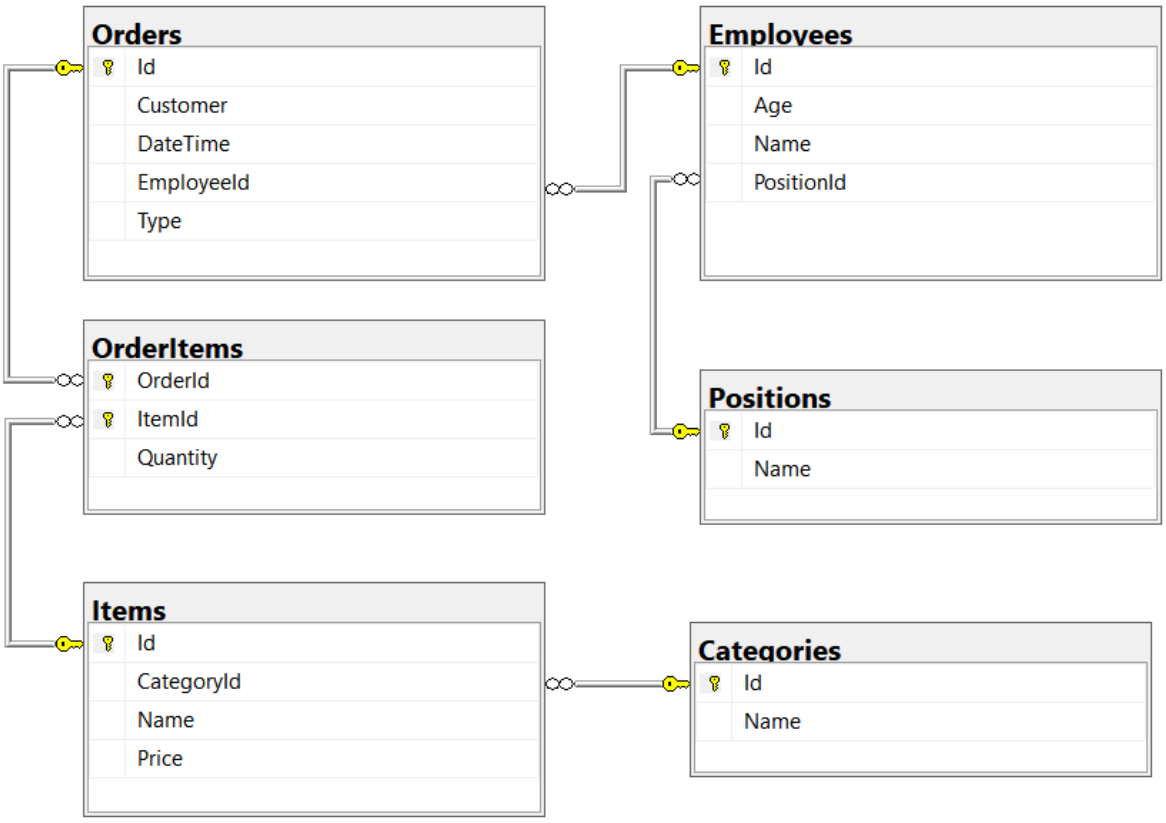
Entity Framework Core: Exam

Exam problems for the [Databases Advanced - Entity Framework course @ SoftUni]. Submit your solutions in the SoftUni judge system (delete all "bin"/"obj" folders alongside with the "packages" folder).

Your task is to create a database application using **Entity Framework Core** using the **Code First** approach. Design the **domain models** and **methods** for manipulating the data, as described below.

Fast Food

Create an application which models a **fast food point of sale system**. Employees process orders for customers. Orders have items inside them. Each category has zero or more items. For more details about the **models** and their **relations** read further.



