Working with NuGet

In the appeared window, please, follow the Installed link and check whether there is Entity Framework among installed packages. As it shown in the picture, in my case, Entity Framework is already installed. At the time of writing this lesson, the EntityFramework 6.1.1 version is installed. You probably face a similar situation. If EntityFramework

is not installed in your project, follow the Browse link on the left, enter EntityFramework into a search window, and when the package will be found, click the Install button on the right. The installation takes very little time.

And now, we are ready to begin to work with Entity Framework. It will be necessary to repeat this action in each project, wherein you will work with Entity Framework. Yes, it increases the project size, but soon, you will make sure that this is perhaps the only drawback of using Entity Framework. There are much more advantages of using Entity Framework.

Now, let's return to the Entity Data Model creation according to the "Database first" technology. This approach assumes that the database is already created. Perhaps, you have already created a database of three tables on your server. But I offer you to consider the issue of creating a new database in a new Microsoft product in order to work with a database in the so-called LocalDB. The experience of working with LocalDB will be useful in any case, therefore, even if you created a database on the allocated server, I advise you to create it on LocalDB. To do this, follow these steps.

### **Database creation in LocalDB**

After you've created a new application, for example, named LibraryDbFirst, it's necessary to create a new database. By default, Visual Studio 2015 creates a database in LocalDB. LocalDB is a simpler version of SQL Server Express Edition. It supports all the functionality of SQL Server Express

Edition (except for FileStream), but it is installed quicker and runs in the user mode and not as a service. LocalDB is not designed for scripts with the remote connection, but since a database is stored in an .mdf file, it allows you to easily pass an application from one computer to another. In other words, LocalDB is a perfect tool for debugging (and not only), and it will be useful for everyone to learn to work with it. You can search for more information on the Web, for example, here:

<http://geekswithblogs.net/krislankford/archive/2012/06/19/sql-server-2012-express-localdb-how-to-get-started.aspx>.

And we'll proceed to a database creation. Enable the View-Server Explorer menu items in the created project (Fig. 5).

The Server Explorer window will appear in the right-hand side of your screen, where you will see the Data Connections section. Hover over this section, call the context menu, and select the Connect to Database command.

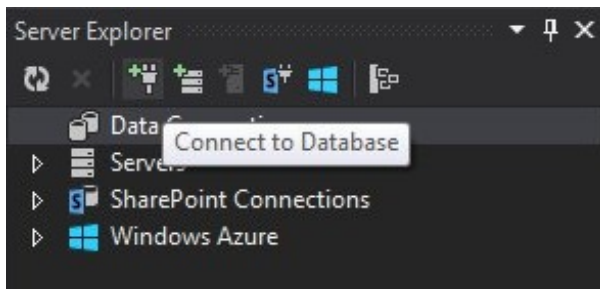


Fig. 6. Creating a database connection

You should specify the Microsoft SQL Server (SqlClient) as a data source in the appeared window in the "Data source" field. If another value is specified, click the Change button



