# Databases Frameworks – Hibernate and Spring Data

# Exam

Exam problems for the [Databases Advanced – Hibernate and Spring Data @ SoftUni](https://softuni.bg/courses/databases-advanced-hibernate).

Download the provided **skeleton** and use it in your solutions. You are **not allowed** to change the structure or the names of the provided interfaces and classes. **It is very important to use the specified dependency versions in the provided pom.xml file.** Any usage of other dependencies is at your own risk.

Submit your project solutions in the SoftUni judge system, as a zip archive file, **not including the test folder**.

Your task is to create a database console application using **Hibernate and Spring Data** 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

**Controllers** – the main classes responsible for importing exporting data. Use their importDataFromJSON, importDataFromXML, exportOrdersByEmployeeAndOrderType(String employeeName, String orderType) and getCategoriesWithMostPopularItemsSorted(List<String> categoryNames)

**DTO’s and Entities** – your database models and import/export classes

**Parser –** the interface which has to be implemented by your JSON and XML parsers

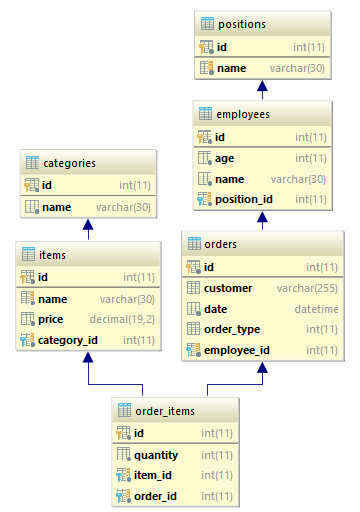
**Repositories** – the repositories assigned to a single model in the database

**Service interfaces –** interfaces which have to be implemented by your own service implementations. Place the main validation and insertion logic there

## Problem 1. Model Definition (48 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**.

The application needs to store the following data:



### Employee (8pts)

Each employee has:

* id – integer, primary identification field
* name – text with **min length** **3** and **max length 30** (**required**)
* age– integer **in the** **range** **[15, 80]** (**required**)
* position – the employee’s **position** (**required**)
* orders – the **orders** the employee has processed

### Position (8pts)

* id – integer, primary identification field
* name – text with **min length** **3** and **max length 30** (**required, unique**)
* employees – collection of employees which belong to the certain position

### Category (8pts)

* id – integer, primary identification field
* name – text with **min length 3** and **max length 30** (**required**)
* items – collection of items belonging to that category

### Item (8pts)

* id – integer, primary identification field
* name – text with **min length 3** and **max length 30** (**required, unique**)
* category – the item’s **category** (**required**)
* price – decimal (**non-negative, minimum value: 0.01**, **required**)
* OrderItems – collection of type OrderItem

### Order (8pts)

* id – integer, primary identification field
* customer – **text (required)**
* date – **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**)
* employee – the employee who will process the order (**required**)
* orderItems – collection of type OrderItem

### OrderItem (8pts)

* id – integer, **Primary Key**
* order – the item’s **order**
* item – the order’s item
* quantity – the quantity of the **item** in the **order** (**required**, **non-negative** and **non-zero**)

## Problem 2. Data Import (32 pts)

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 Parser inteface. Implement those methods in the JSONParser and XMLParser objects accordingly. Use the provided **Data Transfer Objects** and implement them 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:**

|  |  |
| --- | --- |
| **Success message** | **Error message** |
| Record {…} successfully imported. | Error: Invalid data. |

### 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 occurs (such as if their **name** **or position are too long/short** or their **age** is out of range) proceed as described aboves
* 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": "Invaliddddddddddddddddddddddddd"  },  {  "name": "Magda Bjork",  "age": 44,  "position": "CEO"  },  …  ] |
| **Output** |
| Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  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 a category **doesn’t exist yet** (and the rest of item data is **valid**), **create it**.