Project Skeleton Overview

You are given a **project skeleton**, which includes the following items:

- **`FastFood.App`** – contains the **`Startup`** class, which is the **entry point of the application**. Also contains an **`AutoMapper`** profile, which you can configure if you choose to use **`AutoMapper`** in your app.
- **`FastFood.Data`** – contains the **`FastFoodDbContext`** class and the **connection string**
- **`FastFood.Models`** – contains the **entity classes**
- **`FastFood.DataProcessor`** – contains the **`Serializer`** and **`Deserializer`** classes, which are used for **importing** and **exporting** data

Problem 1. Model Definition (50 pts)

Every employee has a **position** and **orders**, which they need to process. Every **order** has a **customer**, **order date** and a **list of items**. Every item has a **category**, a **name** and a **price**. **Categories** have a **list of items**.

Note: Foreign key navigation properties are required!

The application needs to store the following data:

Employee

- **`Id`** – integer, **Primary Key**
- **`Name`** – text with **min length 3** and **max length 30** (**required**)
- **`Age`** – integer **in the range [15, 80]** (**required**)
- **`PositionId`** – integer, **foreign key**
- **`Position`** – the employee's **position** (**required**)
- **`Orders`** – the **orders** the employee has processed

Position

- **`Id`** – integer, **Primary Key**
- **`Name`** – text with **min length 3** and **max length 30** (**required, unique**)
- **`Employees`** – Collection of type **`Employee`**

Category

- **`Id`** – integer, **Primary Key**
- **`Name`** – text with **min length 3** and **max length 30** (**required**)
- **`Items`** – collection of type **`Item`**

Item

- **`Id`** – integer, **Primary Key**
- **`Name`** – text with **min length 3** and **max length 30** (**required, unique**)
- **`CategoryId`** – integer, foreign key
- **`Category`** – the item's **category** (**required**)
- **`Price`** – decimal (**non-negative, minimum value: 0.01**, **required**)
- **`OrderItems`** – collection of type **`OrderItem`**

Order

- **`Id`** – integer, **Primary Key**
- **`Customer`** – text (**required**)
- **`DateTime`** – date and time of the order (**required**)
- **`Type – OrderType`** enumeration with possible values: "**`ForHere`**, **`ToGo`** (default: **`ForHere`**)" (**required**)
- **`TotalPrice`** – decimal value (calculated property, **(not mapped to database)**, **required**)
- **`EmployeeId`** – integer, **foreign key**
- **`Employee`** – The employee who will process the order (**required**)
- **`OrderItems`** – collection of type **`OrderItem`**

OrderItem

- **`OrderId`** – integer, **Primary Key**

- **`Order`** – the item's **order** (**required**)
- **`ItemId`** – integer, **Primary Key**
- **`Item`** – the order's **item** (**required**)
- **`Quantity`** – the quantity of the **item** in the **order** (**required**, **non-negative** and **non-zero**)

Problem 2. Data Import (30pts)

For the functionality of the application, you need to create several methods that manipulate the database. The **project skeleton** already provides you with these methods, inside the **`FastFood.DataProcessor`** project inside your solution. Use **Data Transfer Objects** as needed.

Use the provided **JSON** and **XML** files to populate the database with data. Import all the information from those files into the database.

You are **not allowed** to modify the provided JSON and XML files.

If a record does not meet the requirements from the first section, print an error message:

Error message
`Invalid data format.`

JSON Import (20 pts)

Import Employees

Using the file **employees.json**, import the data from that file into the database. Print information about each imported object in the format described below.

Constraints

- If any validation errors occur (such as if their **name or position are too long/short** or their **age** is out of range) proceed as described above
- If a position **doesn't exist yet** (and the position and rest of employee data is **valid**), **create it**.
- If an employee is **invalid**, **do not** import their **position**.