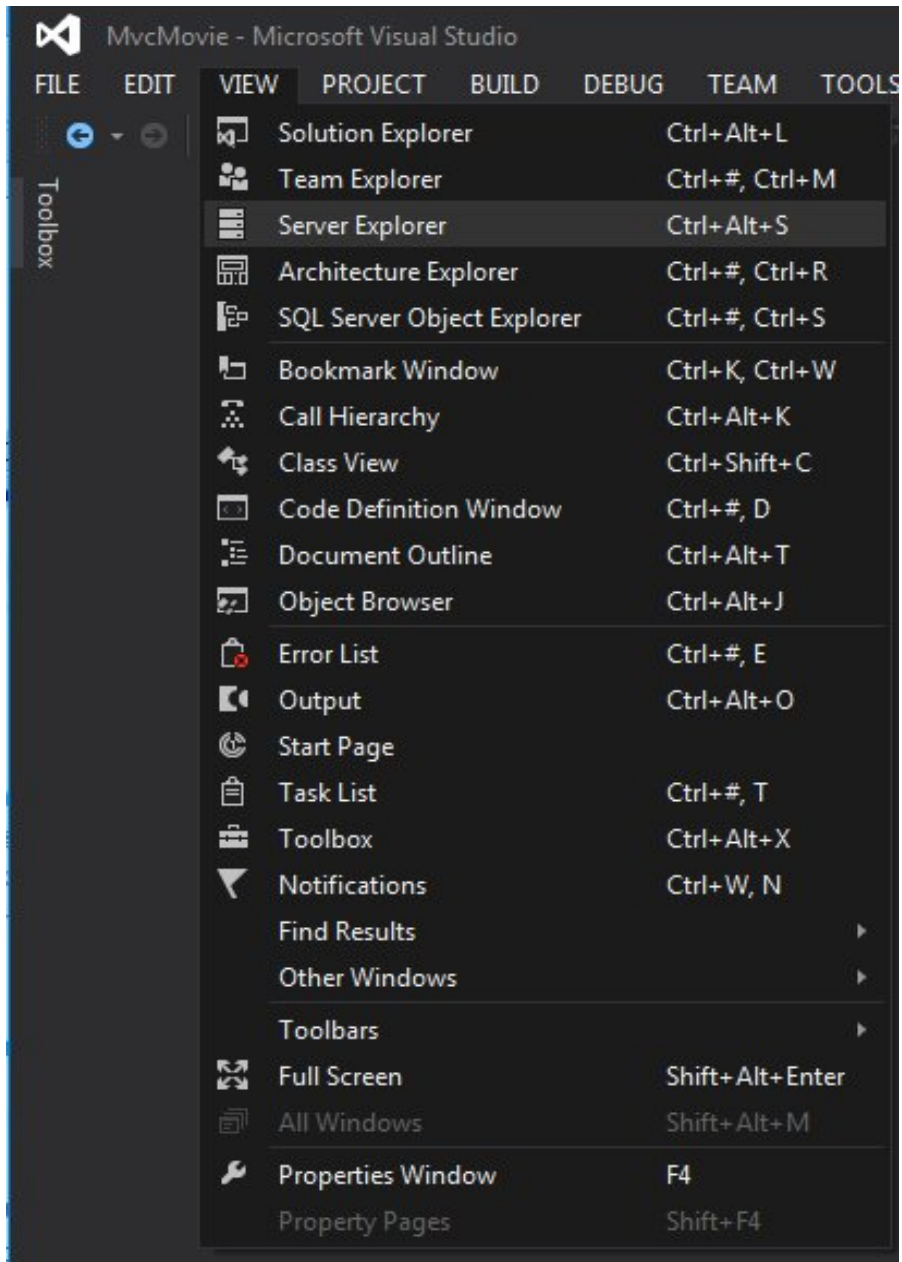


Fig. 5. Selecting Server Explorer

and select this value. You should enter the following value (localdb)\v11.0 into the "Server name" field. This is a LocalDB server registered name. You should specify it to the nearest character, as it is specified here. However, the character case is not important to the localdb value.

**Add Connection** ? X

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

**Data source:**  
Microsoft SQL Server (SqlClient) Change...

**Server name:**  
(localdb)\v11.0 Refresh

**Log on to the server**

☒ Use Windows Authentication  
☐ Use SQL Server Authentication

User name:   
Password:   
☐ Save my password

**Connect to a database**

☒ Select or enter a database name:  
mylibrary

☐ Attach a database file:  
 Browse...

Logical name:

Advanced...

Test Connection OK Cancel

Fig. 6. Creating a database connection

At this stage, you can click the Test connection button in order to check the server accessibility. After that, think of a database name and enter it into the "Select or enter a database name" field. Click the OK button.

After this, you will see a connection to the created database in the Data Connections section of the Server Explorer window. Expand this node and select the Tables node. This node is empty now because there are no tables in a database. Let's create the required tables. It can be done in two ways: either in the graphic mode in the top of the window or using a query in the bottom of the window. Since we already have queries, let's use them. Copy a query for creating the Author table and paste it over the query template in the bottom of the window. Then, click the Update button in the top of the window and click the Update database button in the next window. Similarly, create two other tables.

