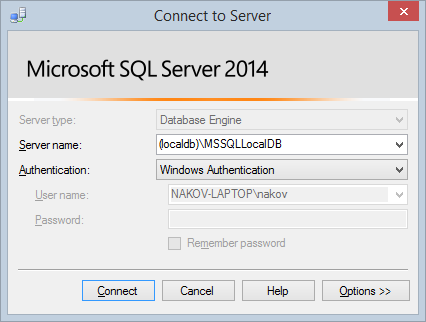
# Database Applications Lab (March 2015) – Geography

The goal of this lab is to learn how to develop database applications with C#, Entity Framework and SQL Server. You will create database, map the database to EF data model (database first and code first), query the database, import and export data, parse and create XML and JSON.

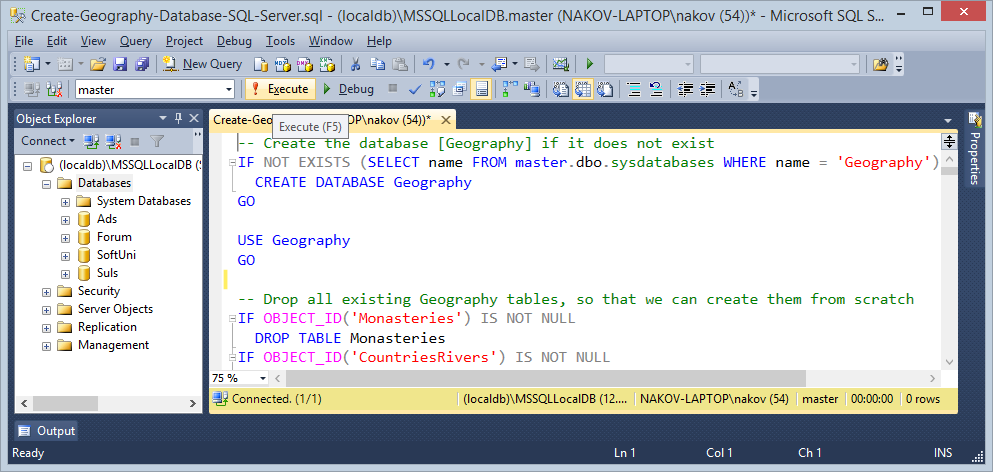
## Problem 0. Restore the Database

You are given a **MS SQL Server database "**Geography**"** holding continents, countries, currencies, monasteries and rivers, available as **SQL script**. **Restore the database "**Geography**"** by running the provided SQL script.

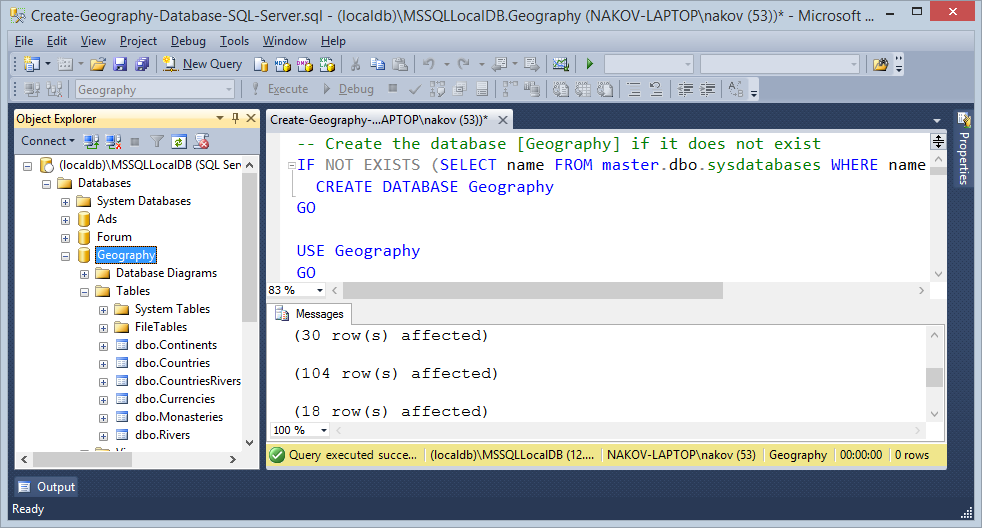
1. **Connect to MS SQL Server** using SQL Server Management Studio. You may use MS SQL Express Edition, MS SQL Developer Edition of MS SQL LocalDB (version 2012, 2014 or later). Example (connecting to SQL 2014 LocalDB):



1. Run the SQL script to create the database schema and load sample data in the tables (drag and drop the file "Create-Geography-Database-SQL-Server.sql" into the SQL Server Management Studio and execute it):

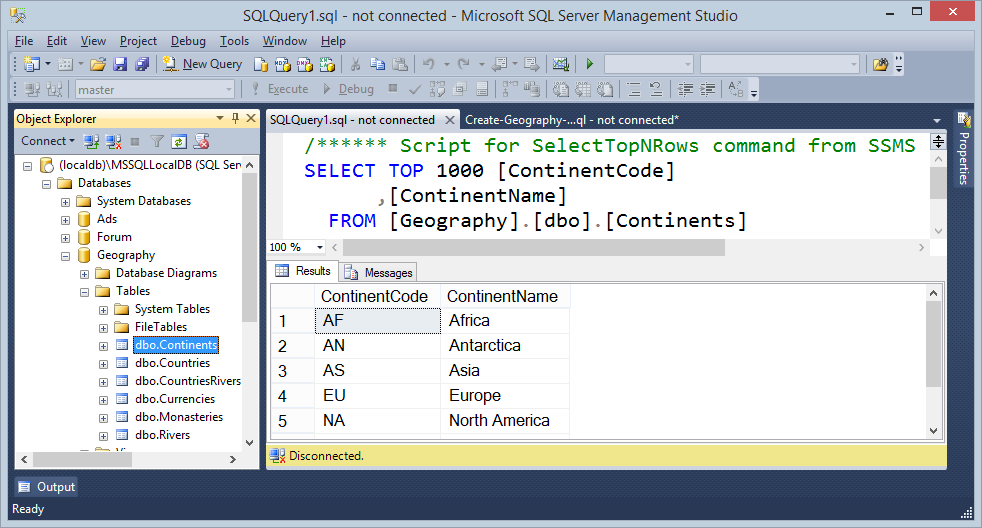


The SQL script should **execute without errors**:



1. Refresh the Databases. You should see the database "Geography". It should hold several tables (Continents, Countries, Currencies, Monasteries, Rivers and CountriesRivers).

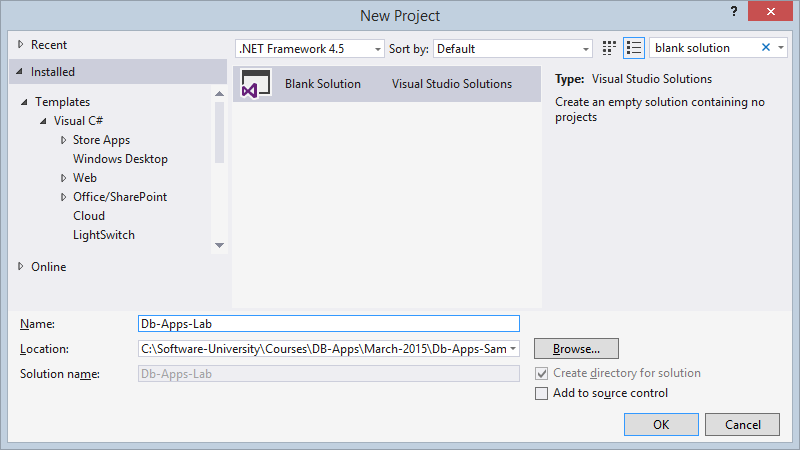
Ensure you have data in all tables:



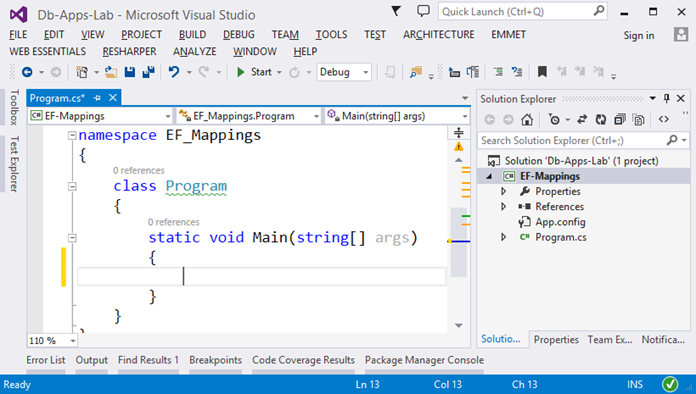
## Problem 1. Entity Framework Mappings (Database First)

Create an **Entity Framework (EF) data model** of the existing database "Geography" (map the database tables to C# classes). Use the "**database first**" model in EF. To test your EF data model, **list all continent names**.

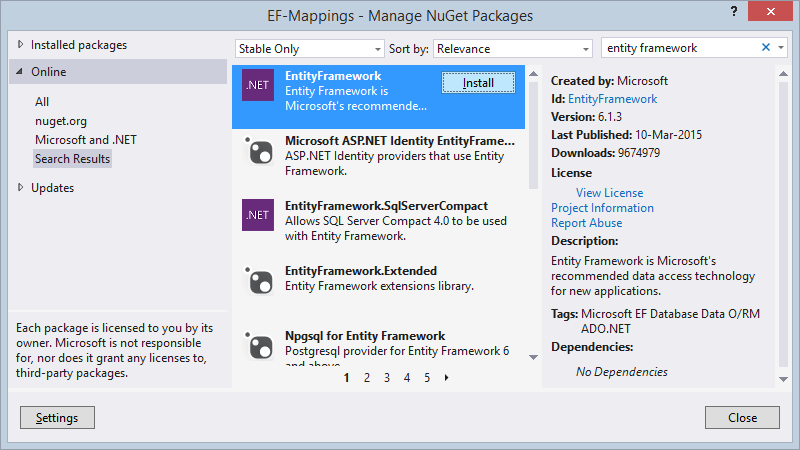
1. Start Visual Studio. Use Visual Studio 2013 or later. Create a **new Blank Solution** called"Db-Apps-Lab":



