Database Frameworks Midterm Exam – Mass Defect

Spring Data

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 **Spring Data** for the Mass Defect Database. You will also need to write several data-driven applications in Java 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

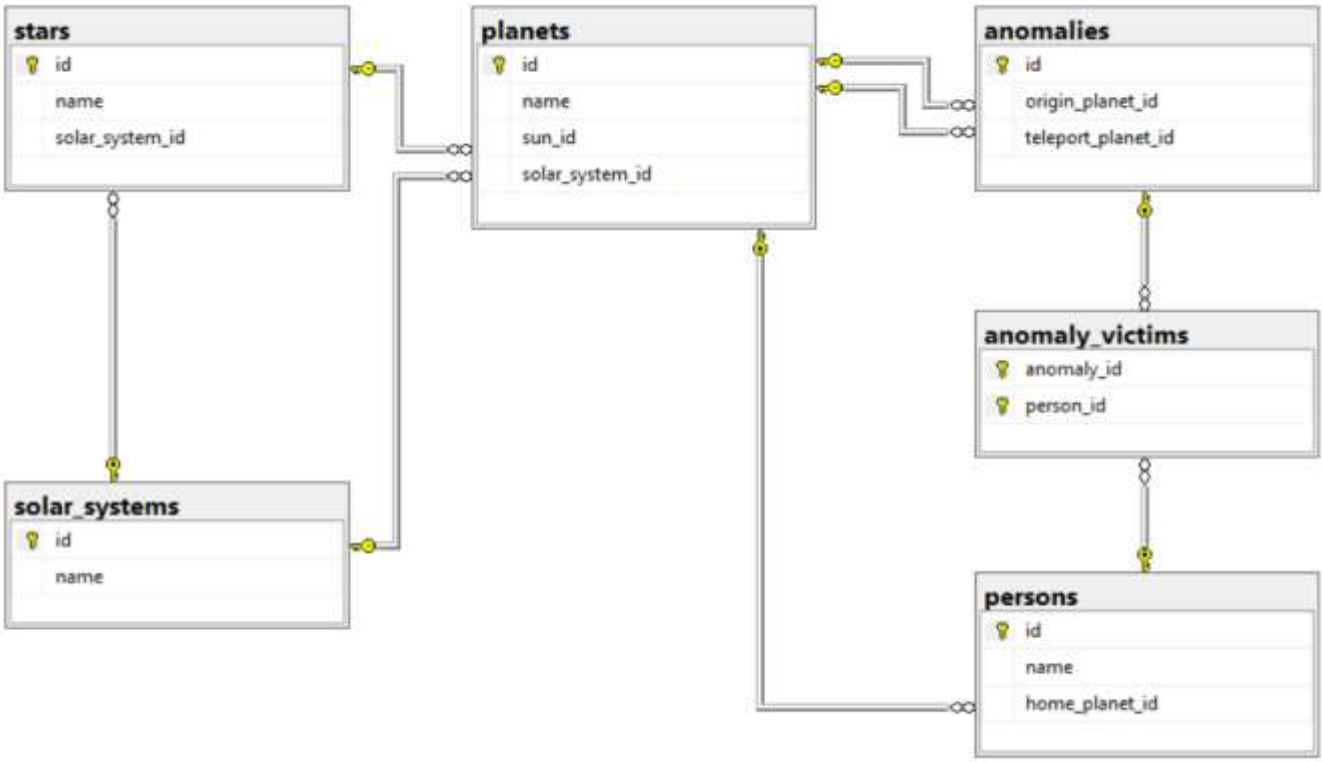
- Have **Id** and **Name**.
- The Persons **must** have a **Home Planet**.

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 **Spring application** to implement the **data model** for the database exactly as specified above.

Make it so that the auto-generated table for the many-to-many relationship is named "**`anomaly_victims`**" and the two columns in it are named "**`anomaly_id`**" and "**`person_id`**".

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.

`NOTE:` Create the appropriate **`Services`** and **`Repositories`** for your entities. The **Services'** main job is to assure the **validity of the data** they are going to **persist**.

Structure your code so that it is well prepared and **READABLE**. Create a terminal and put all functionality there. You are strongly advised to keep the order of the functionality – the same as the assignment.

Importing Data from JSON

Well, certainly, you will need something to read and parse JSON with, and since you are pretty new, the UGC agent has decided to help you out.

`NOTE:` You'll need **`gson`** for this part. Make sure you have it before you start.

Let's start with the implementation of a simple JSON Parser:

```
public class JSONParserImpl {

    private Gson gson;

    public JSONParserImpl() {
        this.setGson(new GsonBuilder().setPrettyPrinting().create());
    }

    private Gson getGson() {
        return this.gson;
    }

    private void setGson(Gson gson) {
        this.gson = gson;
    }
}
```

We have initialized the main data of our JSON parser, now all we need is to write the main functionality:

```
public <T> T[] readFromJSON(Class<T[]> classes, String file) {
    String fileData = null;
    T[] objects = null;
    //InputStream inputStream = getClass().getResourceAsStream("/" + file);
    try {
        fileData = new String(Files.readAllBytes(Paths
                .get(file)));
        objects = this.getGson().fromJson(fileData, classes);
    } catch (IOException e) {
        e.printStackTrace();
    }

    return objects;
}
```

And with this, we have now a fully implemented JSON **reader**. We can read and parse JSON data. Now we can import our data correctly. The JSON reader accepts in its method – the **class** of **an array** of the **DTOs** of the **corresponding Entity**, so be careful what parameters you give. When you parse the data send it to the **service**.

Create your **DTOs** based on your **data model** and seed the data from the **JSON** files you received, 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**:

```
import java.io.Serializable;

public class SolarSystemImportDto implements Serializable {

    private String name;

    public SolarSystemImportDto() {
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) { this.name = name; }
}
```

Of course, it needs to implement the **`Serializable`** interface. Now that you've done that you are almost ready to continue forward.

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:

```
import java.io.Serializable;

public class StarImportDto implements Serializable {

    private String name;

    private String solarSystem;

    public String getName() { return this.name; }

    public void setName(String name) { this.name = name; }

    public String getSolarSystem() { return this.solarSystem; }

    public void setSolarSystem(String solarSystem) { this.solarSystem = solarSystem; }
}
```

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

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.

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.
```

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

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.
```

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 new **DTOs** based on your **data model** and import the data from the "**`new-anomalies.xml`**" file in the database. We had some DTOs for the JSON files, but we'll need different ones for the XML. But don't worry, the UGC agent has decided to give us a little hint.

```
import javax.xml.bind.annotation.*;
import java.io.Serializable;
import java.util.List;

@XmlRootElement(name = "anomalies")
@XmlAccessorType(XmlAccessType.FIELD)
public class AnomaliesImportDto implements Serializable {

    @XmlElement(name = "anomaly")
    private List<AnomalyWithVictimImportDto> anomalies;

    public List<AnomalyWithVictimImportDto> getAnomalies() { return this.anomalies; }

    public void setAnomalies(List<AnomalyWithVictimImportDto> anomalies) { this.anomalies = anomalies; }
}
```

As you see, this is the **`AnomaliesDTO`** for XML. Make sure you remember the imports and the annotations that are used here, you'll have to use them later. There is a also a part of another class:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "anomaly")
public class AnomalyWithVictimImportDto implements Serializable {

    @XmlAttribute(name = "origin-planet", required = true)
    private String originPlanet;

    @XmlAttribute(name = "teleport-planet", required = true)
    private String teleportPlanet;

    @XmlElementWrapper(name = "victims")
    @XmlElement(name = "victim")
    private List<VictimImportDto> victims;

    public AnomalyWithVictimImportDto() {
    }
```

But the agent has given the task of implementing the properties, this time, to you. So you'll do it yourself. Make sure you make a **`VictimDTO`** too, when you finish, you'll need it.

Aside from the DTOs, you might need a XML Parser too, like the JSON one. Well luckily, the UGC agent has decided to help us with this one too. The **JAXB** will be used for this part so make sure you pay attention:

```java
public class XMLParserImpl implements XMLParser {

    @Override
    public <T> T readFromXml(Class<T> classes, String fileName) throws JAXBException, SAXException {
        JAXBContext jaxbContext = JAXBContext.newInstance(classes);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
        File file = new File(fileName);
        T objects = (T) unmarshaller.unmarshal(file);
        return objects;
    }
}
```

This will help us read XML files. Unlike the JSON parser this time the method receives as an argument a **class**. Read the data from the XML and pass it to the **service**.

The given input and expected output is as follows…

Example:

Input

new-anomalies.xml

```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 data.
Error: Invalid data.
Successfully imported data.
...
```

If any field data is **missing**, you should print an **error message**, and **ignore** that **input data**. If a victim's data is **invalid**, not only the victim, but **the whole anomaly input** should be **ignored**.

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 spring data application for those tasks.

For the data exporting, you'll need explicit DTOs, so it is recommended that you build ones for the entities that are **exported**. The DTOs are simple – they just have to hold the **fields** and **properties**, that are **required** for the **output**.

You will have to extend the parsers though. Add one more method to them, so that they can write files too:

JSON Parser

```
public <T> void writeToJSON(T object, String file) {

    String json = this.getGson().toJson(object);
    BufferedWriter bfw = null;
    try {
        bfw = new BufferedWriter(new FileWriter(file));
        bfw.write(json);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (bfw != null) {
                bfw.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This is the method for reading JSON data from a file. Make sure you implement it correctly. Upon exporting the data with the **service**, use the **parser**, to write the **JSON**. Do the same with all the problems from this section.

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**.

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.

Last but not least, comes the hardest part – the XML export. Oh well let us pray it is easy.

First we need to extend the parser:

```
public <T> void writeToXml(T object, String fileName) throws JAXBException {
    JAXBContext jaxbContext = JAXBContext.newInstance(object.getClass());
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
    File file = new File(fileName);
    jaxbMarshaller.marshal(object, file);
}
```

This should be more than enough…

The next thing we need to do is implement the XML Export DTOs:

```
public class AnomaliesExportXMLDto {

    @XmlElement(name = "anomaly")
    private List<AnomalyExportXMLDto> anomalyExportDtos;

    public AnomaliesExportXMLDto() { this.setAnomalyExportDtos(new ArrayList<>()); }

    public List<AnomalyExportXMLDto> getAnomalyExportDtos() { return this.anomalyExportDtos; }

    public void setAnomalyExportDtos(List<AnomalyExportXMLDto> anomalyExportDtos) {
        this.anomalyExportDtos = anomalyExportDtos;
    }
}
```

The UGC agent has given us pretty much any information so far, and he is getting tired, maybe we should not ask for any more help from him. The DTOs follow almost the same logic anyways. Use the **service** to **export the data**, and the parser to **write it to the file**.

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**.

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>
```