# Database Frameworks Midterm Exam – Mass Defect

# Entity Framework

The year is 2306. The united galactic civilizations (UGC), are steadily progressing through the process of evolution. Almost a century ago, an unexpected phenomenon was introduced to the young human civilization and to other alien civilizations. People from the different civilizations were being randomly teleported across the galaxy, without any pattern or logic behind. The situation was too awkward to be explained, so the civilizations decided to form a unity to deal with this phenomenon. That is how the UGC was formed. The UGC established a Database which would keep track of the random teleports and their victims. The human civilization describes these anomalies as the worst discovery in their history. The civilizations of the galaxy call it… Mass Defect.

# Data Model Definition

You have been tasked to create a **code first data model** in **Entity Framework** for the Mass Defect Database. You will also need to write several data-driven applications in C# for importing, querying and exporting data from the database. For some reason the UGC has decided that **JSON** and **XML** are the default data formats, so your imports and exports will be performed with those formats.

The database you need to implement has 5 main entities:

##### Solar Systems

* Have **Id** and **Name**.

##### Stars

* Have **Id** and **Name**.
* The Stars **must** have a **Solar System**.

##### Planets

* Have **Id** and **Name**.
* The Planets **must** have a **Sun**.
* The Planets **must** have a **Solar System**.

##### Persons

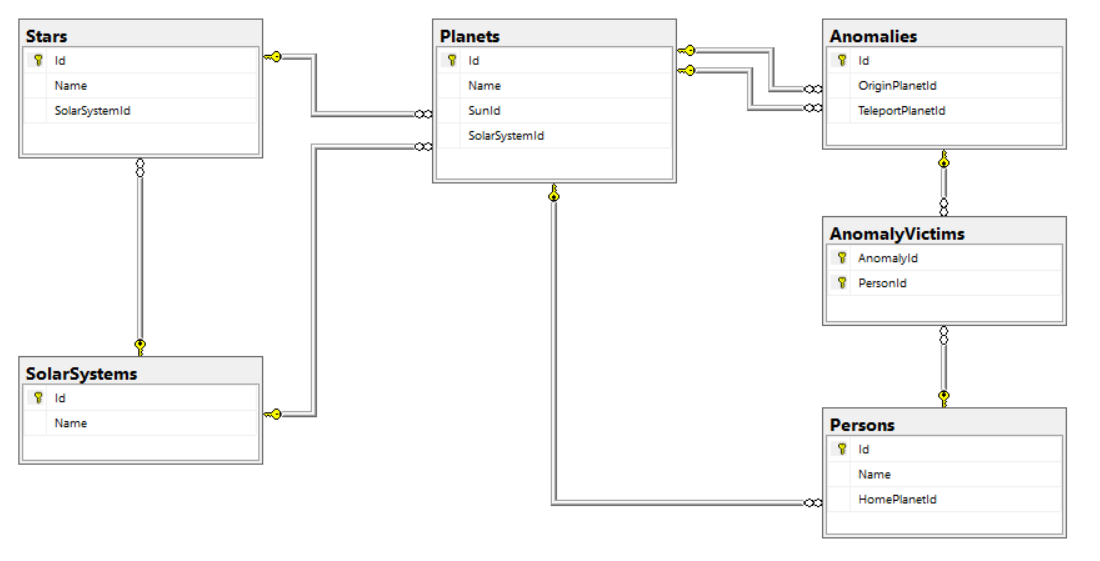
* Have **Id** and **Name**.
* The Persons **must** have a **Home** **Planet**.

##### Anomalies

* Has **Id**.
* The Anomalies **must** have an **Origin Planet** and **Teleport Planet**.
* The Anomalies can have **many Persons** as victims, and one **Person** can be a victim to **many** **Anomalies**.

As you see the only field that isn’t shared between any two entities is the **Name**. So you will have to explicitly make sure that that field is **required**.

As you see some of the entities have relations between themselves. You’ll need to configure them correctly. Here is a E/R diagram of the database to make it easier for you:



A specialist from the UGC has decided to help you a little because you are still new. So be prepared, even with his help it won’t be easy.

Create a **C# application** to implement the **data model** for the database exactly as specified above.