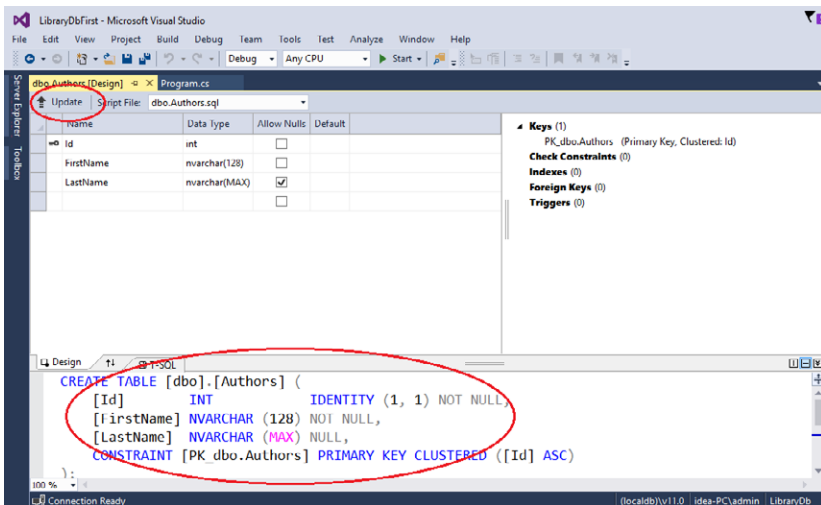


Fig. 8. Creating a database table

Now our database is ready, and we can return to the main topic of our conversation: Entity Framework.

### **EDM creation for Database first**

Now we should consider the process of using Entity Framework for a case of Database first. What does it mean? It means that we want to prepare the created empty application to work with the already existing database using Entity Framework. In other words, in our application, we should create entities, which are classes that correspond to our database tables, and a database context class in order to communicate with a database and execute database queries. In order to do this, you should perform the following actions:

1. Navigate to the Solution Explorer of the created project, select the project there, and enable the context menu (by clicking the right mouse button);
2. Select the Add-New item command;
3. In the appeared window, find and select the ADO.NET EDM Model item with the Model1.edmx default name;
4. Change the name to LibraryModel1.edmx, and click the Add button;
5. In the appeared Entity Data Model Wizard, select the Create from database option and click the Next button (Fig. 9);

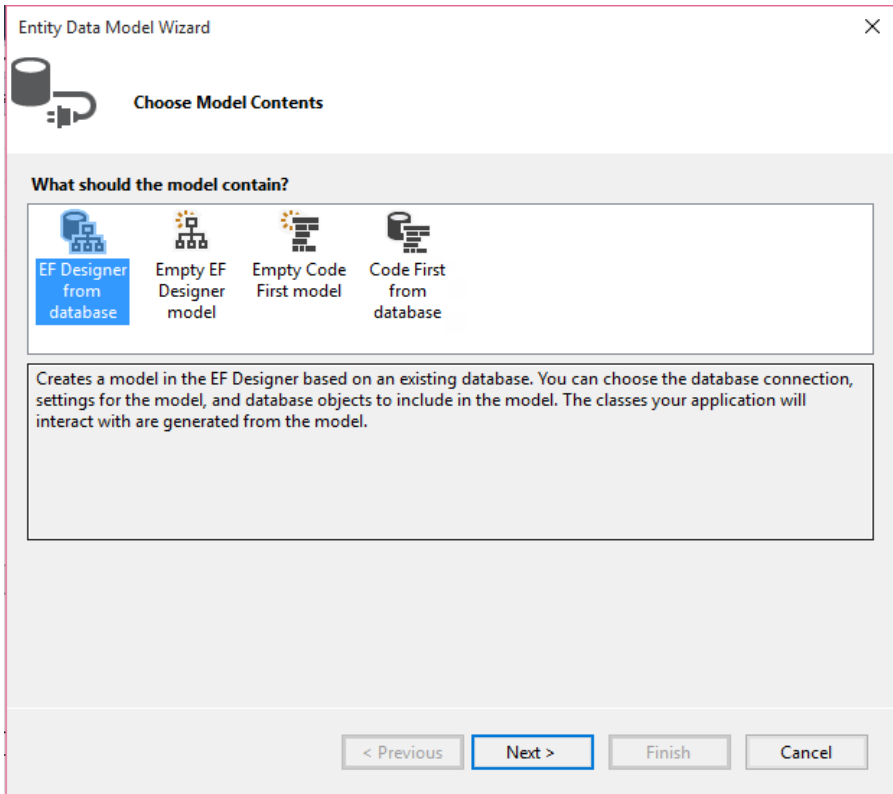


Fig. 9. EDM creation

- At this stage, it's necessary to either select an existing database connection or create a new connection. We select an existing database connection and specify the LibraryEntities name in the bottom window (Fig. 10.);

