Exercises: Hibernate Relations

This document defines the **exercise assignments** for the "Databases Advanced – Hibernate" course @ Software University.

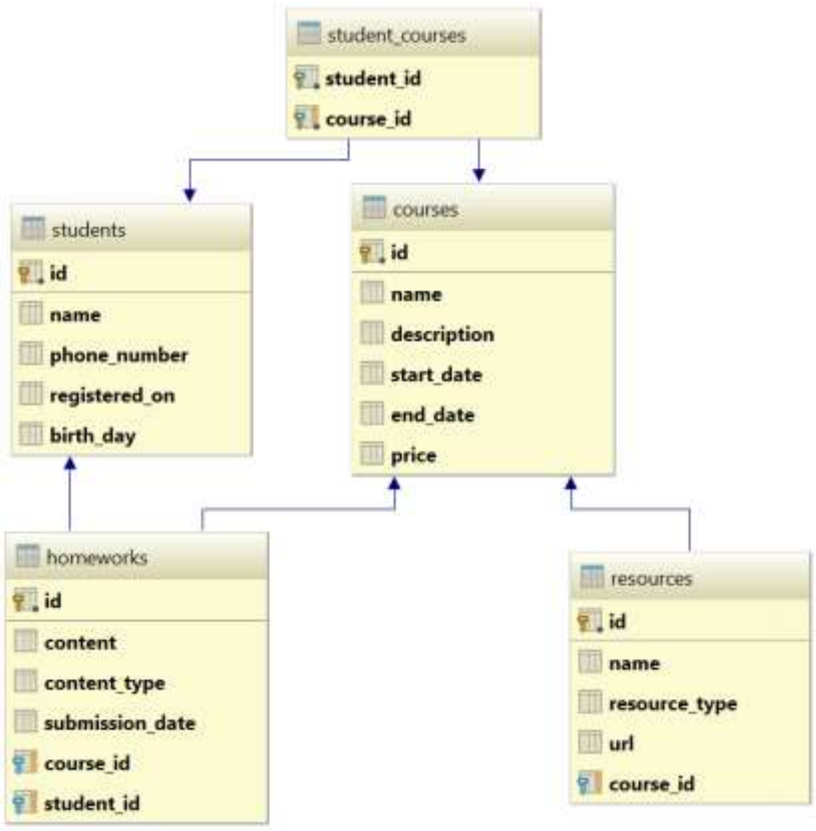
1. Code First Student System

Your task is to create a database for the **Student System**, using the **Code First** approach. Model the following tables:

- **Students**: id, name, phone number (optional), registration date, birthday (optional)
- **Courses**: id, name, description (optional), start date, end date, price
- **Resources**: id, name, type of resource (video / presentation / document / other), URL
- **Homework**: id, content, content-type (e.g. application/pdf or application/zip), submission date

Table relations:

- **Students** can be in **many courses**
- **Courses** can have **many students**
- **Courses** can have **many resources**
- **One course** can have **many homework submissions**
- **Homework submissions** have a **student**



Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

2. Seed Some Data in the Database

Write a **seed method** that fills the database with sample data (randomly generated).

Fill a few **students**, **courses**, **resources** and **homework submissions**.

Run the application several times to ensure that it seeds sample data **only once**.

3. Working with the Database

Write a console application that works with the EF data layer and performs the following CRUD operations:

1. Lists **all students** and their **homework submissions**. Select only their **names** and for each homework - **content** and **content-type**.
2. List **all courses** with their corresponding **resources**. Select the **course name** and **description** and everything for each **resource**. Order the courses by start date (ascending), then by end date (descending).
3. List **all courses** with **more than 5 resources**. Order them by **resources count** (descending), then by **start date** (descending). Select only the **course name** and the **resource count**.
4. List all **courses** which were active on a **given date** (choose the date depending on the data seeded to ensure there are results), and for each course count the **number of students enrolled**.
   Select the **course name**, **start** and **end date**, **course duration** (difference between end and start date) and **number of students enrolled**. Order the results by the **number of students** enrolled (in descending order), then by **duration** (descending).
5. For each student, calculate the **number of courses** he/she has enrolled in, the **total price** of these courses and the **average price per course** for the student.
   Select the **student name**, **number of courses**, **total price** and **average price**. Order the results by **total price** (descending), then by **number of courses** (descending) and then by the **student's name** (ascending).

4. Resource Licenses

Resources should now have many **licenses**. A **license** should have an **Id** and **Name**.