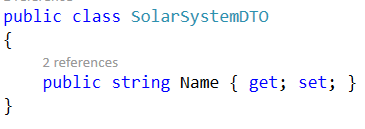
Let’s start. Create your own Database Context Class. You will also need to override the Model Builder. Use the Fluent API to do that. Make it so that the auto-generated table for the many-to-many relationship is named “AnomalyVictims” and the two columns in it are named “AnomalyId” and “PersonId”.

# Importing Data

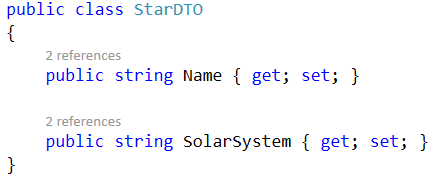
Now that you have successfully created your database, you will need to import the data that has been recorded on paper all this time because the UGC didn’t have a database developer such as yourself. You will need to import some dependencies yourself. Make sure you have the **Newtonsoft JSON** **package**.

### Importing Data from JSON

Create **DTOs** based on your **data model** and the **input data format**, so that you could easily seed the data into the database. Let’s make a DTO for **the Solar System Entity**. In the input we have only a name given, because the Id is auto-generated. So we need just one thing and that is the Solar System’s **name**:



Okay, but that was too easy. Now we need to create a little more complex DTO – for the Star. The Star also has a Solar System, but from the input, we see that it is given as a **name** of a **Solar System**, so we’ll have to extract the Solar System from the database by its name, if it exists that is. Anyways, here is how the Star DTO should look like:



The other **DTOs** are for you to implement. Follow the strategy we used above to implement them.

Create a new **C# application** based on your **data model** and seed the data from the **JSON** files you received, into the database.

Make sure all fields have been entered, otherwise the import **entity** data **should not be considered valid**.

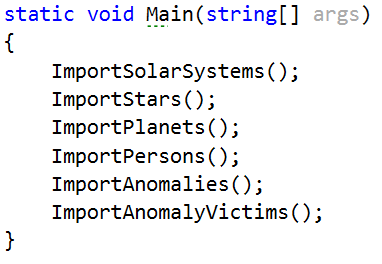
If you import correctly an entity of **Solar System**, **Star**, **Planet** or **Person**, you should print a message on the console, saying: “Successfully imported {entity} {entityName}.”.

If you successfully import data about **Anomalies**, you should print a message on the console saying: “Successfully imported anomaly.”.

Successful imports in the anomaly-victims, should not hold notification.

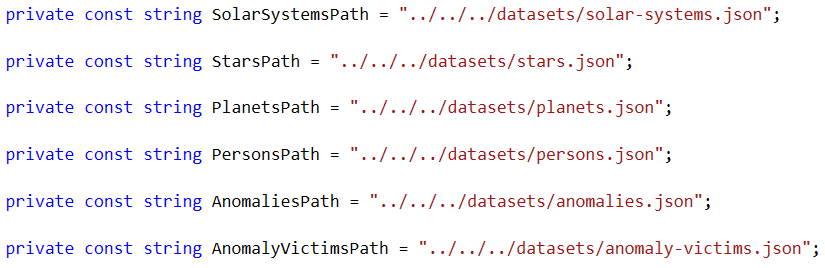
In case one of the fields is missing In the import data, print a message:  
“Error: Invalid data.”… and ignore that **particular** **entity**.

In your newly created C# project for the **JSON imports**, in the Main method, put the following methods:



Implement those methods, so that they are ready for use. We will create the logic behind them, right away.

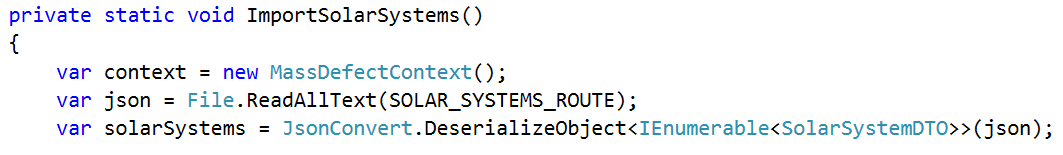
Now, make sure you have the **datasets folder**, in your **solution’s folder**. Let’s build constants for the paths to the **JSON files**.