Example

employees.json

```
[
  {
    "Name": "N",
    "Age": 20,
    "Position": "Invalid"
  },
  {
    "Name": "Too Young",
    "Age": 14,
    "Position": "Invalid"
  },
  {
    "Name": "Too Old",
    "Age": 81,
    "Position": "Invalid"
  },
  {
```

```
    "Name": "Invalid Position",
    "Age": 20,
    "Position": ""
  },
  {
    "Name": "InvalidPosition",
    "Age": 20,
    "Position": "Invalidddddddddddddddddddddddddd"
  },
  {
    "Name": "Magda Bjork",
    "Age": 44,
    "Position": "CEO"
  },
…
```

Output

```
Invalid data format.
Invalid data format.
Invalid data format.
Invalid data format.
Invalid data format.
Record Magda Bjork successfully imported.
…
```

Import Items

Using the file **items.json**, import the data from that file into the database. Print information about each imported object in the format described below.

Constraints

- If any validation errors occur (such as invalid item name or invalid category name), **ignore** the entity and **print an error message**.
- If an item with the same name **already exists**, **ignore** the entity and **do not import it**.
- If an item's category **doesn't exist**, **create it** along with the item.

Example

items.json

```
[
  {
    "Name": "Hamburger",
    "Price": 0.00,
    "Category": "Invalid"
  },
  {
    "Name": "Hamburger",
    "Price": -5.00,
    "Category": "Invalid"
  },
  {
    "Name": "x",
    "Price": 1.00,
    "Category": "Invalid"
  },
```

```
  {
    "Name": "Invalidddddddddddddddddddddddddd",
    "Price": 1.00,
    "Category": "Invalid"
  },
  {
    "Name": "Invalid",
    "Price": 1.00,
    "Category": "x"
  },
  {
    "Name": "Invalid",
    "Price": 1.00,
    "Category": "Invalidddddddddddddddddddddddddd"
  },
  {
    "Name": "Hamburger",
    "Price": 5.00,
    "Category": "Beef"
  },
  {
    "Name": "Hamburger",
    "Price": 1.00,
    "Category": "Beef"
  },
  {
    "Name": "Cheeseburger",
    "Price": 6.00,
    "Category": "Beef"
  },
…
```

Output

```
Invalid data format.
Invalid data format.
Invalid data format.
Invalid data format.
Invalid data format.
Invalid data format.
Record Hamburger successfully imported.
Invalid data format.
Record Cheeseburger successfully imported.
```

XML Import (10 pts)

Import Orders

Using the file **orders.xml**, import the data from the file into the database. Print information about each imported object in the format described below.

If any of the model requirements is violated continue with the next entity.

Constraints

- The order dates will be in the format "**dd/MM/yyyy HH:mm**". Make sure you use **CultureInfo.InvariantCulture**.
- If the order's **employee** doesn't exist, **do not** import the order.

- If **any** of the **order's items** do not exist, **do not** import the order.
- If there are any other validation errors (such as **negative** or **non-zero price**), proceed as described above.
- Every employee will have a **unique name**

Example

orders.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Orders>
  <Order>
    <Customer>Garry</Customer>
    <Employee>Maxwell Shanahan</Employee>
    <DateTime>21/08/2017 13:22</DateTime>
    <Type>ForHere</Type>
    <Items>
      <Item>
        <Name>Quarter Pounder</Name>
        <Quantity>2</Quantity>
      </Item>
      <Item>
        <Name>Premium chicken sandwich</Name>
        <Quantity>2</Quantity>
      </Item>
      <Item>
        <Name>Chicken Tenders</Name>
        <Quantity>4</Quantity>
      </Item>
      <Item>
        <Name>Just Lettuce</Name>
        <Quantity>4</Quantity>
      </Item>
    </Items>
  </Order>
...
</Orders>
```

Output

```
Order for Garry on 21/08/2017 13:22 added
```

Problem 3. Data Export (20 pts)

Use the provided methods in the **`FastFood.DataProcessor`** project. Usage of **Data Transfer Objects** is **optional**.

JSON Export (10 pts)

Export All Orders by Employee

The given method in the project skeleton receives an **employee name** and an **order type** as **strings**. Export all **orders** that were processed by the **employee** with that **name**, which have **that order type**. For each order, get the customer's **name** and the **order's items** with their **name**, **price** and **quantity**. Apart from that, for every order, also list the **total price** of the order. Sort the orders by their **total price** (**descending**), then by the **number of items** in the order (**descending**). Finally, also export the **total money made** from all the orders.

Example

Serializer.ExportOrdersByEmployee(context, "Avery Rush", "ToGo")