##### 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": "Invaliddddddddddddddddddddddddd",  "price": 1.00,  "category": "Invalid"  },  {  "name": "Invalid",  "price": 1.00,  "category": "x"  },  {  "name": "Invalid",  "price": 1.00,  "category": "Invaliddddddddddddddddddddddddd"  },  {  "name": "Hamburger",  "price": 5.00,  "category": "Beef"  },  {  "name": "Hamburger",  "price": 1.00,  "category": "Beef"  },  {  "name": "Cheeseburger",  "price": 6.00,  "category": "Beef"  },  … |
| **Output** |
| Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Error: Invalid data.  Record Hamburger successfully imported.  Error: Invalid data.  Record Cheeseburger successfully imported. |

### XML Import (12 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 you have problems with importing the JSON files, you are provided with a exported-db.sql file in your resources zip. Use it to manually import data so you can test the orders insertion.**

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

|  |  |
| --- | --- |
| **Success message** | **Error message** |
| Order for Garry on 21/08/2017 13:22 added. | Error: Invalid data. |

##### Constraints

* The order dates will be in the format “dd/MM/yyyy HH:mm”.
* 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, proceed as described above.

##### Example

|  |
| --- |
| **orders.xml** |
| *<?*xml version="1.0" encoding="utf-8"*?>* <orders>  <order>  <customer>Garry</customer>  <employee>Maxwell Shanahan</employee>  <date>21/08/2017 13:22</date>  <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>  … |
| **Output** |
| Order for Garry on 21/08/2017 13:22 added. |

## Problem 3. Data Export (20 pts)

**Use the provided methods in the** Controllers and Services.

### JSON Export

#### 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**. Sort the orders by their **total price** (**descending**), then by the **number of items** in the order (**descending**). Sort the items in each order by id.

##### Example

|  |
| --- |
| exportOrdersByEmployeeAndOrderType(“Avery Rush”, “ToGo”) |
| {  "employeeName": "Avery Rush",  "orders": [  {  "customer": "Stacey",  "items": [  {  "name": "Cheeseburger",  "price": 6.00,  "quantity": 5  },  {  "name": "Double Cheeseburger",  "price": 6.50,  "quantity": 3  },  {  "name": "Bacon Deluxe",  "price": 9.00,  "quantity": 1  },  {  "name": "Luigi",  "price": 2.10,  "quantity": 5  }  ]  },  {  "customer": "Pablo",  "items": [  {  "name": "Double Cheeseburger",  "price": 6.50,  "quantity": 3  },  {  "name": "Bacon Deluxe",  "price": 9.00,  "quantity": 5  }  ]  },  {  "customer": "Bobbie",  "items": [  {  "name": "Fries",  "price": 1.50,  "quantity": 2  },  {  "name": "Crispy Fries",  "price": 2.00,  "quantity": 5  },  {  "name": "Tuna Salad",  "price": 3.00,  "quantity": 2  }  ]  },  {  "customer": "Joann",  "items": [  {  "name": "Bacon Deluxe",  "price": 9.00,  "quantity": 1  },  {  "name": "Minion",  "price": 2.20,  "quantity": 2  }  ]  }  ]  } |

### Bonus: XML Export (10pts)

#### 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

|  |
| --- |
| getCategoriesWithMostPopularItemsSorted(List<String> categoryNames) |
| <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <categories>  <category>  <name>Chicken</name>  <most-popular-item>  <name>Chicken Tenders</name>  <total-made>44.0</total-made>  <times-sold>11</times-sold>  </most-popular-item>  </category>  <category>  <name>Toys</name>  <most-popular-item>  <name>Minion</name>  <total-made>24.200000000000003</total-made>  <times-sold>11</times-sold>  </most-popular-item>  </category>  <category>  <name>Drinks</name>  <most-popular-item>  <name>Purple Drink</name>  <total-made>9.1</total-made>  <times-sold>7</times-sold>  </most-popular-item>  </category>  </categories> |