Now that we have all the preparations, we can start building the import methods one by one.

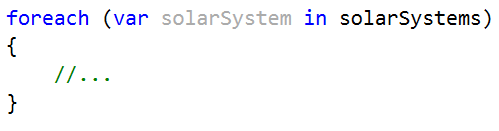
#### Importing Solar Systems

Let’s start with the importing of the Solar Systems. First we need to initialize all core data, such as – initialize the context, load and deserialize the JSON file:

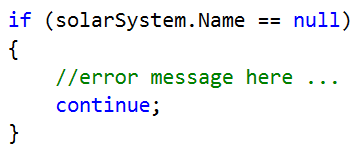


As you see we simply initialize the **context**, we load the JSON file with the help of **System.IO** library, and we use the **Newtonsoft JSON** to deserialize the data from the file and turn it into a **collection** of items – SolarSystemDTO.

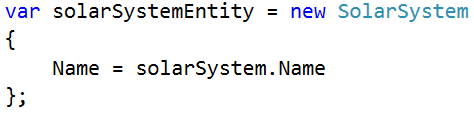
Now that we have our data extracted and parsed, we can start traversing, validating and ultimately responding to it.



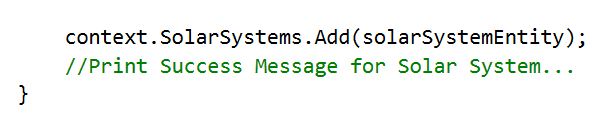
Let us traverse the data with a simple foreach loop. The first thing we need to do is check if our DTO has acquired the data correctly. If not, we should print an error message and ignore that input:



If there is no error, we should create an Entity with the data extracted from the DTO:



Now that we have even our entity **declared** and **initialized**, we can add it to the **context’s collections**:



After the main loop that traverses the whole data, there is only one thing left to do in this method – save the changes on the context:



That should be all for the Solar systems. Test your functionality before continuing with the next method.

### Example:

#### Input

|  |
| --- |
| **solar-systems.json** |
| [  { "name" : "Kepler-Epsilon" },  { "name" : "Alpha-Nebula" },  { "name" : "Beta-Cluster" },  { "name" : "Voyager-Sentry" },  { "name" : "Zeta-Cluster" },  ] |

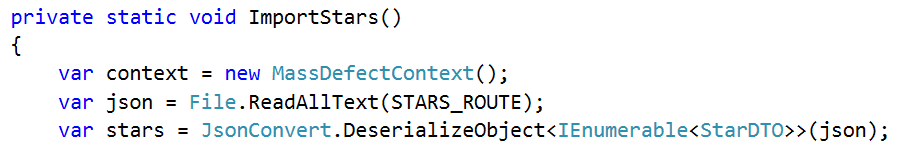
#### Output

|  |
| --- |
| Successfully imported Solar System Kepler-Epsilon.  Successfully imported Solar System Alpha-Nebula.  Successfully imported Solar System Beta-Cluster.  Successfully imported Solar System Voyager-Sentry.  Successfully imported Solar System Zeta-Cluster. |

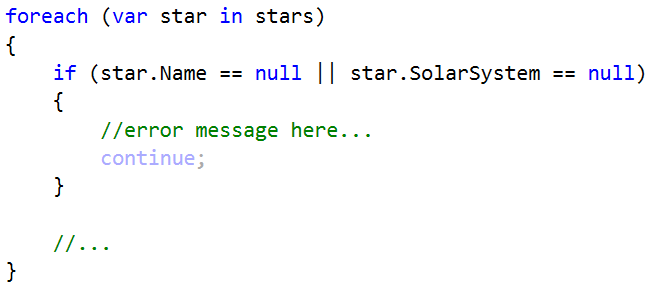
The Stars a bit more interesting, because they require a Solar System, and all they get is a name.