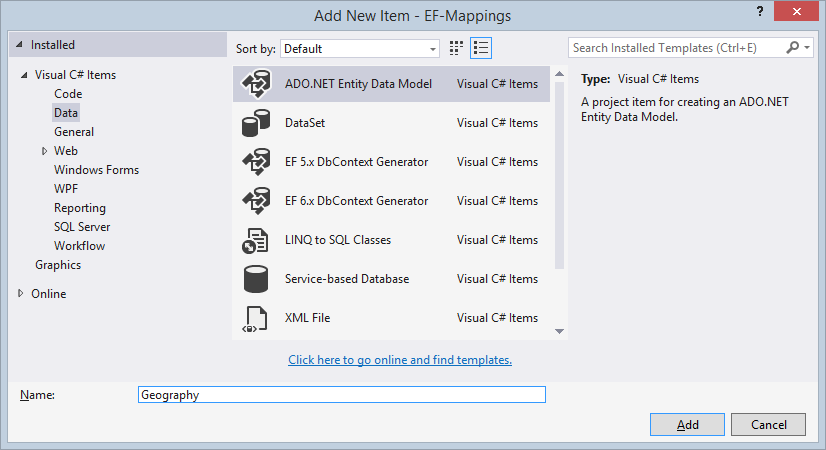
1. **Create a new C# Console Application** in your blank VS solution called "**EF-Mappings**":



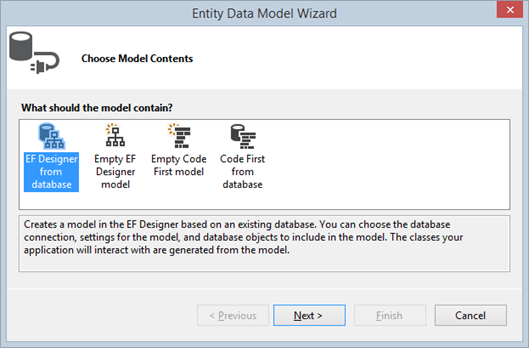
1. Add a **reference to "Entity Framework"** through the **NuGet** Package Manger in Visual Studio:



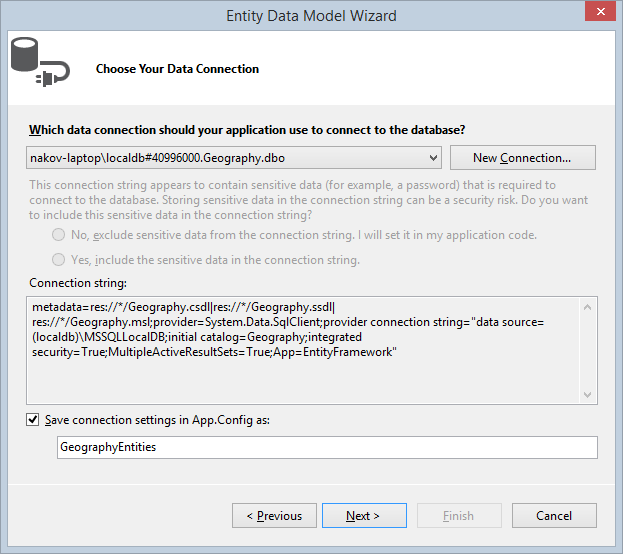
1. Add a new ADO.NET **Entity Data Model** named "Geography":



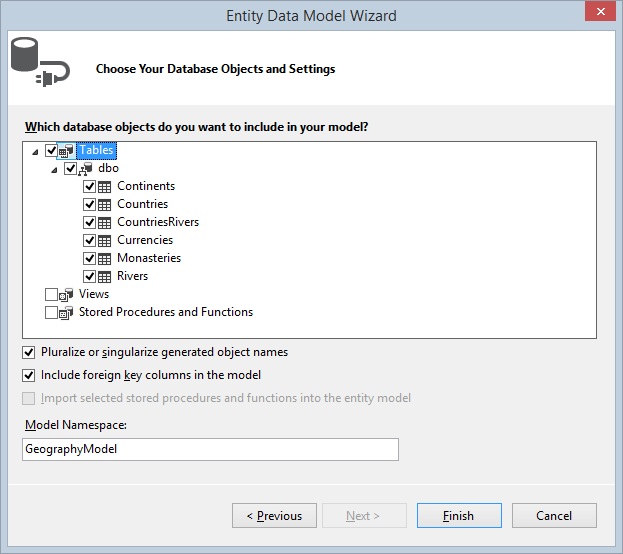
1. Create a database first EF mappings (choose the **EF Designer from existing database**):



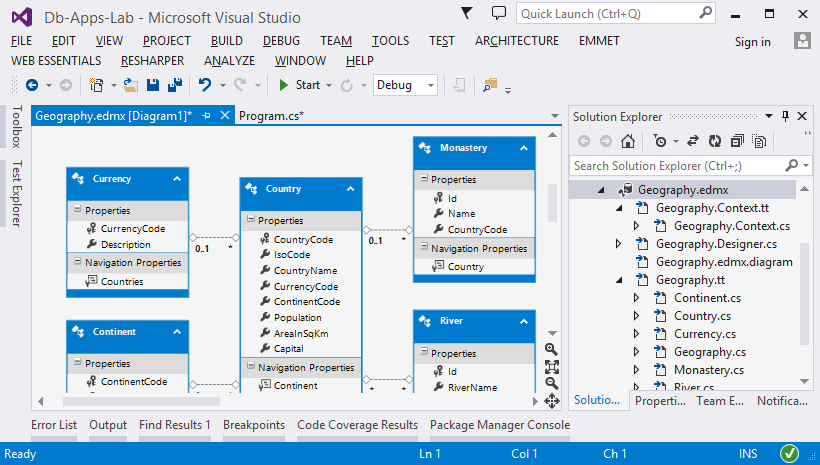
1. Create a database connection to your "Geography" database in MS SQL Server:



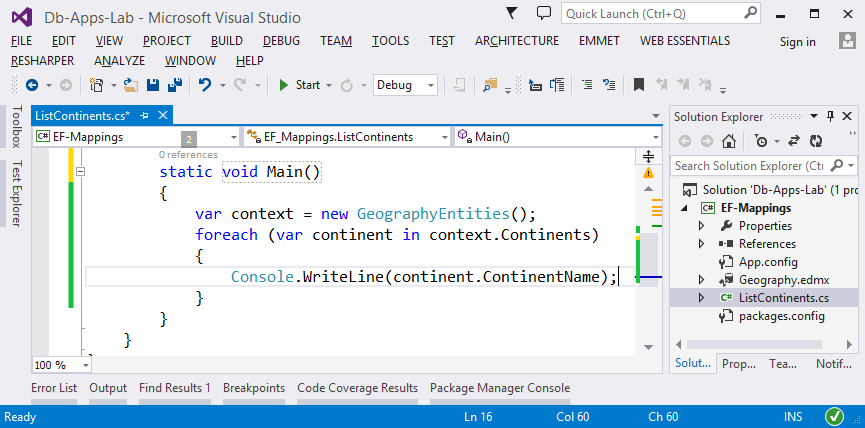
1. **Select all tables** to be mapped in the EF data model:



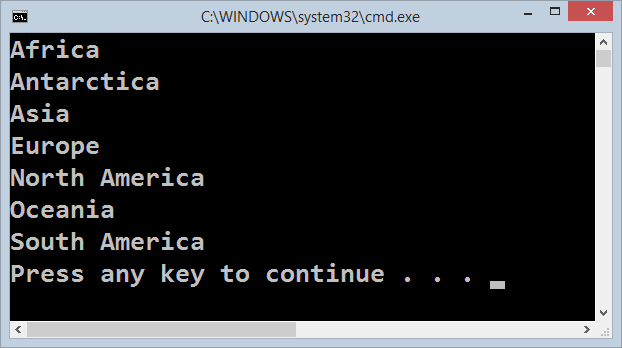
1. Visual Studio will create for you a EF database first data model (EDMX file):



1. To **list all continent names**, write some code:



1. **Run your program**. It should list the continent names from the database:



1. Rename the class "Program" to "ListContinents". Also, rename the file name. Rename your project from "EF-Mappings" to "1. EF-Mappings" (this is the solution of the first lab problem and you will have more).

## Problem 2. Export Rivers as JSON

Write a **C# application** based on your EF data model for **exporting all rivers along with their countries** in the following JSON format:

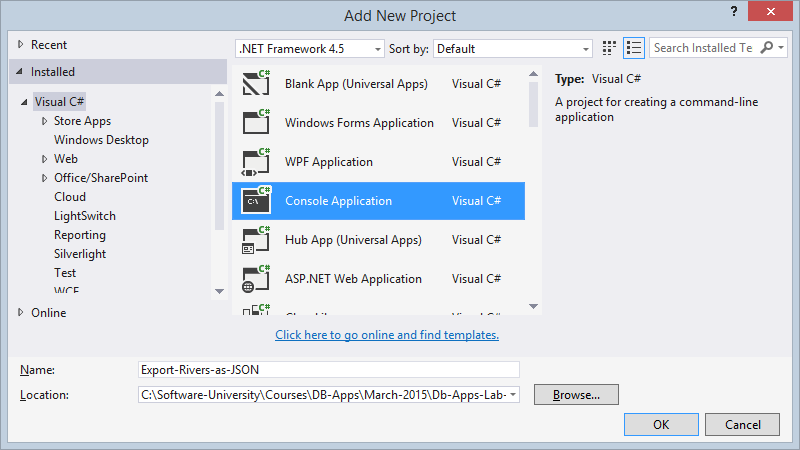
|  |
| --- |
| **rivers.json** |
| [  { "riverName": "Nile", "riverLength": 6650, "countries": ["Burundi","Democratic Republic of the Congo","Egypt","Eritrea","Ethiopia","Kenya","Rwanda","South Sudan","Sudan","Tanzania","Uganda"] },  { "riverName": "Amazon", "riverLength": 6400, "countries": ["Bolivia","Brazil","Colombia","Ecuador","Guyana","Peru","Venezuela"] },  { "riverName": "Yangtze", "riverLength": 6300, "countries":["China"] },  …  ] |

Write the output in a JSON file named rivers.json. Include in the output the rivers with no countries (if any). The JSON file code formatting is not important.

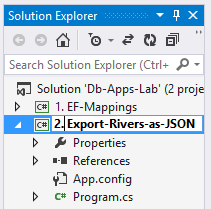
Order the **rivers** by **length** (from the longest) and the countries for each river **alphabetically**.

For better performance, ensure your program executes a **single DB query** and retrieves from the database only the required data (without any unneeded rows and columns).