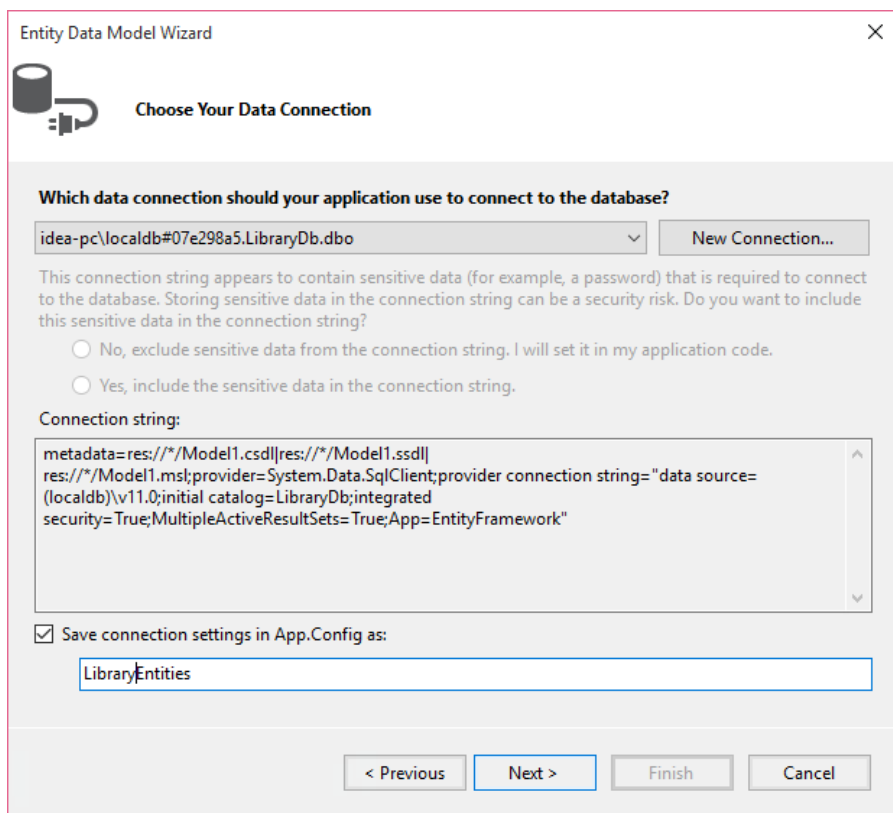


Fig. 10. Creating a connection

7. Click Next and get access to the window displayed in the Fig. 11;
8. In the next window, you will see the database structure in the tree list, here, you should expand the Tables node, mark the check boxes next to each of our three tables, and click the Finish button.

Now, Visual Studio will display the database diagram. In my case, it looks as follows (Fig. 12).

## Entity Data Model Wizard



## Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- ☒ Tables  
☐ Views  
☐ Stored Procedures and Functions

☐ Pluralize or singularize generated object names

☒ Include foreign key columns in the model

☐ Import selected stored procedures and functions into the model Includes foreign key columns as properties in the model

Model Namespace:

LibraryDbModel

&lt; Previous

Next &gt;

Finish

Cancel

Fig. 11. Selecting database components

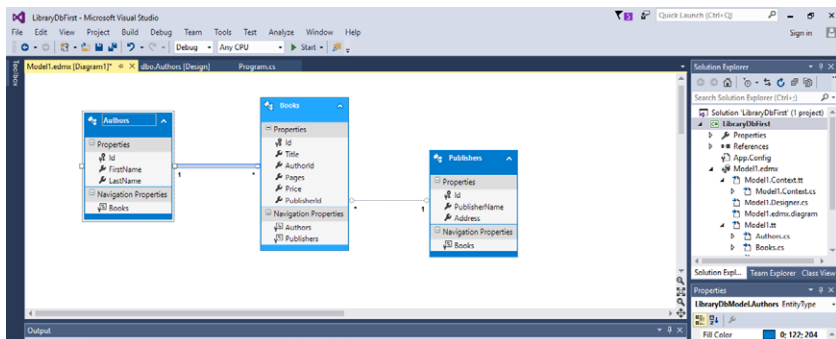


Fig. 12. EDM diagram

After these actions, you will see the created connection in the Server Explorer. You can expand the node for the created connection and see the connected database structure. The LibraryModel1.edmx object will appear in the Solution Explorer. We will consider its structure later.

So, we created a database and a console project, wherein we installed Entity Framework, and created Entity Data Model for our database. So, where does this leave us?

Let's recall what Conceptual Model and Storage Model in the Entity Data Model architecture are. If you remember what this is you should understand that after creating Entity Data Model, three classes named Author, Publisher, and Book appeared in our project. These classes are called entities, and they are created based on the database tables. The class was created for each database table in the project. The table fields turned into class properties with the same names as the table field names and with the relevant data types. Let's now perform some actions, which will demonstrate the Entity Framework features and then, we will discuss the results and talk about the inner mechanism. Add the following two methods next to the Main() method:

```
static void AddAuthor(Author author)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        db.Author.Add(author);
        db.SaveChanges();

        Console.WriteLine("New author added:" +
            author.LastName);
    }
}
```



```

}

static void GetAllAuthors()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        var au = db.Author.ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+" "+
                              a.LastName);
        }
    }
}

```

Call these methods in the Main() method, for example, as follows:

```

static void Main(string[] args)
{
    Author author = new Author{ FirstName="Isaac",
                                LastName="Azimov"};

    AddAuthor(author);
    GetAllAuthors();
}

```

Run the application, and you will see that the "Isaac Azimov" record appeared in the Author table. Let's consider the created methods carefully. I think that first of all, you are interested in the following code string of the added methods:

```

LibraryEntities db = new LibraryEntities()

```

Expand the LibraryModel1.edmx node in the Solution Explorer, expand the LibraryModel1.Context.tt node inside of

it, and, finally, inside of the last node, select the LibraryModel1.Context.cs file, wherein the LibraryEntities class is defined. We talked about the role of this class in Entity Framework, and now, you see it in your project. In my case, this class looks as follows:

```
public partial class LibraryEntities: DbContext
{
    public LibraryEntities()
        : base("name=LibraryEntities")
    {
    }

    protected override void
    OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Author> Author { get; set; }
    public DbSet<Book> Book { get; set; }
    public DbSet<Publisher> Publisher { get; set; }
}
```

It's important to note that DbContext inherits the IDisposable interface, therefore, it's convenient to use a context object in the using block. In addition, it's useful to know that the definition of this class can be changed manually.

Open the App.config configuration file in the Solution Explorer and find a database connection string in it. Entity Framework created this configuration file and a connection string. As you see, the connection string name corresponds to what is used in the LibraryEntities constructor and is transferred to the constructor of the DbContext basic class.

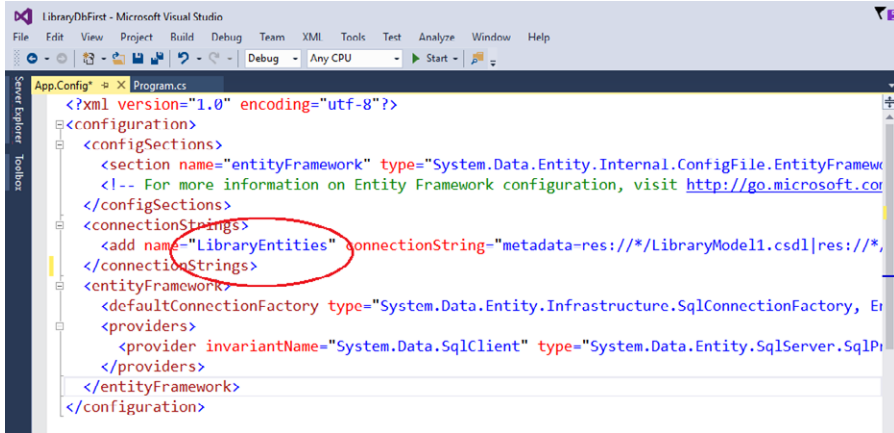


Fig. 13. Application configuration file

As you understand, the `AddAuthor()` and `GetAllAuthors()` methods are simply illustrations, which prove the Entity Framework efficiency. Of course, it's necessary to modify these methods slightly in order to work with them more conveniently. For example, it would be better if the `AddAuthor()` method would check whether there is already information about a new author in the database before adding this information there. In addition, it would be useful to have a method that returns not a whole list of authors, but only the selected author, for example, according to an `Id`. Any of these modifications requires writing certain database queries. Therefore, let's consider Linq to Entities in more detail in order to execute the required queries effectively.

# LINQ to Entities (introduction)

---

## **First() и FirstOrDefault()**

For simplicity, all below examples will be considered in relation to the Author database table and to the Author application entity. Let's assume that you want to retrieve the information about the author named "Charles" from the table, herewith, the first found record, which correspond to your search criteria, will satisfy you. In this case, you should use the First() or FirstOrDefault() methods. You should remember that when using Linq, you can write a code both in a query format and in a method format. If you use the method format, all the parameters of methods should be written using lambda-expressions. Let's consider and discuss two options of the method that allows you to get the first record about the author, whose name coincides with the value of the fname parameter, from a database table.

The first option is written in the query format:

```
static Author GetAuthorByName(string fname)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var author = (from s in db.Author
                      where s.FirstName == fname
                      select s).FirstOrDefault<Author>();
        return author;
    }
}
```

The second option is written in the method format:

```
static Author GetAuthorByName1(string fname)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var author = db.Author.Where( (x) =>
            x.FirstName == fname).FirstOrDefault();
        return author;
    }
}
```

Both methods should return an object of the Author type. You shouldn't be misled by the fact that we described the author object type using the var specifier. You already have to get used to C# extensions. Let's talk about the First() and FirstOrDefault() methods and about the difference between them. Both methods are designed to select a single table record (first found) that satisfies the selection criteria. In our case, this is `s.FirstName == fname`. The difference between them appears only when there is no such record in the table. In this case, First() throws an exception, and FirstOrDefault() returns null.

### **Single() и SingleOrDefault()**

Let's say that now you want to retrieve the information about the author with the Id value equal to 2 from the table, herewith, you understand that either there is a single record in the table, or there are no records at all. In order to solve this task, you can use the Single() or SingleOrDefault() methods. These methods work in such a way in order to return a record that satisfies the specified selection criteria and herewith, to verify that it's actually

a single record. Let's assume that we search for records with Id=2 in the table with million strings. Let's suppose that it's the second string from the beginning of the table. If we have searched using the First() method we would obtain the result after processing the second table string. If you use the Single() method, it will find the search string after processing the second table string as well. But this method WILL NOT STOP TO WORK even if the search string is found. It will continue to iterate the whole million strings in order to make sure that the found record is a single record that satisfies the selection criteria. And if it finds one more string that satisfies the selection criteria, it will throw an exception. Single() and SingleOrDefault() behave differently, when there is no element that satisfies the selection criteria: the first one throws an exception, the second one returns null. This is a main difference between them. The relevant methods can look as follows in two forms of writing:

- in the query format:

```
static Author GetAuthorById(int id)
{
    using (LibraryEntities db = new
LibraryEntities())
    {
        var author = (from s in db.Author
                        where s.Id == id
                        select s).Single();
        return author;
    }
}
```

- in the method format:

```
static Author GetAuthorById1(int id)
{
    using (LibraryEntities db = new
        LibraryEntities())
    {
        var author = db.Author.Where( (x) =>
            x.Id == id).SingleOrDefault();
        return author;
    }
}
```

## ToList()

If you want to retrieve all table strings that satisfy the selection criteria you can use the ToList() method. For example, we want to input a list of all authors, whose surname begins with the letter "A":

- in the query format:

```
static void GetAllAuthors()
{
    using (LibraryEntities
        db = new LibraryEntities ())
    {
        var au = db.Author.Where((x) =>
            x.LastName.StartsWith("A")).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+" "
                +a.LastName);
        }
    }
}
```

- in the method format:

```
static void GetAllAuthors1()  
{  
    using (LibraryEntities db = new LibraryEntities ())  
    {  
        var au = (from a in db.Author  
                   where a.LastName.StartsWith("A")  
                   select a).ToList();  
        foreach (var a in au)  
        {  
            Console.WriteLine(a.FirstName+" "+a.  
                               LastName);  
        }  
    }  
}
```

## OrderBy()

If you want to sort the obtained result, you can use the `OrderBy()` method. For example, we want to input a list of all authors sorted by surnames.

- in the query format:

```
static void GetAllAuthors()  
{  
    using (LibraryEntities db = new LibraryEntities ())  
    {  
        var au = (from a in db.Author orderby  
                   a.LastName ascending select a).ToList();  
        foreach (var a in au)  
        {  
            Console.WriteLine(a.FirstName+"  
                               "+a.LastName);  
        }  
    }  
}
```



- in the method format:

```
static void GetAllAuthors1()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Author.OrderBy((x) =>
            x.LastName).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+"
                               "+a.LastName);
        }
    }
}
```

## Find()

If you want to find and get one specific object, you can use the Find() method. For example, we want to input the author's name by its Identifier.

In the method format:

```
static Author GetAuthorById(int id)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Author.Find(id);
        Console.WriteLine(au.FirstName + " " +
            au.LastName);
        return au;
    }
}
```

In our example, the Find() method searches the author according to the specified unique key. If an object is not found according to the specified unique key, Find() returns null. You should know one interesting feature of this method. You should

already understand that there can be a pause between creating an entity object and adding this object to a database. Entity is added to a database by calling the `db.SaveChanges()` method. So, the `Find()` method searches objects not only in a database, but in RAM, that is, it searches objects that are not added to a database yet.

It will be enough for us to get acquainted with these methods for creating those applications, through which we will get acquainted with the use of Entity Framework. We will get acquainted with other methods later.

# Filling the database

It's enough to consider these methods in order to enter records into all three database tables. In order to do this, you should add the below methods to the application.

Adding a new publishing house:

```
static void AddPublisher(Publisher publisher)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        Publisher a = db.Publisher.Where((x) =>
            x.PublisherName == publisher.
            PublisherName).FirstOrDefault();
        if (a == null)
        {
            db.Publisher.Add(publisher);
            db.SaveChanges();
            Console.WriteLine("New publisher
                               added:" + publisher.PublisherName);
        }
    }
}
```

Adding a new book:

```
static void AddBook(Book book)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        Book a = db.Book.Where((x) => x.Title ==
            book.Title).FirstOrDefault();
        if (a == null)
        {
            db.Book.Add(book);
        }
    }
}
```

```

        db.SaveChanges();
        Console.WriteLine("New book added:" +
            book.Title);
    }
}

```

## Filling all tables:

```

static void Init()
{
    Author author = new Author { FirstName =
        "Ray", LastName = "Bradbury" };
    AddAuthor(author);
    author = new Author { FirstName = "Harry",
        LastName = "Harrison" };
    AddAuthor(author);
    author = new Author { FirstName = "Clifford",
        LastName = "Simak" };
    AddAuthor(author);

    Publisher publisher = new Publisher {
        PublisherName = "Rainbow", Address = "Kyiv" };
    AddPublisher(publisher);
    publisher = new Publisher { PublisherName =
        "Exlibris", Address = "Kyiv" };
    AddPublisher(publisher);
    Book book = new Book { Title = "Way Station",
        IdPublisher = 1, IdAuthor = 4,
        Pages = 350, Price = 85 };
    AddBook(book);
    book = new Book { Title = "Ring Around the
        Sun", IdPublisher = 1, IdAuthor = 4,
        Pages = 420, Price = 99 };
    AddBook(book);
    book = new Book { Title = "The Martian
        Chronicles", IdPublisher = 2, IdAuthor = 2,
        Pages = 410, Price = 105 };
}

```

```
AddBook(book);  
book = new Book { Title = "I, Robot",  
    IdPublisher = 3, IdAuthor = 1,  
    Pages = 378, Price = 100 };  
AddBook(book);  
}
```

Now, it's necessary to call the `Init()` method in the `Main()` method, and all database tables will be filled, and we will be able to deepen the understanding of the Entity Framework principles. Perform the specified actions, re-build and execute the modified application in order to fill all tables.

Let's talk about the application we have made. This application is created according to the "Database First" technology. You should understand that it doesn't matter, where your database is created, it's only important that it already exists, and the application uses the existing database for creating classes, which will correspond to database tables. You should note that all entities, the database context class, and the database connection string are created by Entity Framework. We, as developers, created only the applied logic. In our case, this logic is very simple, but you can now implement other use cases by yourself as well.

# Navigation properties

Now, let's look at the information stored in the Book table. To do this, you can create the following simple method:

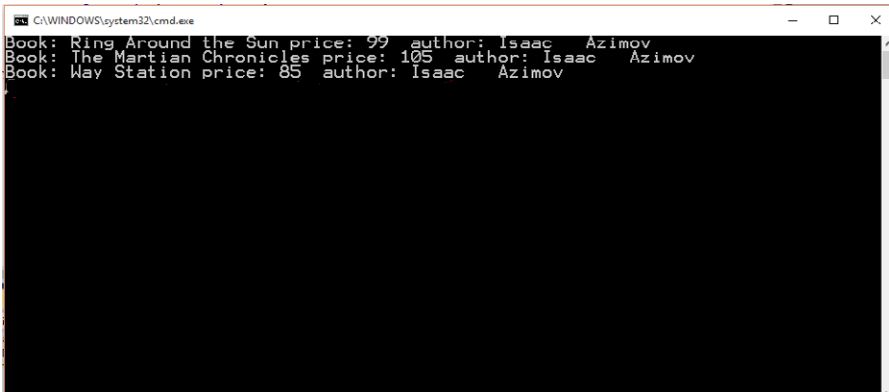
```
static void GetAllBooks()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Book.OrderBy((x) => x.Title).
            ToList();

        foreach (var a in au)
        {
            Console.WriteLine("Book: "+a.Title + "
                                price: " + a.Price + "  author: "+
                                a.Author.FirstName + "    "+a.Author.
                                LastName);
        }
    }
}
```

Add the call of this method to Main() and run our application. You shouldn't be afraid of that when executing the Init() method in the database, the duplicated information will be entered into the Author, Publisher, and Book tables. The AddXXX() methods check now the added information and prevent the duplication in the database.

```
static void Main(string[] args)
{
    Init();
    GetAllBooks();
}
```

When the application is executed, you will see the following window.



```

C:\WINDOWS\system32\cmd.exe
Book: Ring Around the Sun price: 99 author: Isaac Azimov
Book: The Martian Chronicles price: 105 author: Isaac Azimov
Book: Way Station price: 85 author: Isaac Azimov

```

Fig. 14. Displaying the GetAllBooks() method

Look at this figure and the GetAllBooks() method code carefully. Don't you see anything strange? Pay attention, we output the information from the Book table but, however, we see the authors' name and surname. But there is no information about the author's name and surname in the Book table. This information is stored in the Author table, and the Book table stores only the author's id. Keep this in mind. The fact is that there is no information about the author's name and surname in the Book database table, but our query works with the Book entity, that is, with the class created on the base of the Book table. But there is information about the author in the Book entity. Moreover, all information is here. Climb a little higher on the text to the Figure 12 that displays the EDM diagram. This diagram presents the created entities, and you can see that the Book entity has the so-called Publisher and Author

"navigation properties". Navigation properties are data of the related tables that are transferred entirely and stored in the entity. These properties are created on the base of table foreign keys, that is, on the base of analysis of relationships between the tables. Navigation properties allow us to avoid using multi-table queries in order to obtain data of the related tables.

Are you wondering how navigation properties are described in the entity definition? Expand the LibraryModel1.edmx node in the Solution Explorer, after that, expand the LibraryModel1.tt node, and, finally, click on the Book.cs file. In my case, the Book entity definition looks as follows:

```
public partial class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int IdAuthor { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int IdPublisher { get; set; }

    public virtual Author Author { get; set; }
    public virtual Publisher Publisher { get; set; }
}
```

Pay attention that the Author and Publisher navigation properties are described as virtual. Also, it's interesting to note the description of the Pages and Price properties. Their type is not simply int, but Nullable<int>. It's done in the case if there is no suitable value in the table, and it will be necessary to return null. Herewith, the IdPublisher property type is not expanded to Nullable<int>. Because this field is



a database foreign key, and, consequently, can't contain null. The created entity corresponds exactly to the Book database table in everything else.

### *New terms*

- ***Navigation properties*** are the entity properties containing data of the related tables. These properties are created by Entity Framework automatically.

# Home task

---

Create a Windows Forms application to work with our database using Entity Framework according to the "Database First" technology. The main application window should contain the TabControl element with three tabs: Books, Authors, and Publishers. Each tab should serve one of the database tables and add, delete, and edit records in its table. Use forms in order to add, delete, and edit records. Use DataGridView in order to display results of the GetAllAuthors(), GetAllBooks() and GetAllPublishers() methods.