#### Importing Stars

Same as the Solar Systems, we initialize all core data:

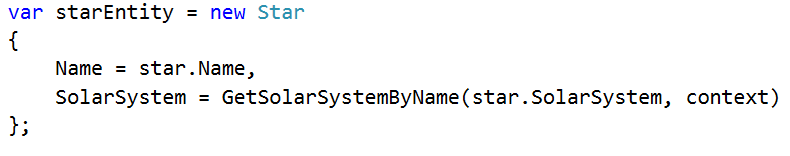


With slight differences, but it is the same as the last method…



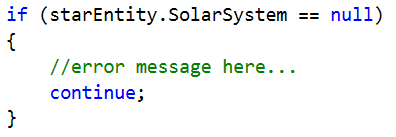
This time we have an additional parameter with our input, which is the Solar System’s name, so we have to validate that it exists too.

And now comes the real part – the entity initialization.

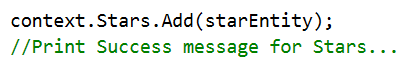


We have the name but we do not own the Solar System as an Entity… We have to extract it from the already made Solar System Collection. That is why we make a method GetSolarSystemByName(). We pass to the method the DTO’s Solar System and the context. The method should extract from the database, the Solar System Entity, corresponding to the given one.

Now, we traverse through all the Solar Systems, currently in the Solar System collection, and if any of them has a **name** **equal** to **the passed one**, we **return it**, otherwise we just return **null**. This method will either return us a Solar System entity, or **null**. And after we have finished the initialization of the **Star entity**, we have to perform **another check** to validate that the **Solar System** we’ve just extracted is not, in fact, **null**.



When even that check passes, we can safely import the newly created Star entity to the **collection** of **Stars**:



Of course, as soon as we get out of the loop, we **save** the **changes** on the **context**, and we end the method.

Test your functionality before continuing with the next method.

### Example:

#### Input

|  |
| --- |
| **stars.json** |
| [  { "name" : "Visilus", "solarSystem" : "Kepler-Epsilon" },  { "name" : "Neb-X10", "solarSystem" : "Alpha-Nebula" },  { "name" : "Scarlet-Sentry" },  { "name" : "Indigo-Sentry", "solarSystem" : "Voyager-Sentry" },  { "name" : "Neb-X11", "solarSystem" : "Alpha-Nebula" },  ] |

#### Output

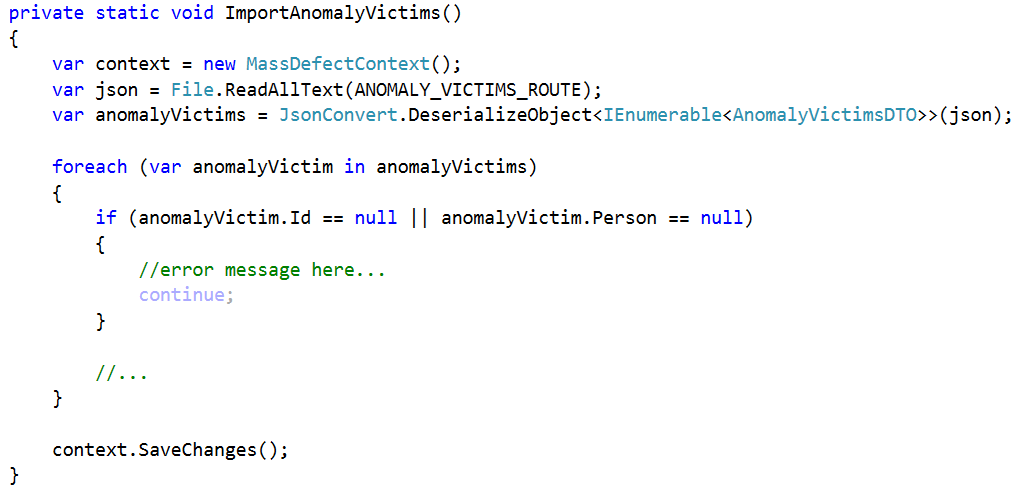
|  |
| --- |
| Successfully imported Star Visilus.  Successfully imported Star Neb-X10.  Error: Invalid data.  Successfully imported Star Indigo-Sentry.  Successfully imported Star Neb-X11. |

As you’ve seen the import methods follow a similar pattern, which is why the UGC specialist has decided you could do pretty well on your own.

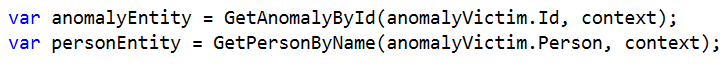
Following the implementation logic, you’ve seen by now, implement the methods for the importing of **Planets**, **Persons** and **Anomalies**.