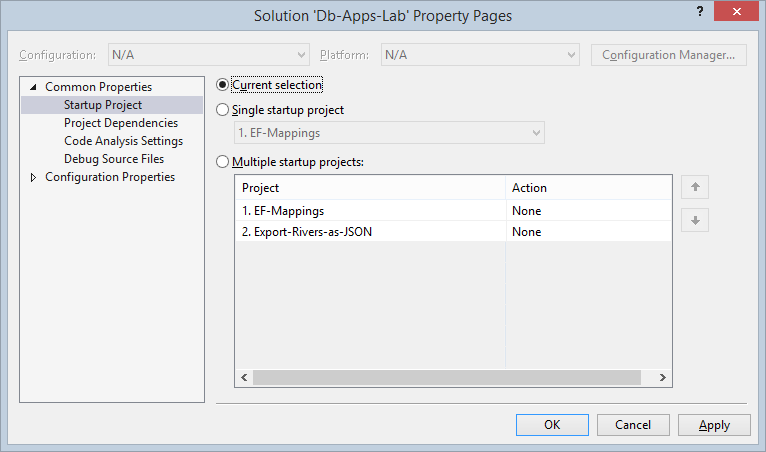
1. Create a **new Console Application** called "Export-Rivers-as-JSON"in your VS solution:



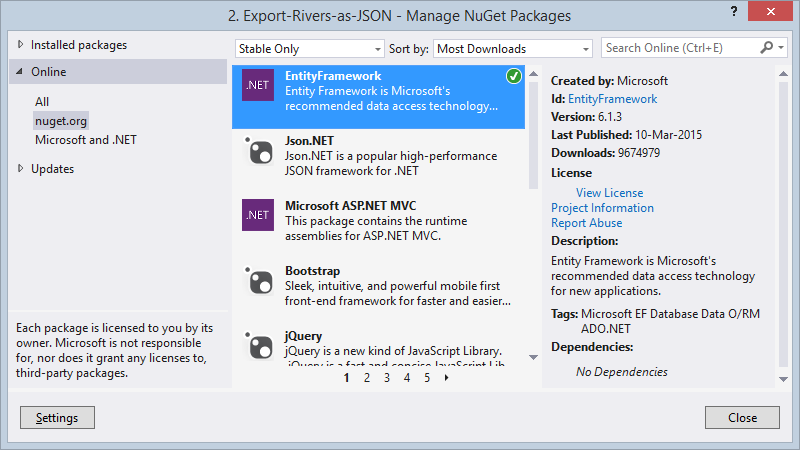
1. Rename the project from "Export-Rivers-as-JSON"to "2. Export-Rivers-as-JSON":



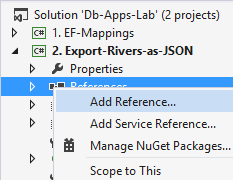
1. Set the current VS project as startup project:

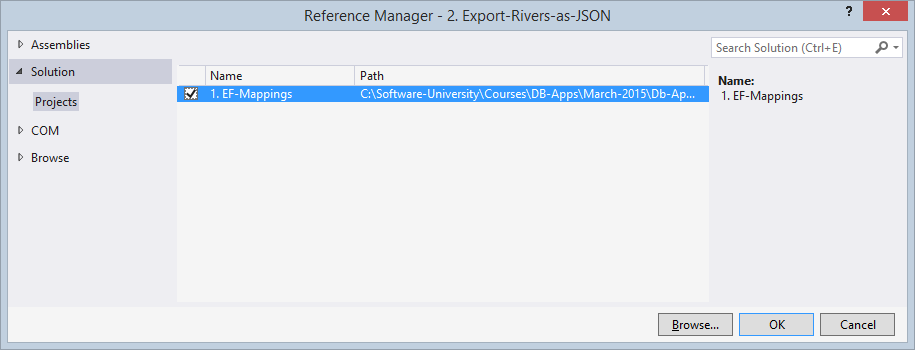


1. Add a **NuGet reference** to the package "**Entity Framework**":

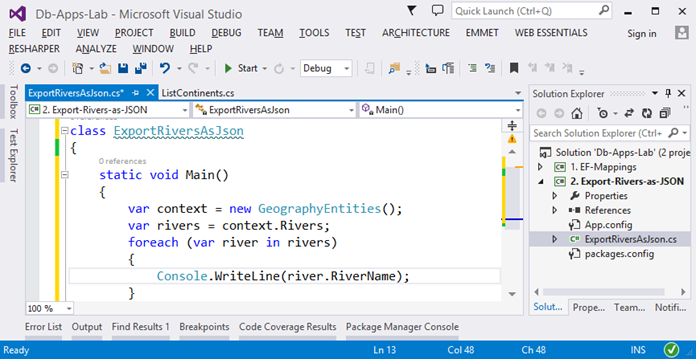


1. To reuse the EF data model from the previous exercise, **add a reference** to the project holding the previous problem "1. EF-Mappings":

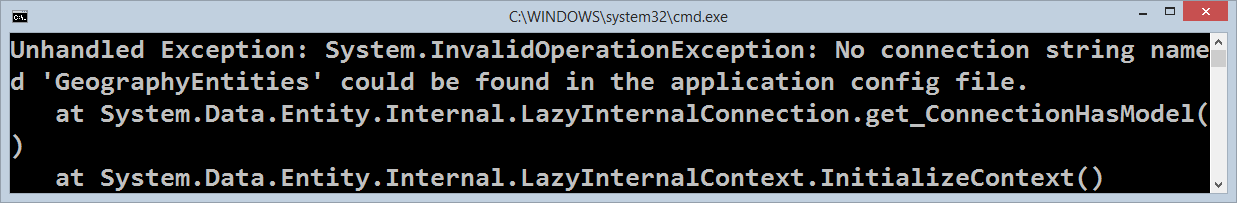




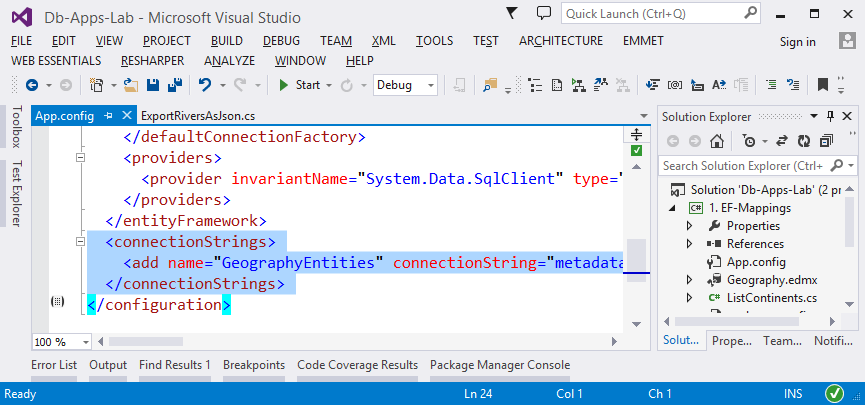
1. Rename the class "Program" to "ExportRiversAsJson".
2. Write some C# code to **list all rivers**:



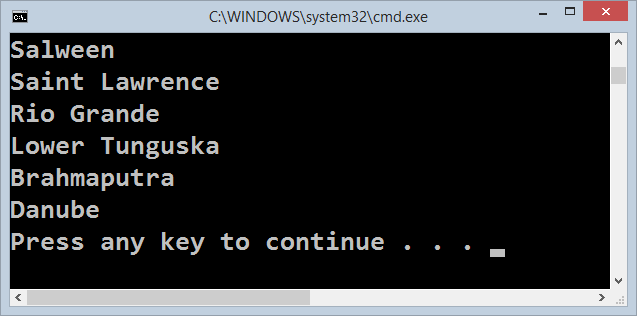
1. Run the program to see the **"No connection string" exception**:



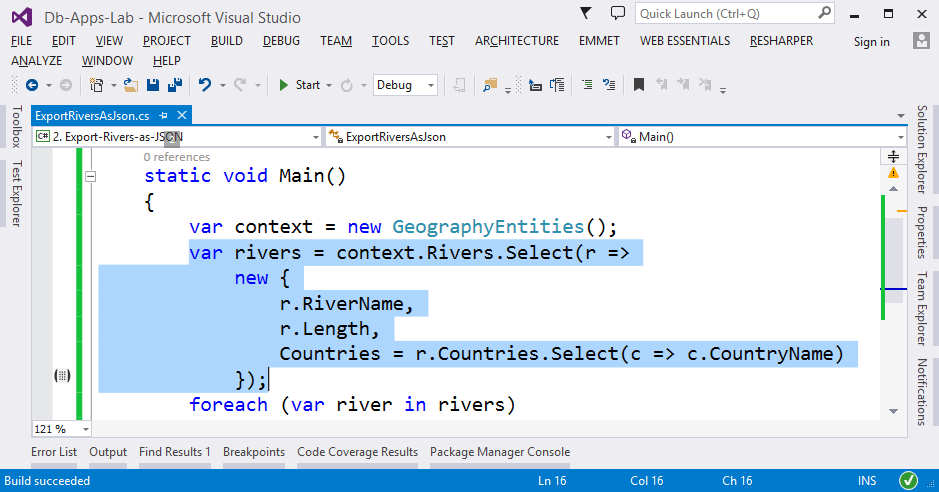
1. Add the **connection string** settings to your App.config file or copy the entire App.config file from the previous example:



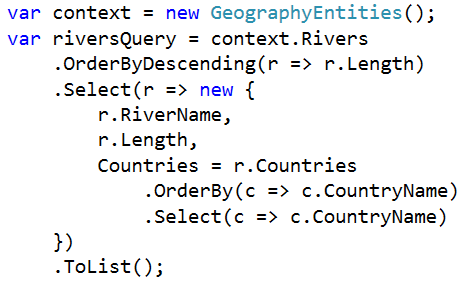
1. Now run your program again. It should **list all rivers**:



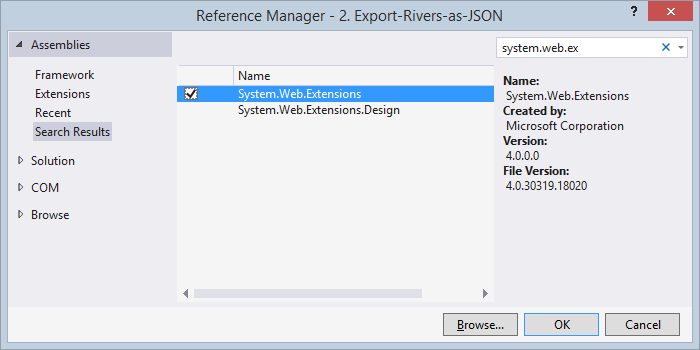
1. Write a **LINQ query** to select all rivers, their name, length and countries:



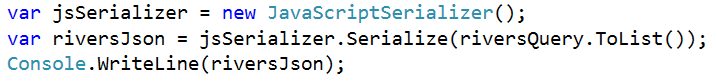
1. **Test your query** by running the program and checking the output.
2. Add the requested **sorting by river length** (from longest) and **sorting by country name** alphabetically. Finally materialize the query to execute it and get the results from the DB by invoking ToList():



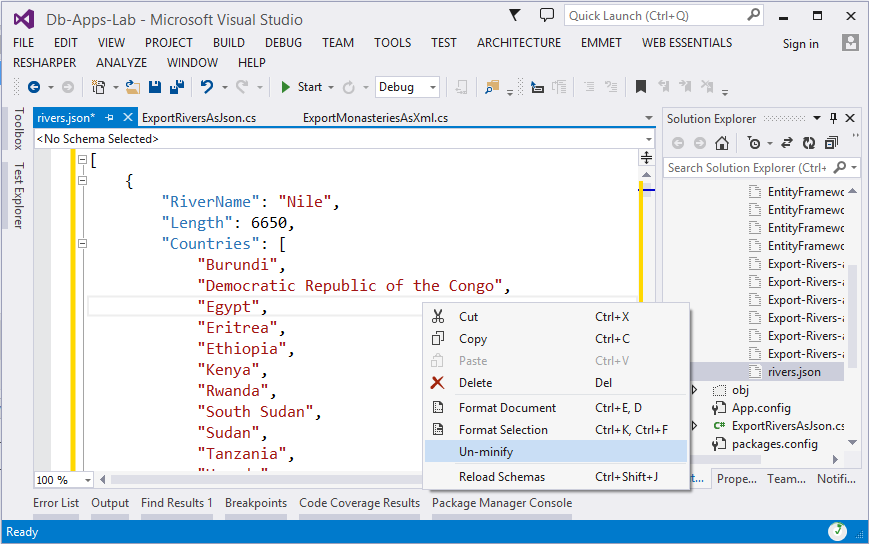
1. **Print the results** to the console with a foreach loop.
2. Finally, you have to **convert the results to JSON**. You can use the JavaScriptSerializer. First add a reference to the .NET assembly System.Web.Extensions.dll:



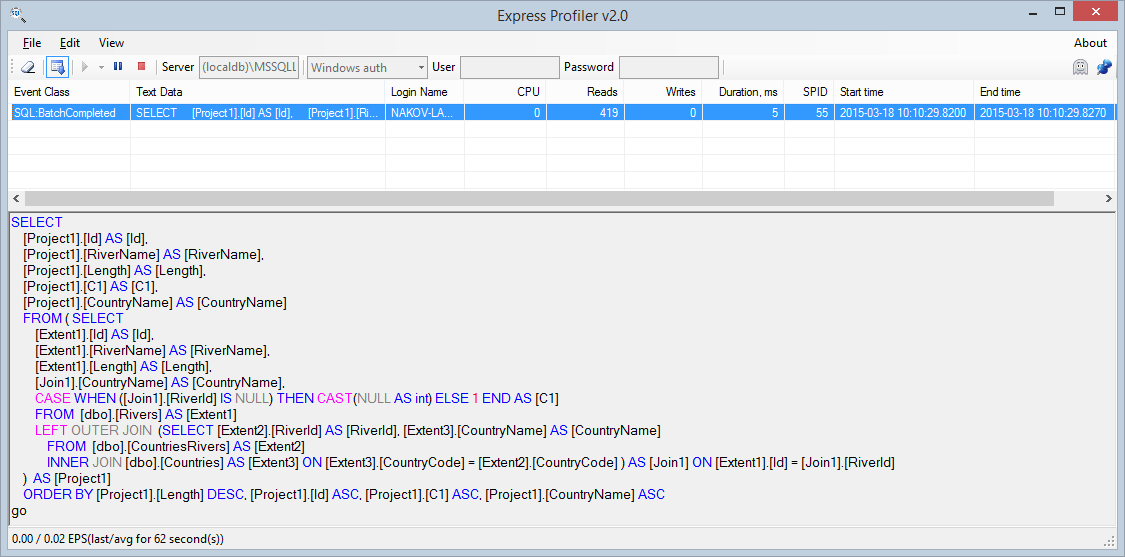
1. Use the following code to **serialize the results as JSON** string:



1. Finally, save the result JSON string to a text file named "rivers.json". Find in Internet how to "write text to file in C#". You may use File.WriteAllText(…).
2. Run and **test your program**. Open the output file bin\Debug\rivers.json in Visual Studio and check whether is holds the correct results as expected. **Un-minify the JSON** to check it for correctness:



1. Check for **performance problems** and ensure your program executes a **single SQL query**. You might use the **SQL Express Profiler** to see all executed queries:



## Problem 3. Export Monasteries by Country as XML

Write a **C# application** based on your EF data model for **exporting all monasteries by country** to a XML file named monasteries.xml in the following XML format:

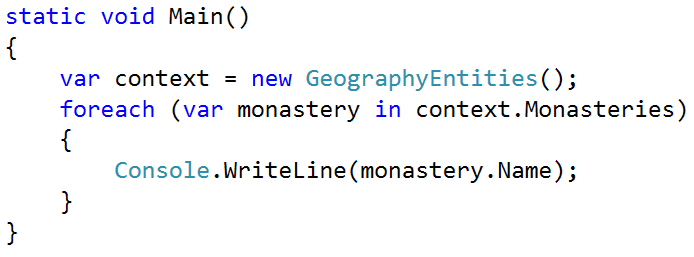
|  |
| --- |
| **monasteries.xml** |
| <?xml version="1.0" encoding="utf-8"?>  <monasteries>  <country name="Bhutan">  <monastery>Taktsang Palphug Monastery</monastery>  </country>  <country name="Bulgaria">  <monastery>Bachkovo Monastery “Virgin Mary”</monastery>  <monastery>Rila Monastery “St. Ivan of Rila”</monastery>  <monastery>Troyan Monastery “Holy Mother's Assumption”</monastery>  </country>  …  </monasteries> |

Exclude all countries with no monasteries. Use an XML parser by choice.

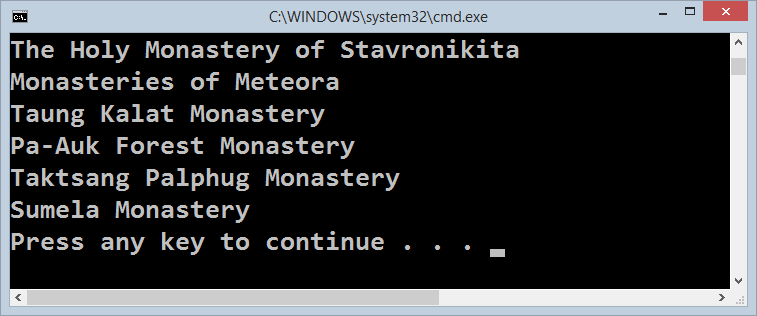
Order the **countries** **alphabetically** and the **monasteries** in each country also **alphabetically**.

For better performance, ensure your program executes a **single DB query** and retrieves from the database only the required data (without unneeded rows and columns).

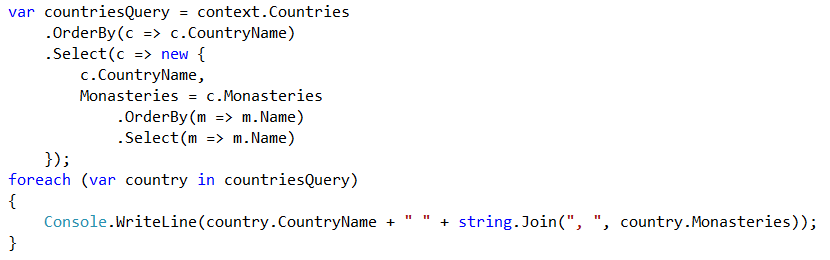
1. Create a **new Console Application** called "Export-Monasteries-as-XML"in your VS solution.
2. Rename the project from "Export-Monasteries-as-XML"to "3. Export-Monasteries-as-XML".
3. Add a **NuGet reference** to the package "**Entity Framework**".
4. Add a reference to the project holding EF data model "1. EF-Mappings".
5. Rename the class "Program" to "ExportMonasteriesAsXml".
6. Copy the App.config from the project "1. EF-Mappings". Thus, you will copy the DB connection string.
7. Write some code to **list all monasteries**, for example:



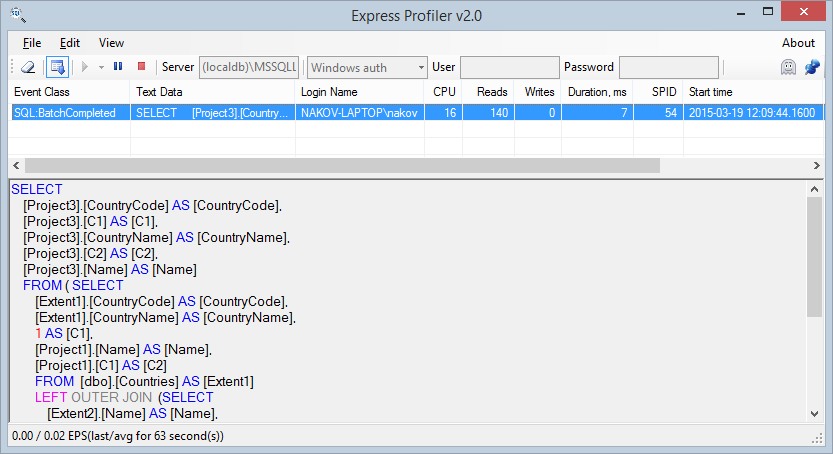
1. Run and **test your code**. Ensure the monasteries are listed correctly:



1. Write a query to select all the **monasteries alphabetically** along with all their **countries alphabetically**. To **test your query**, print the monasteries and their countries to the console. Your code might look similar to this:



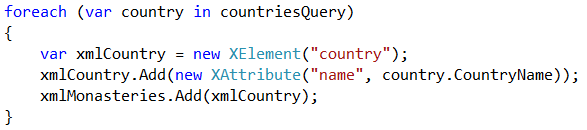
1. Exclude from the query the countries with no monasteries. You may use a .Where(…) filter along with monasteries.Any().
2. **Test again** to ensure all countries with their monasteries are listed correctly.
3. Check for **performance problems** and ensure your program executes a **single SQL query** (and the N+1 query problem is avoided). You might use the **SQL Express Profiler** to see all executed queries:



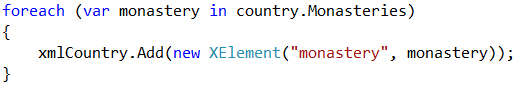
1. Now you have to **build the output XML**. The easiest way is to use the XElement class (the LINQ to XML parser). You can start with the root element:



1. Then you can **iterate through all countries** and append each country as child XElement:



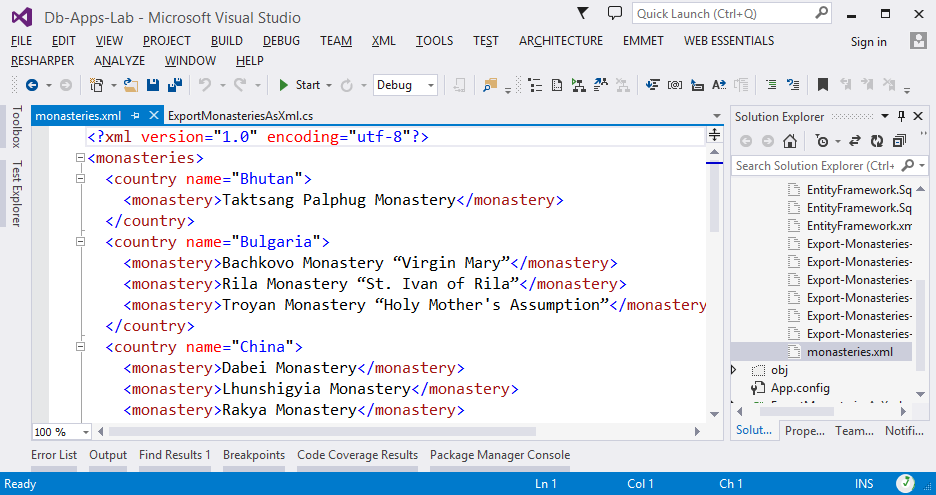
1. As next step you could **iterate though all monasteries** in each country and attach the current monastery as child XElement:



1. Print the output XML on the console to ensure it is correct.
2. Finally save the XML to file:



1. Check the output file bin\Debug\monasteries.xml:



## Problem 4. Import Rivers from XML

Write a **C# application** based on your EF data model for **importing into the DB a set of rivers** given in the XML file rivers.xml. The rivers come in the following XML format:

|  |
| --- |
| **rivers.xml** |
| <?xml version="1.0" ?>  <rivers>  <river>  <name>Maritsa</name>  <length>480</length>  <outflow>Aegean Sea</outflow>  <countries>  <country>Bulgaria</country>  </countries>  </river>  <river>  <name>Madre de Dios</name>  <length>1130</length>  <drainage-area>125000</drainage-area>  <average-discharge>4915</average-discharge>  <outflow>Beni River</outflow>  <countries>  <country>Peru</country>  <country>Bolivia</country>  </countries>  </river>  …  </rivers> |

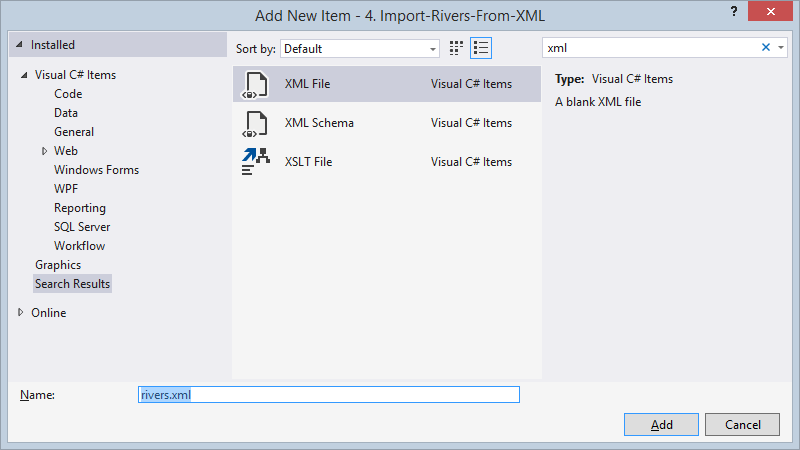
The name, length and outflow elements are **mandatory**. The drainage-area, average-discharge and countries elements are **optional**.

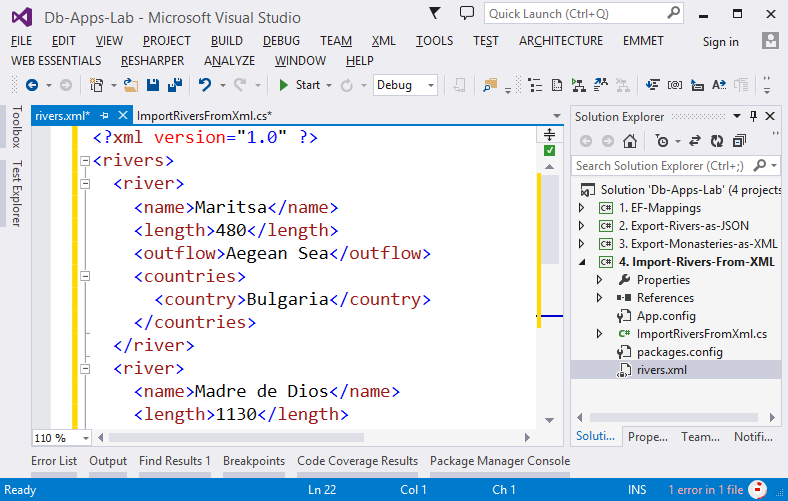
You should **parse the XML** and throw an **exception** in case of incorrect data, e.g. when a required element is missing or an invalid value is given. The size of the XML file will be less than **10 MB**. Use an XML parser by choice.

1. Create a **new Console application** named "Import-Rivers-From-XML" in your VS solution.
2. Rename the application to "4. Import-Rivers-From-XML".
3. Add a **NuGet reference** to the package "**Entity Framework**".
4. To reuse the EF data model, **add a reference** to the project "1. EF-Mappings".
5. Rename the class "Program" to "ImportRiversFromXml". Rename also the file "Program.cs" to "ImportRiversFromXml.cs".
6. Replace your local App.config file with the App.config from the project "1. EF-Mappings". This will copy transfer your connection string settings.
7. To **test whether you have correctly configured EF**, the DB connection string and you have database access, you can query for the number of rivers from the database and print the result on the console:



1. The next step is to **parse the XML** input file. First, create a sample file rivers.xml and put inside the rivers from the example:





1. As a next step, **load the XML** and print it on the console. You might use the XDocument parser:

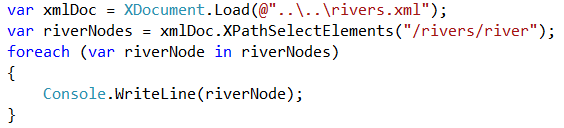


1. **Test your program** to ensure you have loaded the XML correctly.

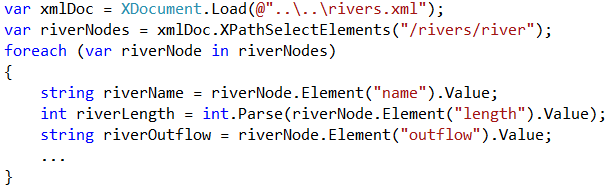
Your compiled project (.exe file) is located in bin\Debug folder, so if you want to load the rivers.xml from the root of your C# project, it should be accessed through the path "..\..\rivers.xml".

If the XML **fails to parse**, check whether is it correct. In the problem description, the sample XML is unfinished (it holds "…" at the last line), so you need to finish it.

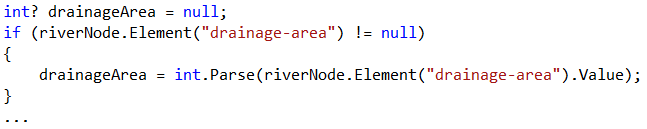
1. Now you have loaded the XML in the memory as XDocument. The next step is to iterate through all rivers. You may select them with an XPath selector "**/rivers/river**". Then, you can print the selected nodes:



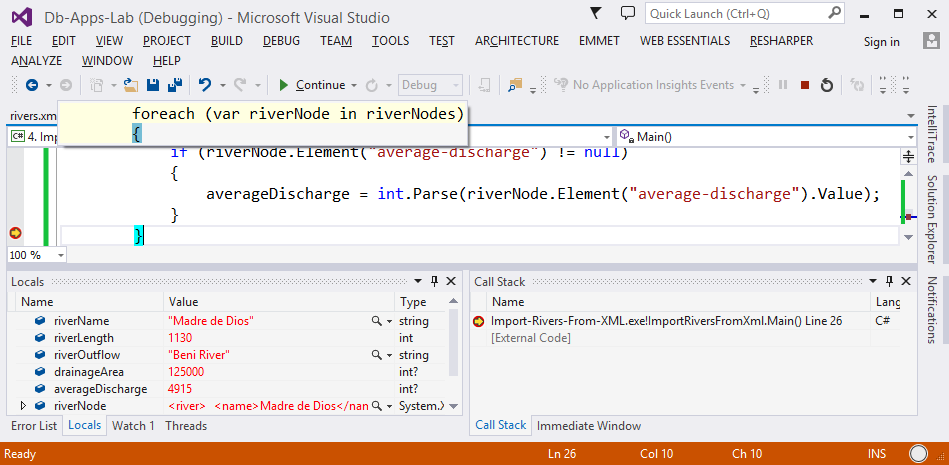
1. **Test your code** to ensure it selects and prints all rivers correctly.
2. Now you should extract the data from each river. First, **extract the mandatory fields**:



1. Next, **extract the optional fields:** drainage area and average discharge. Each optional field should be explicitly checked for null:



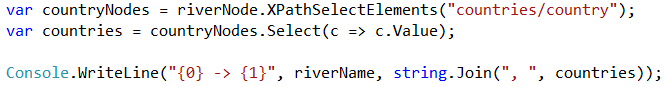
1. **Test your code**. The easiest way is to put a breakpoint and ensure the river data is correctly parsed:



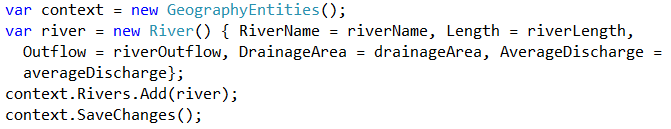
Alternatively, you can print the parsed river data on the console:



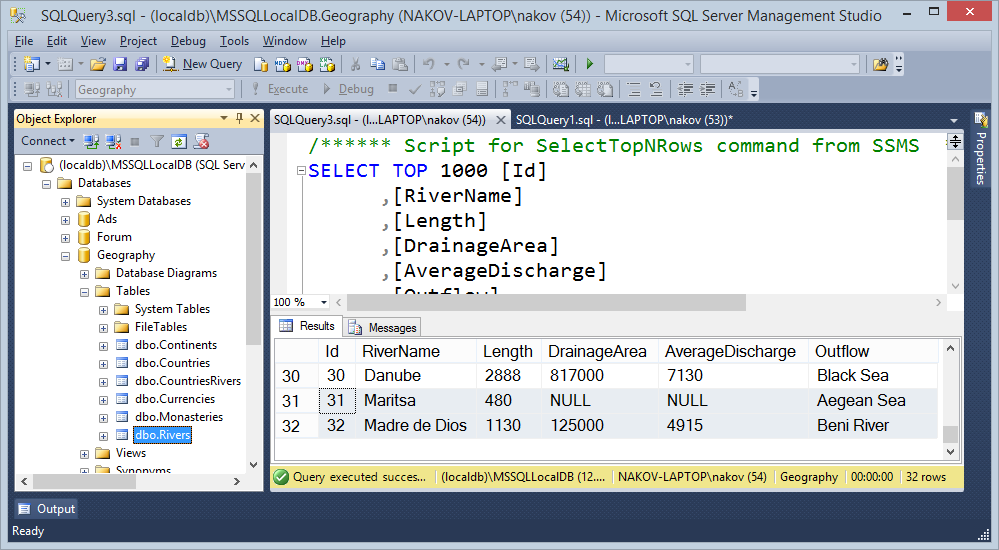
1. Now **parse the countries for each river** in the same way, with an XPath selector "countries/country", then print the rivers with its countries on the console:



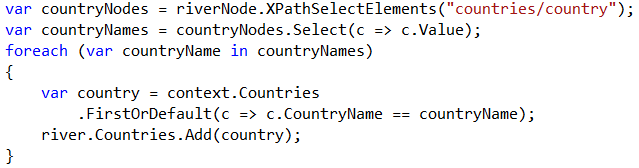
1. **Test your project** to ensure the rivers data is correctly parsed.
2. The next step is to **import the parsed rivers into the database**. First create a River entity, fill its properties and save it to DB:



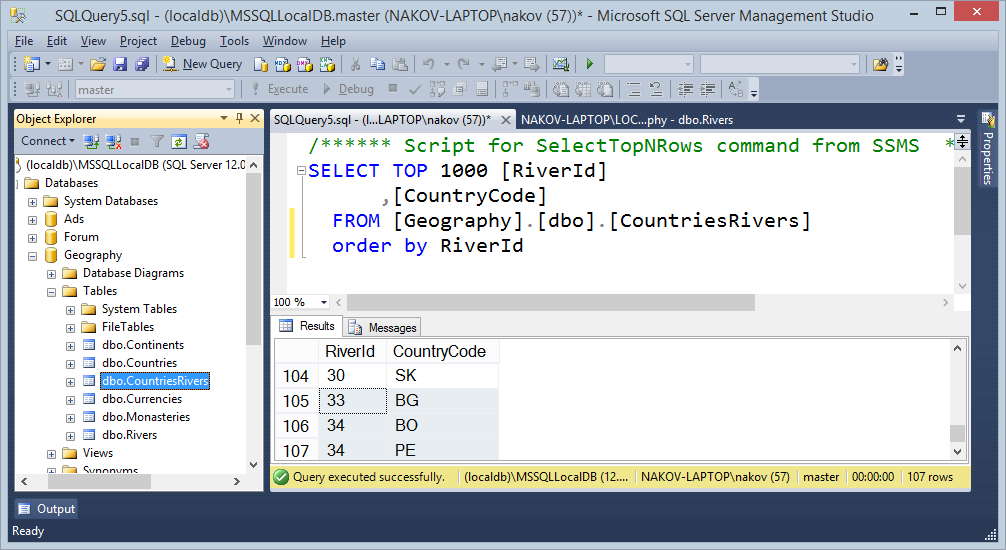
1. Check whether the **rivers are imported** in the database with SQL Server Management Studio:



1. Now write the code to **load the countries for each river** in the database. You need to find each country by its name and then add it to the current river:



1. Finally, run the program and check in the database whether the countries are correctly added to the imported rivers:

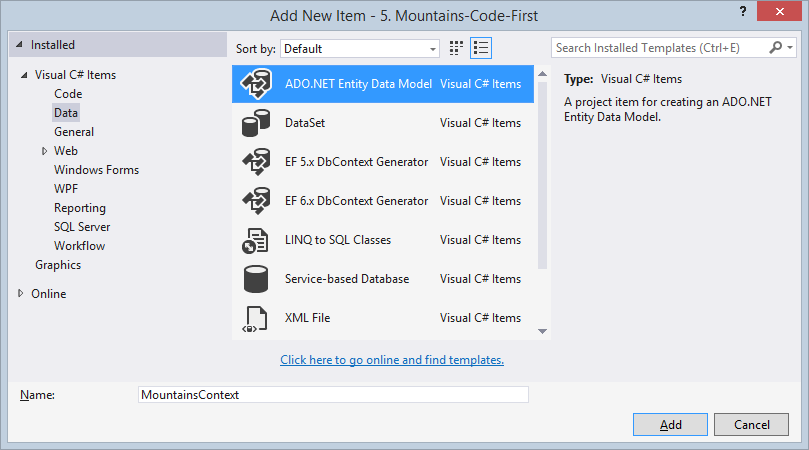


## Problem 5. EF Code First: Countries, Mountains and Peaks

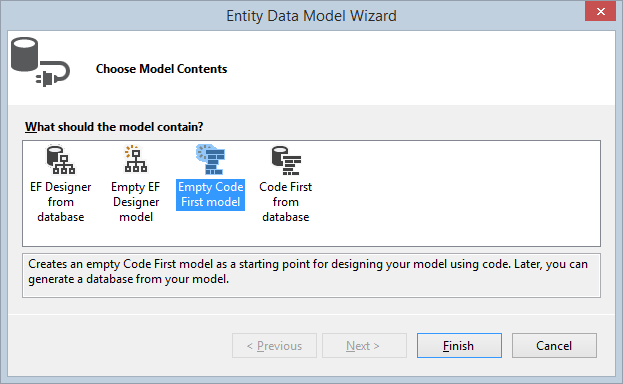
Create an **Entity Framework (EF) code first data model** for keeping countries, mountains and peaks.

* **Countries** have **country code** (2 Latin letters) and **country name**.
* **Mountains** have a **name** and belong to **multiple countries**.
* **Peaks** have a **name**, **elevation** and **mountain**.

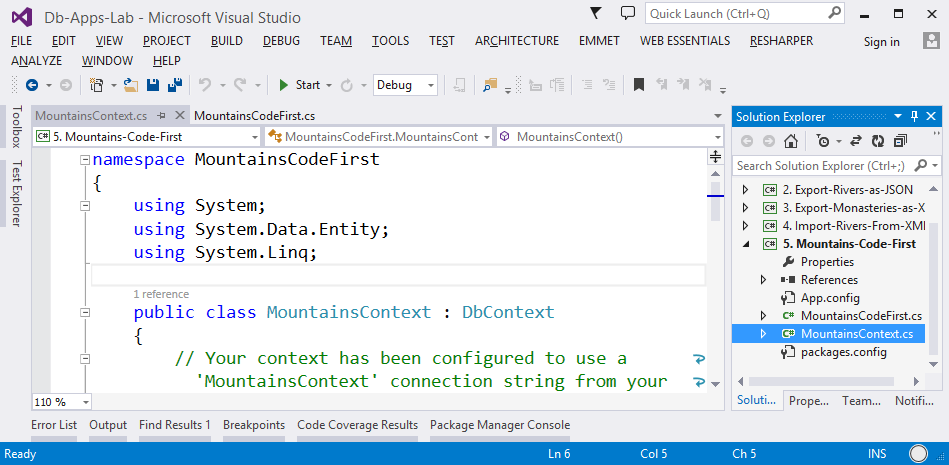
1. Create a **new Console application** named "Mountains-Code-First" in your VS solution.
2. Rename the application to "5. Mountains-Code-First ".
3. Add a **NuGet reference** to the package "**Entity Framework**".
4. Rename the class "Program" to "MountainsCodeFirst". Rename also the file "Program.cs" to "MountainsCodeFirst.cs".
5. Create a **new Entity Framework Code First Data Model**. Click on the VS project, select [Add] 🡪 [New Item…] 🡪 [Data] 🡪 [ADO.NET Entity Data Model]. Name it "MountainsContext":



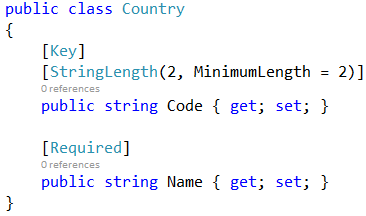
Choose "**Empty Code First model**":



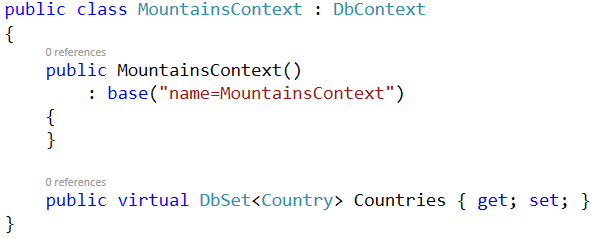
Visual Studio will generate for you a skeleton for your MountainsContext:



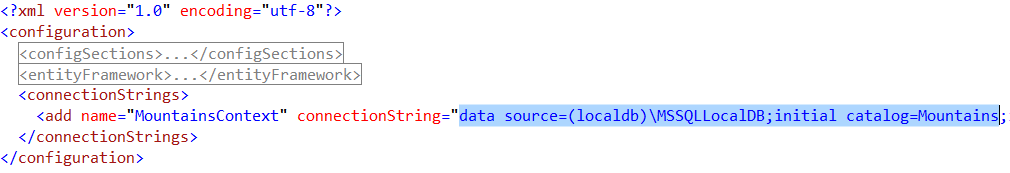
1. Now create your first entity class Country. It should hold **country code** (two characters, primary key) and **country name** (string, required). Entity classes should be public. The Country class might look like this:



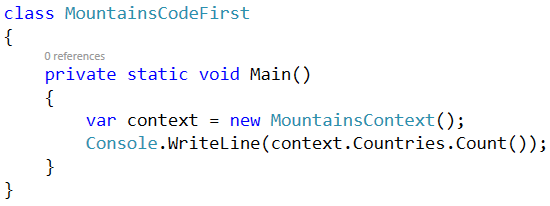
1. Add the Country entity in your DB context. First clean-up the MountainsContext class, then add a DbSet of countries:



1. The best way to continue is to **test the project**. First, configure the database **connection string** in App.config. It should hold the server name (e.g. "(localdb)\MSSQLLocalDB" or ".\SQLEXPRESS") and the database name (e.g. "Mountains"):

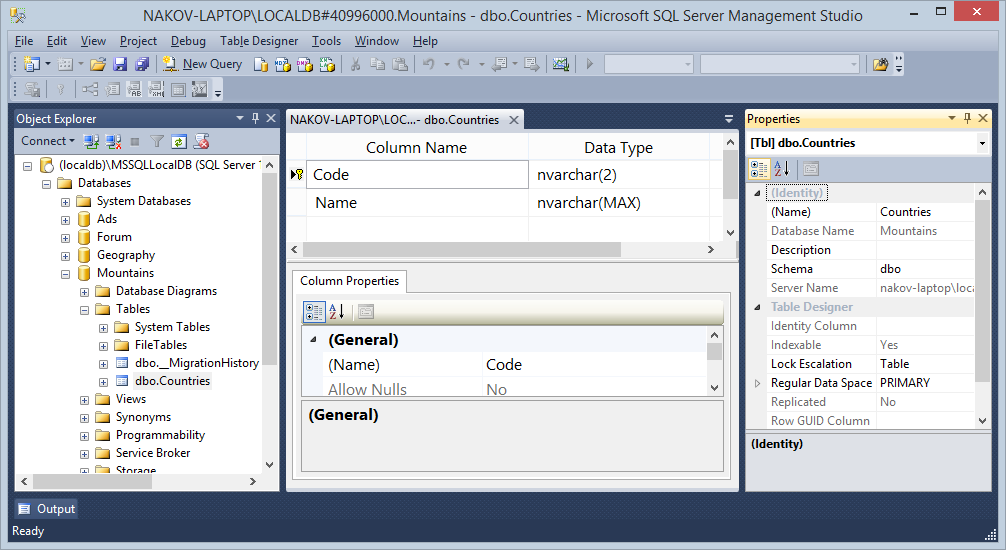


1. To **test the EF code first data model**, try to find the countries count from the database:



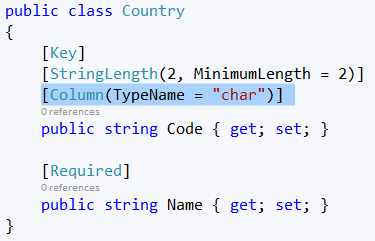
1. Run your application. It should print "0" on the console, without any error messages.

* If your application **runs without errors**, it should have created the database "Mountains" (see your connection string). Try to open the database and check whether the "Countries" tables is correctly defined. It should have columns "Code" (primary key) and "Name" of correct data type.

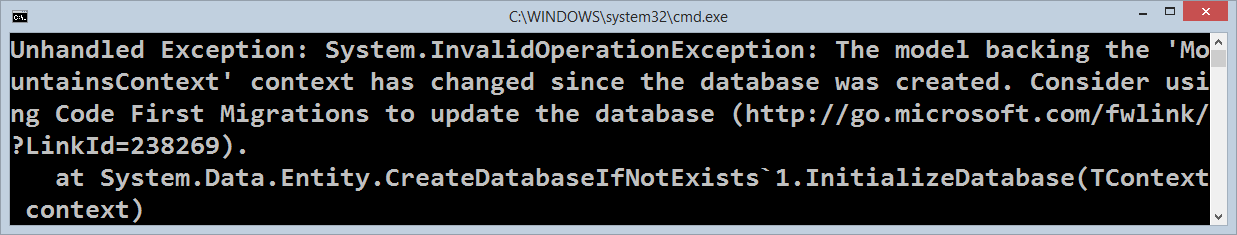


* If your application **fails with an exception**, this is most probably due to incorrect connection string. You can fix your connection string in your App.config file.

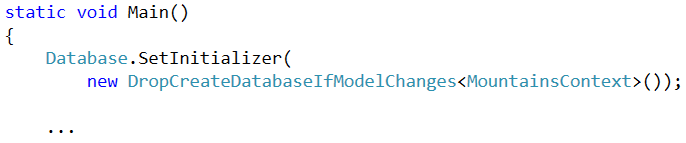
1. It looks like EF has generated the column Country.Code of type nvarchar(2) instead of char(2). This can be fixed by the configuration attribute [Column]:



1. Now **run the code** after the fix. It will fail with "***InvalidOperationException: The model has changed since the database was created."***. This is completely normal, because the project has not been configured to use "**Code First Migrations**":



1. By default the migration strategy in EF code first projects is CreateDatabaseIfNotExists. You can change this strategy with another: DropCreateDatabaseIfModelChanges. This EF database migration strategy will work well in our case, but avoid it in production systems, because it **drops the database with all data in all tables** each time when you modify the EF data model. To change the default DB migration strategy, put this code at the application start:

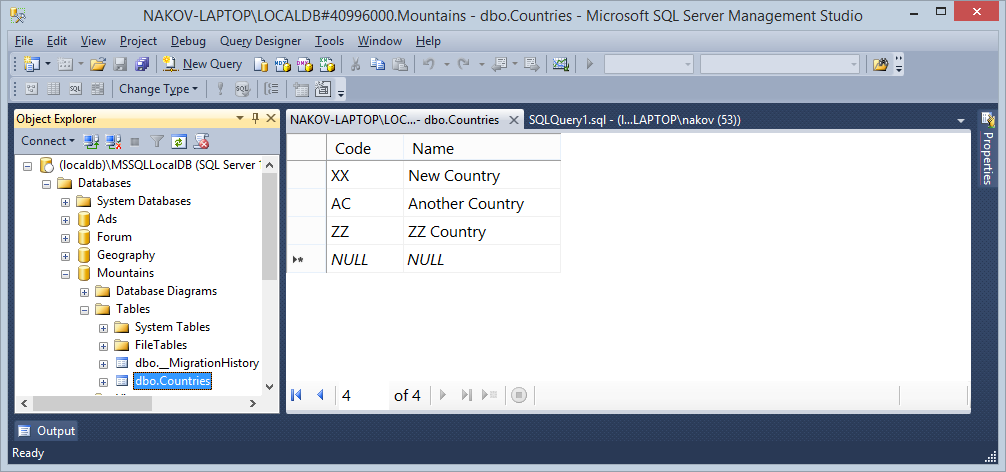


1. Now **run again your project**. It should **drop the existing database** and **re-create it** again to apply the changes in the EF code first data model.

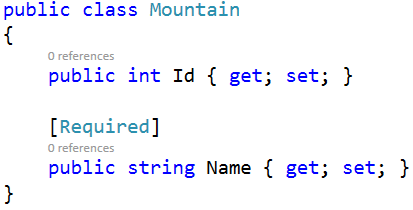
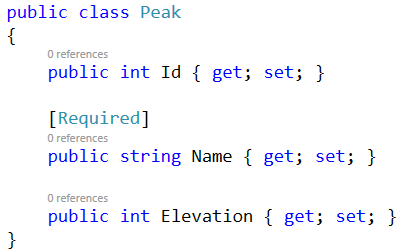
Sometimes, you could get an exception: ***System.Data.SqlClient.SqlException: Cannot drop database "Mountains" because it is currently in use***. If this happens, ensure you have closed SQL Server Management Studio and any other applications that use the database.

In the typical case, your program will run and will display "0" countries in the database, because it will be empty after the drop and create.

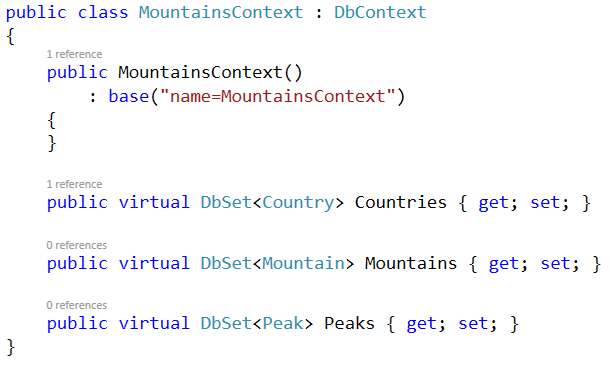
1. Now open the SQL Server Management Studio and **insert a few new records** in the Countries table:



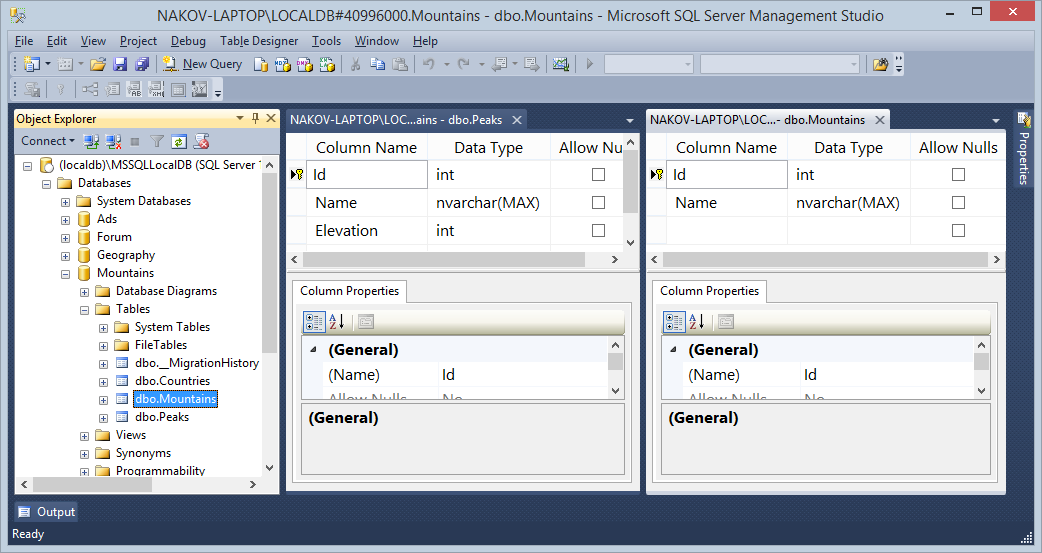
1. **Run your program again**. It should show how many countries are available in the database.
2. The next step is to define the Mountain and Peak entity classes:

1. Modify your EF database context to **register the new entity classes**:

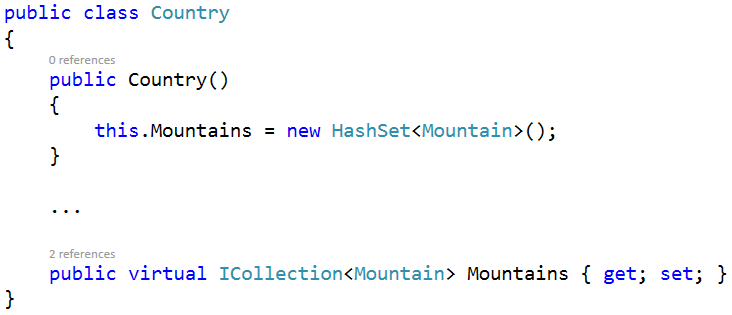


1. **Run your program** to test the modified EF data model. The database should be automatically dropped and re-created with tables for the new entity classes. Check the structure of the **tables** in the database:

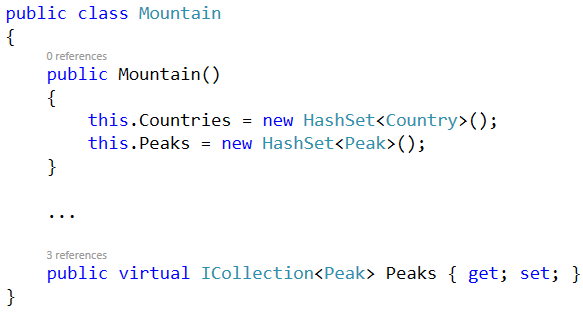


1. Now we have the Countries, Mountains and Peaks tables in the DB, but no relations are defined between them. Let's **define the relationships**:

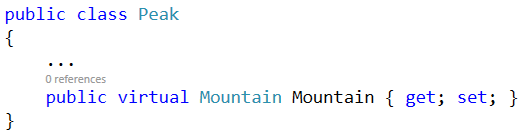
* Countries have many mountains (many-to-many):



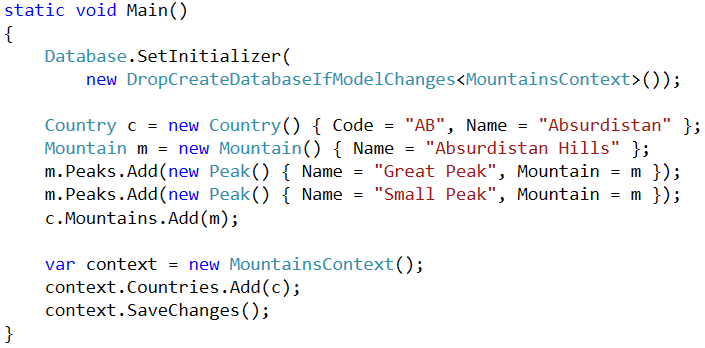
* Mountains have many peaks (one-to-many) and belong to many countries (many-to-many):



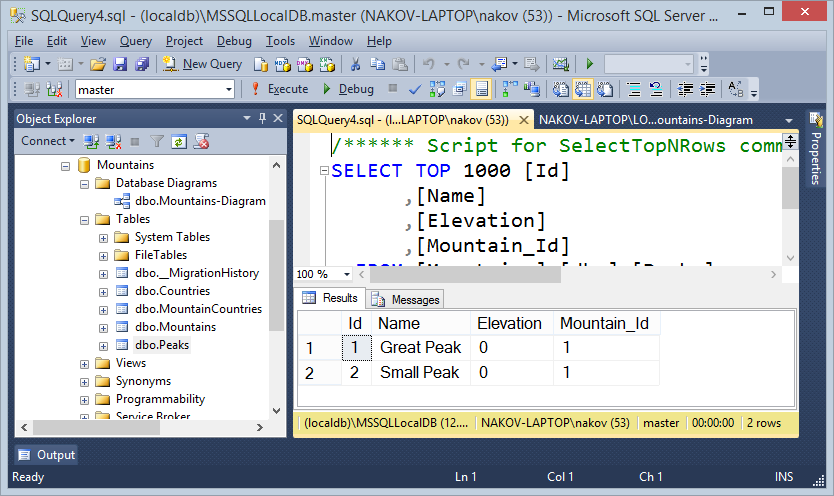
* Peaks belong to certain mountain (many-to-one):



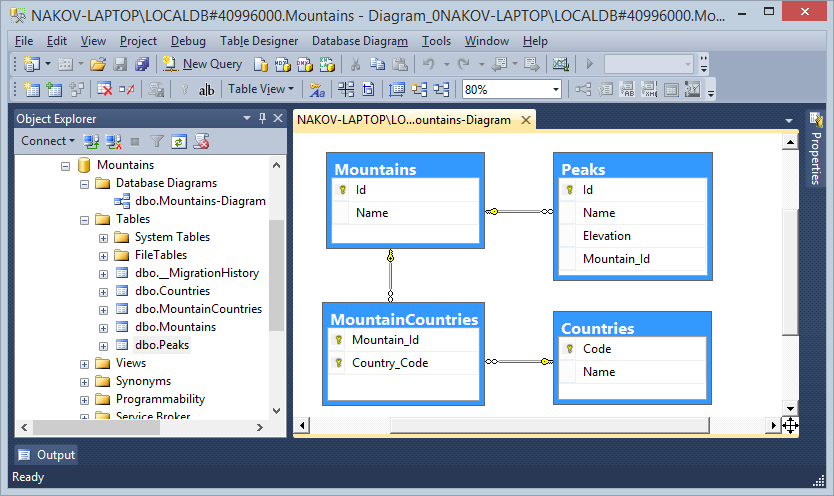
1. Now test your EF code first data model by **creating a few countries, mountains and peaks**:



1. Run the code. Your database should have a few countries, mountains and peaks:



1. To check whether the tables have correct relationships, create a database diagram in SQL Server Management Studio. It should hold the tables Countries, Mountains, Peaks and the many-to-many table MountainsCountries and correct relationships:



## Problem 5. EF Code First: Seed the Database

**Seed** your database with a few countries, mountains and peaks using the EF migrations framework. It is OK to drop the database in case of model changes or use any other migration strategy.

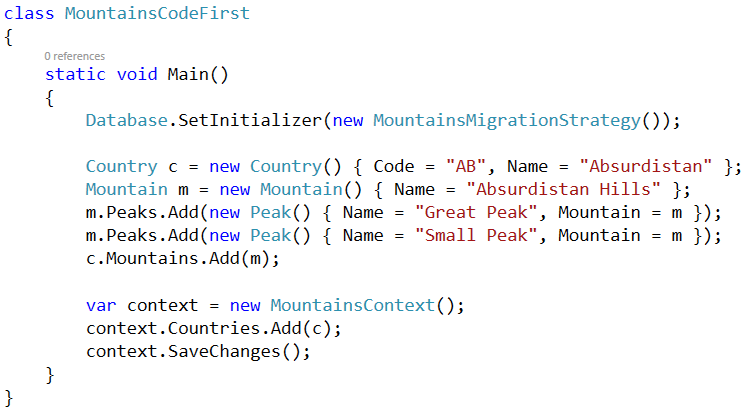
* **Countries**: *Bulgaria* and *Germany*.
* **Mountains**: *Rila*, *Pirin* and *Rhodopes*, all in country *Bulgaria*.
* **Peaks**: *Musala* (elevation *2925*, *Rila*), *Malyovitsa* (elevation *2729*, *Rila*) and *Vihren* (elevation *2914*, *Pirin*).

To test your data model, **list all mountains along with their countries and peaks**.

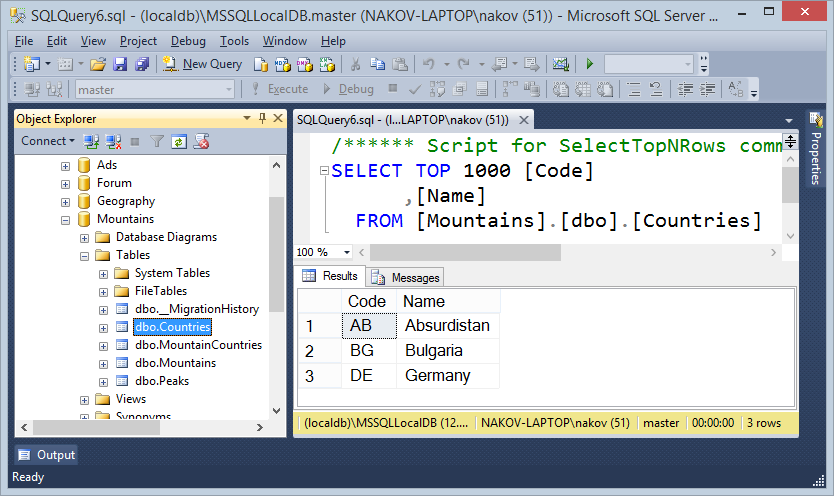
1. ***Seeding the database*** means to initially load some data after the database is created or modified. To seed the database, we could create our own database migration class by inheriting some of the existing EF migration classes (e.g. DropCreateDatabaseIfModelChanges) and override their Seed(…) method:



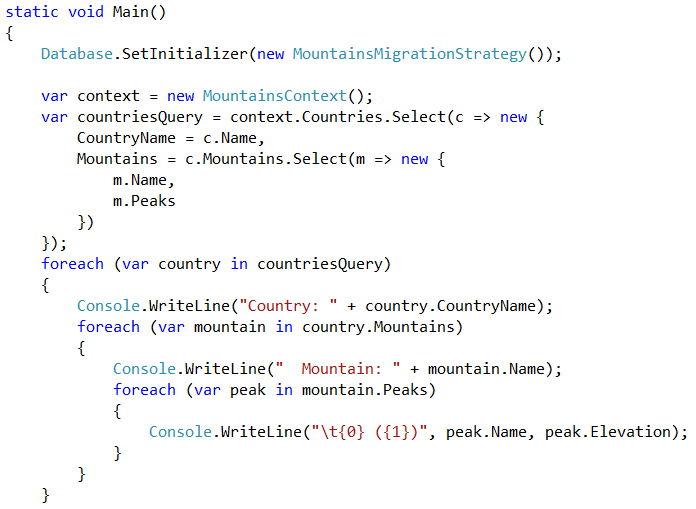
The Seed(…) method will be invoked after the database is dropped and re-created. You need to configure the application to use your new migration strategy class:



1. Now **drop the database** and run the program. It should invoke the Seed(…) method. Check the results in the database. You should have 3 countries ("Bulgaria" and "Germany" from the seed method and "Absurdistan" from the main program):



1. Now **list all mountains along with their countries and peaks** in order to test the seed:



1. Run the program to test whether it works correctly. The output should be like this:

