# Database Applications Exam (March 2015) – Geography

Your exam consists of several parts, explained below. You may work independently on each exam part. Submit your solutions as single **ZIP file holding the full source code**, without any libraries and compiled binaries.

## Part I – Query Existing Database, Import and Export Data

You are given a **MS SQL Server database "Geography"** holding continents, countries, currencies, monasteries and rivers, available as **SQL script**. Your task is to write a few data-driven applications in C# for importing data, querying data and exporting data from the database.

### Entity Framework Mappings (Database First)

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

static void Main(string[] args)

{

var context = new GeographyEntities();

foreach (var c in context.Continents)

{

Console.WriteLine(c.ContinentName);

}

}

5 score

### Export the 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.

8 score

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

3 score

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).

static void Main()

{

var context = new GeographyEntities();

var riversQuery = context.Rivers

.OrderByDescending(r => r.Length)

.Select(r => new

{

riverName = r.RiverName,

riverLength = r.Length,

countries = r.Countries

.OrderBy(c=> c.CountryName)

.Select(c => c.CountryName)

});

var json = new JavaScriptSerializer().Serialize(riversQuery.ToList());

File.WriteAllText(@"rivers.json", json);

}

4 score

### Export Monasteries by Country as XML

Write a **C# application** based on your EF data model for **exporting all monasteries by country** in 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.

8 score

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

3 score

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).

var context = new GeographyEntities();

var countriesQuery =

context.Countries

.Where(c=> c.Monasteries.Any())

.OrderBy(c => c.CountryName)

.Select(c => new

{

CountryName = c.CountryName,

Monasteries = c.Monasteries

.OrderBy(m => m.Name)

.Select(m => m.Name)

});

var xmlDoc = new XDocument();

var xmlRoot = new XElement("monasteries");

xmlDoc.Add(xmlRoot);

foreach (var country in countriesQuery)

{

var countryXml = new XElement("country",

new XAttribute("name", country.CountryName));

xmlRoot.Add(countryXml);

foreach (var monastery in country.Monasteries)

{

var monasteryXml = new XElement("monastery", monastery);

countryXml.Add(monasteryXml);

}

}

xmlDoc.Save("monasteries.xml");

4 score

### 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.

8 score

You should correctly **import the rivers into the DB**.

7 score

You should correctly import the **countries for each river** into the DB.

5 score

### \* Rivers by Country Query

Write a **C# application** based on your EF data model for **listing alphabetically all rivers that pass through a given set of countries**. The application should process multiple queries and produce an **XML output**. Each query might have an optional attribute max-results="*xxx*", that limits the results up to *xxx* items.

The input comes from an XML file rivers-query.xml in the following format:

|  |
| --- |
| **rivers-query.xml** |
| <?xml version="1.0" encoding="utf-8" ?>  <queries>  <query max-results="2">  <country>Brazil</country>  </query>  <query max-results="20">  <country>Mexico</country>  <country>Afganistan</country>  </query>  <query>  <country>Bulgaria</country>  <country>Romania</country>  <country>Austria</country>  </query>  </queries> |

The **output** should be printed on the console in the following **XML format** (without the XML prolog):

|  |
| --- |
| <results>  <rivers total-count="6" listed-count="2">  <river>Amazon</river>  <river>Madeira</river>  </rivers>  <rivers total-count="0" listed-count="0" />  <rivers total-count="1" listed-count="1">  <river>Danube</river>  </rivers>  </results> |

The query results should come in the same order like the queries in the input. Display in the output the **total count** of matched rivers and the **listed count**: the smaller of total count and max results.

20 score (bonus)

## Part II – EF Code First: Define Data Model, Import and Export Data

You are assigned to define a **code first data model in EF** and write a few data-driven applications in C# for importing data, querying data and exporting data from the database.

Use a new database "**Mountains**". Do not modify the "**Geography**" database.

### 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**.

public class Country

{

public Country()

{

this.Mountains = new HashSet<Mountain>();

}

[Key]

[StringLength(2)]

public string Code { get; set; }

[Required]

public string Name { get; set; }

public virtual ICollection<Mountain> Mountains { get; set; }

}

* **Mountains** have a **name** and belong to **multiple countries**.

public class Mountain

{

public Mountain()

{

this.Countries = new HashSet<Country>();

this.Peaks = new HashSet<Peak>();

}

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public virtual ICollection<Peak> Peaks { get; set; }

public virtual ICollection<Country> Countries { get; set; }

}

* **Peaks** have a **name**, **elevation** and **mountain**.

public class Peak

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public int Elevation { get; set; }

public int MountainId { get; set; }

public virtual Mountain Mountain { get; set; }

}

7 score

**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*).

public class MountainsContext : DbContext

{

public MountainsContext() : base("name=MountainsContext")

{

}

public virtual DbSet<Country> Countries { get; set; }

public virtual DbSet<Mountain> Mountains { get; set; }

public virtual DbSet<Peak> Peaks { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

Database.SetInitializer(new MountainsMigration());

}

}

internal sealed class MountainsMigration : DropCreateDatabaseIfModelChanges<MountainsContext>

{

protected override void Seed(MountainsContext context)

{

// Seed data if not already seeded

var bulgaria = new Country { Code = "BG", Name = "Bulgaria" };

context.Countries.Add(bulgaria);

var germany = new Country { Code = "DE", Name = "Germany" };

context.Countries.Add(germany);

var rila = new Mountain { Name = "Rila", Countries = { bulgaria } };

context.Mountains.Add(rila);

var pirin = new Mountain { Name = "Pirin", Countries = { bulgaria } };

context.Mountains.Add(pirin);

var rhodopes = new Mountain { Name = "Rhodopes", Countries = { bulgaria } };

context.Mountains.Add(rhodopes);

var musala = new Peak { Name = "Musala", Elevation = 2925, Mountain = rila };

context.Peaks.Add(musala);

var malyovitsa = new Peak { Name = "Malyovitsa", Elevation = 2729, Mountain = rila };

context.Peaks.Add(malyovitsa);

var vihren = new Peak { Name = "Vihren", Elevation = 2914, Mountain = pirin };

context.Peaks.Add(vihren);

}

}

5 score

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

static void Main()

{

var context = new MountainsContext();

var mountains = context.Mountains.Select( m => new

{

m.Name,

Countries = m.Countries.Select(c => c.Name),

Peaks = m.Peaks.Select(p => new { p.Name, p.Elevation })

});

foreach (var mountain in mountains)

{

Console.WriteLine(

"{0}, countries: {1}, peaks: {2}",

mountain.Name,

string.Join(", ", mountain.Countries),

string.Join(", ", mountain.Peaks.Select(

p => p.Name + " (elevation " + p.Elevation + ")")));

}

}

3 score

### Import Mountains, Peaks and Countries from JSON

Write a **C# application** based on your EF code first data model for **importing into the DB a set of mountains and peaks** given in the JSON file mountains.json. The mountains and peaks come in the following JSON format:

|  |
| --- |
| **mountains.json** |
| [  { "mountainName" : "Andes",  "peaks" : [  { "peakName" : "Aconcagua", "elevation" : 6962 },  { "peakName" : "Monte Pissis", "elevation" : 6793 },  { "peakName" : "Ojos del Salado", "elevation" : 6893 } ],  "countries" : ["Argentina", "Chile"] },  { "mountainName" : "Unknown Mountains" },  { },  { "peaks" : [ { "peakName" : "Incorrect peak" } ],  "mountainName" : "Strange Mountain" },  { "mountainName" : "Caucasus",  "peaks" : [  { "peakName" : "Dykh-Tau", "elevation" : 5205 },  { "peakName" : "Elbrus", "elevation" : 5642 } ],  "countries" : ["Russia", "Georgia"] }  ] |

The properties mountainName, peakName and elevation are **mandatory**. The properties peaks and countries are **optional**.

You should **parse the JSON** 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 JSON file will be less than **10 MB**. Use a JSON parser by choice.

10 score

You should correctly **import the mountains into the DB**.

3 score

You should correctly import the **countries for each mountain** into the DB.

3 score

**Create all missing countries** on demand. As **country code** use the first 2 letters of the country name (in uppercase).

5 score

You should correctly import the **peaks for each mountain** into the DB.

4 score

Insert each mountain in a **separate transaction**. A mountain should either be inserted correctly along with all its peaks and countries, or no part of it should be inserted at all. Print as **output** a single line for each mountain from the input JSON: either "*Mountain <name> imported*" or "*Error: <error message>*". Error messages should describe briefly the problem (as free text) and may optionally include exception stack-trace. Sample output:

|  |
| --- |
| Mountain Andes imported  Mountain Unknown Mountains imported  Error: Missing mountain name  Error: Missing peak elevation  Mountain Caucasus imported |

5 score

## Exam Information

You are allowed to use any resources you have, e.g. Internet, software, existing code.

You are not allowed to get help from other people. Skype, ICQ, FB, email, talks, phone calls, etc. are forbidden.

Exam time: **6 hours**.