#### Importing Anomaly Victims

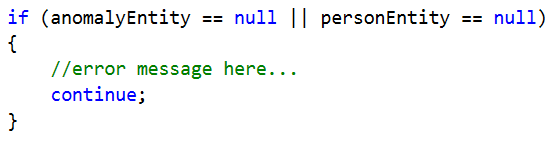
Now this importing functionality is a bit more different, that is why we the agent has decided to assist you here.



Everything so far is the same as the previous methods. The real difference comes in the entities… Instead of adding to the context, we will extract an **anomaly** by **its Id**, and a **person** by **his name**, and add the person to the **collection** of **Victims** the **anomaly** holds.



The GetAnomalyById() method might extract in a bit different way than the others, but it’s on the same principle.



Only if both entities are existent, we can proceed. The last part is **the adding** of the **person entity** to the **victim collection** of the **anomaly entity**:



With that, we finish the loop, we save the context changes, and we finish the JSON imports.

### Importing Data from XML

New reports have come about several new anomalies. The format of the reports, however, this time is **XML**. Nevertheless, you need to put them in the database.

Create a new **C# application** based on your **data model** and import the data from the “new-anomalies.xml” file in the database.

The UGC specialist will assist you on this one. In your newly created C# project, in the Main class, we need to create a constant for the **route** to our **xml file**.

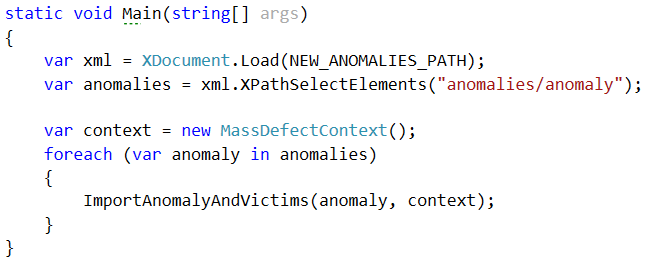


Again … make sure the **datasets folder** is in **your solution’s folder**.

You might need this in your code:

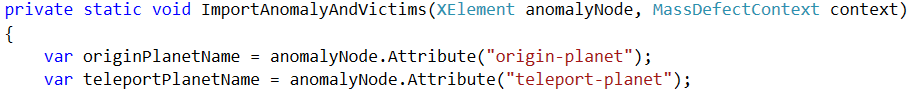


Next, we need to write the main logic in the Main method:

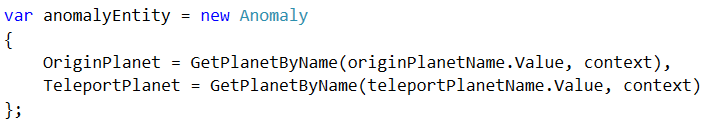


As you see we read the xml file at the given path, and we store it. Then we extract the anomalies as a collection, so that we can traverse it. We initialize our context, and we traverse the collection of anomalies, importing each of them.

So let’s begin implementing the method for importing anomalies:

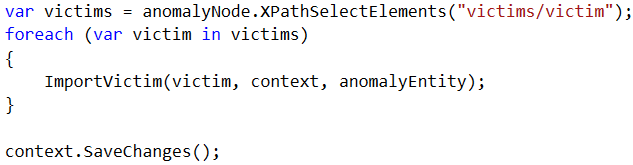


We extract the two attributes, but they are still only attributes, we need to extract their values. Before you do that, however, make sure they are **existent**.

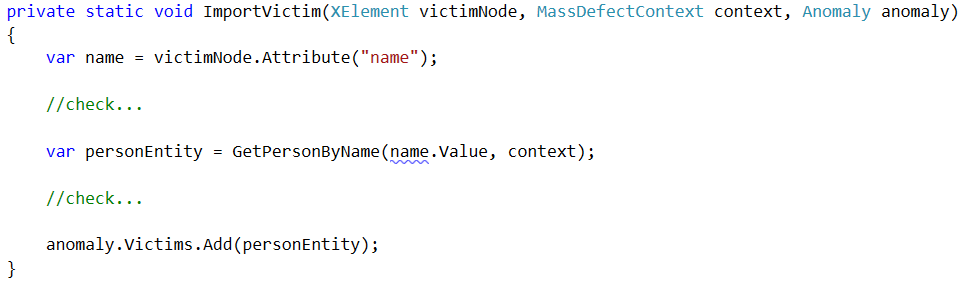


And here we are actually extracting the values from the attributes. As you see the anomaly entity initialization is nearly the same as with the JSON. Make sure you implement the GetPlanetByName() method correctly.