```
{
  "Name": "Avery Rush",
  "Orders": [
    {
      "Customer": "Stacey",
      "Items": [
        {
          "Name": "Cheeseburger",
          "Price": 6.00,
          "Quantity": 5
        },
        {
          "Name": "Double Cheeseburger",
          "Price": 6.50,
          "Quantity": 3
        },
        {
          "Name": "Luigi",
          "Price": 2.10,
          "Quantity": 5
        },
        {
          "Name": "Bacon Deluxe",
          "Price": 9.00,
          "Quantity": 1
        }
      ],
      "TotalPrice": 69.00
    },
    {
      "Customer": "Pablo",
      "Items": [
        {
          "Name": "Double Cheeseburger",
          "Price": 6.50,
          "Quantity": 3
        },
        {
          "Name": "Bacon Deluxe",
          "Price": 9.00,
          "Quantity": 5
        }
      ],
      "TotalPrice": 64.50
    },
    {
      "Customer": "Bobbie",
      "Items": [
        {
          "Name": "Tuna Salad",
          "Price": 3.00,
          "Quantity": 2
        },
        {
          "Name": "Crispy Fries",
          "Price": 2.00,
```

```
          "Quantity": 5
        },
        {
          "Name": "Fries",
          "Price": 1.50,
          "Quantity": 2
        }
      ],
      "TotalPrice": 19.00
    },
    {
      "Customer": "Joann",
      "Items": [
        {
          "Name": "Minion",
          "Price": 2.20,
          "Quantity": 2
        },
        {
          "Name": "Bacon Deluxe",
          "Price": 9.00,
          "Quantity": 1
        }
      ],
      "TotalPrice": 13.40
    }
  ],
  "TotalMade": 165.90
}
```

XML Export (10 pts)

Export Categories with their Most Popular Item

Use the method provided in the project skeleton, which receives a string of **comma-separated category names**. Export the **categories**: for each **category**, export its **most popular item**. The most popular item is the item from the category, which made the **most money** in **orders**. **Sort** the categories by **the amount of money the most popular item made (descending)**, then by **the times the item was sold** (**descending**).

Example

Serializer.ExportCategoryStatistics(context, "Chicken,Drinks,Toys")

```
<Categories>
  <Category>
    <Name>Chicken</Name>
    <MostPopularItem>
      <Name>Chicken Tenders</Name>
      <TotalMade>44.00</TotalMade>
      <TimesSold>11</TimesSold>
    </MostPopularItem>
  </Category>
  <Category>
    <Name>Toys</Name>
    <MostPopularItem>
      <Name>Minion</Name>
      <TotalMade>24.20</TotalMade>
```

```
      <TimesSold>11</TimesSold>
    </MostPopularItem>
  </Category>
  <Category>
    <Name>Drinks</Name>
    <MostPopularItem>
      <Name>Purple Drink</Name>
      <TotalMade>9.10</TotalMade>
      <TimesSold>7</TimesSold>
    </MostPopularItem>
  </Category>
</Categories>
```

Problem 4. Bonus Task (10 pts)

Implement the bonus method in the **FastFood.DataProcessor** project for an **additional amount** of points.

Update Item Price

Implement the method **DataProcessor.Bonus.UpdateItemPrice**, which receives an item's **name** and a **new price**. Your task is to **find the item** by that name and **update its price**.

After the price is updated, return the message "**{item.Name} Price updated from ${oldPrice:F2} to ${newPrice:F2}**".

If the item is not found, return "**Item {item.Name} not found!**"

Examples

DataProcessor.Bonus.UpdateItemPrice(context, "Cheeseburger", 6.50m)
Cheeseburger Price updated from $6.00 to $6.50

DataProcessor.Bonus.UpdateItemPrice(context, "Ribs", 8.00m)
Item Ribs not found!