Now that we have our entity, a validation will be needed to ensure the planets were valid. After that we can proceed to adding the anomaly to the anomalies collection, and printing a success message. However, there is one more thing before we save the context changes and leave the method. The anomaly isn’t the only thing that needs to be inserted in here.



Now we have an importing logic for the victims too, and we need to implement it:



Implement the checks, so that there is **no invalid input data**.

And we that we finish the XML importing and the Importing logic as a whole. Test your code before moving to the next section.

### Example:

#### Input

|  |
| --- |
| **new-anomalies.xml** |
| <?xml version="1.0" encoding="utf-8"?>  <anomalies>  <anomaly origin-planet="Kepler-3" teleport-planet="Voyager-10">  <victims>  <victim name="Eifell Sync" />  </victims>  </anomaly>  <anomaly origin-planet="Kepler-1">  <victims>  <victim name="Eifell Sync" />  </victims>  </anomaly>  <anomaly origin-planet="Voyager-10" teleport-planet="Voyager-11">  <victims>  <victim name="Eifell Sync" />  </victims>  </anomaly>  ...  </anomalies> |

#### Output

|  |
| --- |
| Successfully imported anomaly.  Error: Invalid data.  Successfully imported anomaly.  ... |

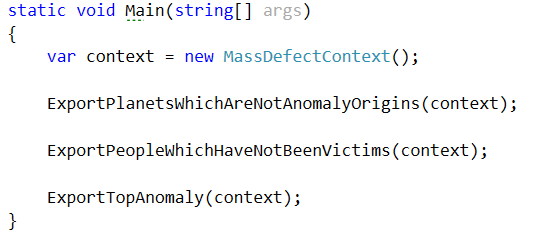
If any field data is **missing**, you should print an **error message**, and **ignore** that **input data**.

# Data Exporting

The UGC has requested applications for **exporting** **data** from the database, so that statistics can be made and presented publically. You know the drill, you will have to build applications for exporting data in both **JSON** and **XML** formats. People need to be informed of the events that are happening around the Galaxy. There are several **query** **tasks** you need to do. Create a C# application for those tasks.

So let’s start, the UGC specialist does not have much time left to look over you, he’s got important work to do.

In your newly created C# project, in the Main method, put the following:



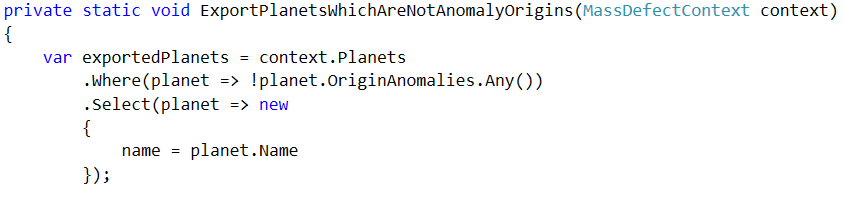
Those are the methods that will do the querying over our database. You’ll need to implement them good, so that results are perfect. The data should be exported into **JSON files**, so you will need to concentrate on this task and give it your all.

Here are the several tasks you need to do.

### Planets which have no people teleported FROM them

Extract all planets from the database, which are not an **origin planet** to any Mass Defect anomaly. Extract the planets’ **names**.

The UGC specialist has decided to help you with this one, but for the next two, you are on your own.



Here, now we have taken from the collection of Planets, and we have filtered the planets that have not been an Origin planet to any Anomaly. After the filtering, we have **mapped** the **received Entities** to **custom objects**, which have a single property – name. Nothing more is necessary for the JSON output.



The real deal is here, make sure you do this correctly, otherwise it will not work. Run your code to check your results:

|  |
| --- |
| **planets.json** |
| [  {  "name": "Alpha-N45"  }  ] |

### People which have not been victims of anomalies

Extract all persons from the database, which have **not been a victim** of a Mass Defect anomaly. Extract the persons’ **names** and **home planets’ names**.

|  |
| --- |
| **people.json** |
| [  {  "name": "Asylus Ovelox",  "homePlanet": {  "name": "Kepler-1"  }  },  {  "name": "Seren Joseph",  "homePlanet": {  "name": "Alpha-N20"  }  },  {  "name": "Isdislav Irenovic",  "homePlanet": {  "name": "Voyager-11"  }  },  {  "name": "Aina",  "homePlanet": {  "name": "Alpha-N45"  }  },  {  "name": "Nero",  "homePlanet": {  "name": "Kepler-6"  }  }  ] |

### Anomaly which affected the most people

Extract the anomaly which has affected the most victims. Extract the anomaly’s **id**, **origin planet name**, **teleport planet name**, and **number of victims**.

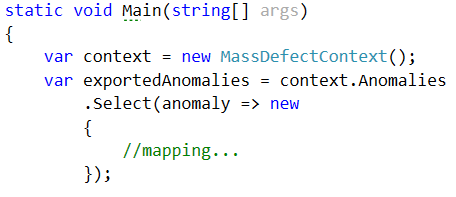
|  |
| --- |
| **anomaly.json** |
| [  {  "id": 14,  "originPlanet": {  "name": "Kepler-1"  },  "teleportPlanet": {  "name": "Alpha-N20"  },  "victimsCount": 5  }  ] |

# Exporting to XML

You also need to export data about the anomalies in **XML** format, for the **3-rd Galaxy Solar Systems**, which still haven’t evaluated enough to read **JSON** format.

Extract **all anomalies**, and the **people affected by them**. Extract the **anomalies**’ **origin planets’ names** and **teleport planets’ names**. For the **persons**, extract only their **names**.

This is the last task the UGC specialist is going to help you for. In your newly created C# project, in the main method:



Extract the anomalies from the Anomaly collection, and map them to custom objects, which have id, originPlanetName, teleportPlanetName, and victims, which is a **list of strings**. Order the results you’ve extracted by id, in **ascending order**.

Next, we need to create an XML element for the anomalies:



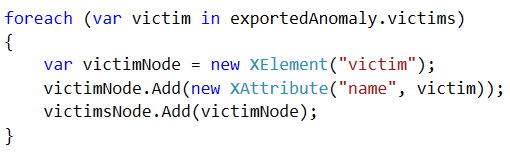
Now that we have that, we can start traversing and exporting the data and entities we’ve retrieved, one by one:



This will create a XML node for the anomalies and add XML attributes to it, with the values from the retrieved database data. But anomalies aren’t the only thing we need to extract here.



… We also need their victims. Traverse through the exported anomaly’s victims, create a special XML node for them, set It’s attribute a value from the database and add each of them to the **victims’ node**:



Before you end the main loop, add the **victims’ node** to the **anomaly node** and the **anomaly node** to the xmlDocument. Then, save the document and check what you have done



|  |
| --- |
| **anomalies.xml** |
| **<?xml version="1.0" encoding="utf-8"?>**  **<anomalies>**  **<anomaly id="1" origin-planet="Voyager-10" teleport-planet="Kepler-1">**  **<victims>**  **<victim name="Baron Newhousen" />**  **<victim name="Vox Populi" />**  **<victim name="Antra Foul" />**  **</victims>**  **</anomaly>**  **<anomaly id="2" origin-planet="Alpha-N20" teleport-planet="Kepler-3">**  **<victims>**  **<victim name="Saine" />**  **</victims>**  **</anomaly>**  **<anomaly id="3" origin-planet="Kepler-6" teleport-planet="Kepler-1">**  **<victims>**  **<victim name="Pokolri Paputo" />**  **</victims>**  **</anomaly>**  **...**  **<anomaly id="20" origin-planet="Voyager-10" teleport-planet="Voyager-11">**  **<victims>**  **<victim name="Moria Bane" />**  **<victim name="Saine" />**  **</victims>**  **</anomaly>**  **</anomalies>** |