Make these changes using Code First Migrations. Make sure no data is lost after the update.

5. Friends

Now in that and the next several tasks we are going to extend the database where we created the table User from the previous exercise. Let's say that the **user can have many friends** that would be again other users (or in other words **many to many self-relationship**).

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

6. Albums

Previously 1 user was able to upload only 1 picture (just his/her profile picture). Now each user is capable of creating **personal albums**. Each album has name, background color and information whether is public or not. Each picture has title, caption and path on the file system. An album can contain many pictures and one picture can be present in many albums. Each user can have many albums but an album can have only one owner user.

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

7. Tags

Imagine how much cooler would be if the user can put tags on each album so they can be easily organized (such as, #NewYear2016, #HolidaySummer, #NoMakeup etc…). A tag is just simply a string without any spaces. Each album can have as many tags the user wants and each tag can be placed on unlimited number of albums.

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

8. Tag Attribute

Make a **`[Tag]`** attribute that would validate if the given string is valid tag. A valid tag is a string starting with pound sign (#), do not contain any spaces in it and is no more than 20 symbols long.

Write a static class **`TagTransofrmer`** that would have a single public static method inside **`Transform(string tag)`**. That method would convert given tag to a valid one (remove all spaces, put pound sign at first position if it is not present and reduce the length of the tag if it is more than 20 symbols).

Write a program that receives as an input tags and insert them into the database. Use the **`[Tag]`** attribute and **`TagTransformer`** class to make sure only valid attributes are inserted in the database.

Example

Input	Output
`#summer`	`#summer was added to database`
`myCat`	`#myCat was added to database`
`#no make up`	`#nomakeup was added to database`
`#aaaaaaaaaaaaaaaaaaXCutThisEnd`	`#aaaaaaaaaaaaaaaaaaX was added to database`
`me and my bff doing selfie`	`#meandmybffdoingself was added to database`

9. Shared Albums

Currently an album can have just one owner lets modify it so the user can share its albums with other users. To do that just change the type of the relationship between user and album from one to many to more appropriate one.

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

10. *User Roles

Right now, if some user share album with a friend for example. His friend has total control over his/her album. That means he can add or delete photos without the permission of the initial owner of the album. To restrict that we can set role for each user for given album. The roles should be:

- **Owner** - can modify the album
- **Viewer** - can only see the pictures in that album but cannot add or delete any

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

11. *Football Betting Database

Your task is to create a database for the **Football Bookmaker System**, using the **Code First** approach. Model the following tables:

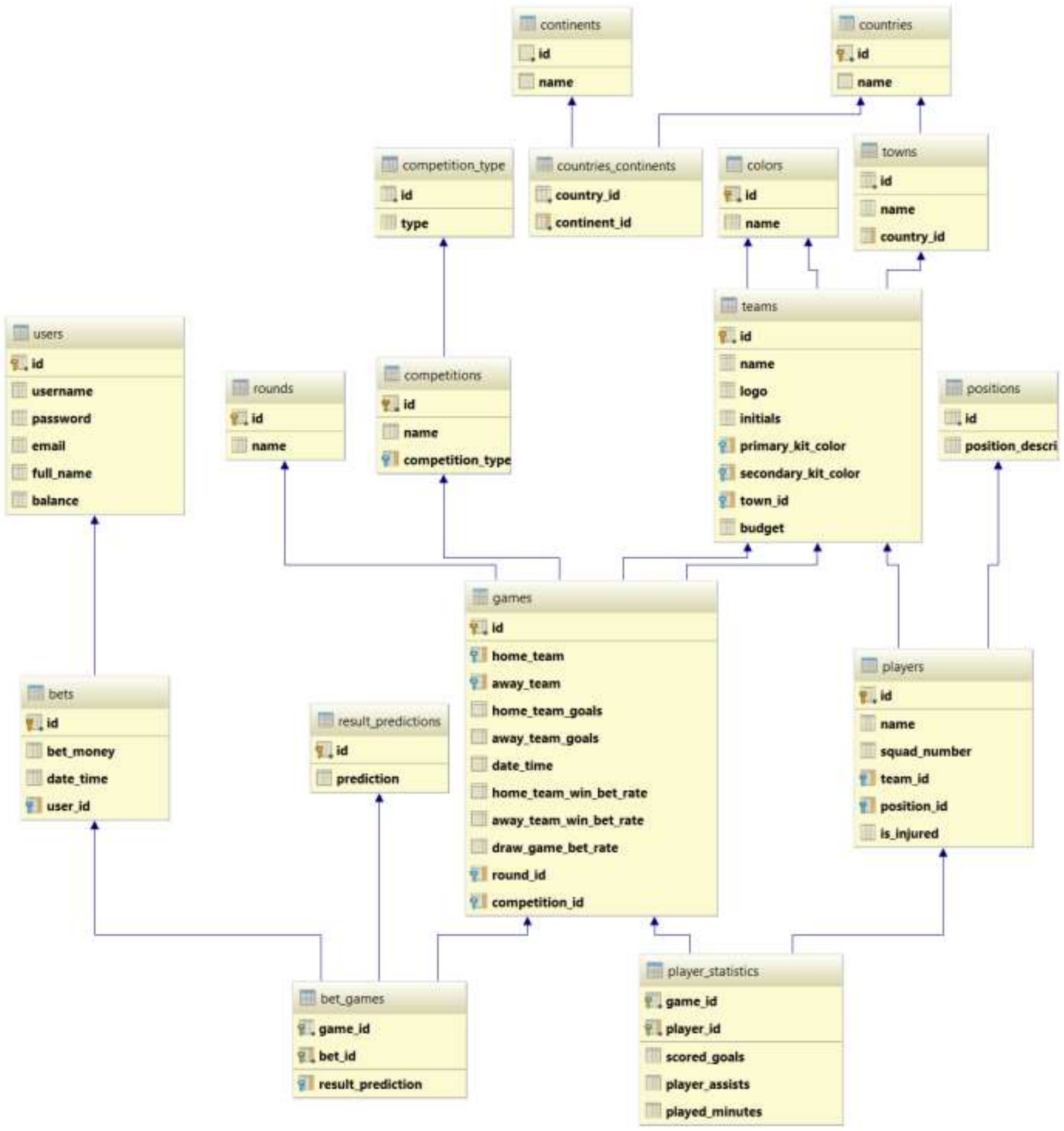
- **Teams** – Id, Name, Logo, 3 letter Initials (JUV, LIV, ARS…), Primary Kit Color, Secondary Kit Color, Town, Budget
- **Colors** – Id, Name
- **Towns** – Id, Name, Country
- **Countries** – Id (3 letters – for example BUL, USA, GER, FRA, ITA…), Name, Continent
- **Continents** – Id, Name
- **Players** – Id, Name, Squad Number, Team, Position, Is Currently Injured
- **Position** – Id (2 letters – GK, DF, MF, FW…), position description (for example – goal keeper, defender…)

- **PlayerStatistics** – Game, Player, Scored Goals, Player Assists, Played Minutes During Game, (PK = Game + Player)
- **Games** – Id, Home Team, Away Team, Home Goals, Away Goals, Date and Time of Game, Home team Win bet rate, Away Team Win Bet Rate, Draw Game Bet Rate, Round, Competition)
- **Rounds** – Id, Name (for example Groups, League, 1/8 Final, 1/4 Final, Semi-Final, Final…)
- **Competitions** – Id, Name, Type (local, national, international)
- **CompetitionTypes** – Id, Name
- **BetGame** – Game, Bet, Result Prediction (PK = Game + Bet)
- **Bets** – Id, Bet Money, Date and Time of Bet, User
- **ResultPrediction** – Id, Prediction (possible values - Home Team Win, Draw Game, Away Team Win)
- **Users** – Id, Username, Password, Email, Full Name, Balance

Table relationships:

- Team has one primary kit color and one secondary kit color
- Team resident in one town
- Each town can host several teams
- Town can be placed in one country and a country can have many towns
- Country can be placed in several continents and a continent can have many countries
- Player can play for one team and one team can have many players that play for it
- Player can play at one position and one position can be played by many players
- Player can play in many games and in each game, many players take part
- Additionally, for each player for given game is kept statistics such as scored goals, goal assists and minutes played during given game
- A game can be played in one round and in one round many games can be played
- A game can be played in one competition and in one competition many games can be played
- On a game, many bets can be placed and one bet can be on several games
- Each bet for given game must have prediction result
- A bet can be placed by only one user and one user can place many bets

Hint - Database Schema



12. Bills Payment System

Your task is to create a database for **Bills Payment System**, using the **Code First** approach. In the database, we should keep information about the **users** who are using that system (**first name, last name, email, password, billing details**). Every **billing detail** have **number** and **owner**. Also, there are **two types** of billing details **credit card** and **bank account**. The credit card has **card type, expiration month, expiration year**. And the bank account has **bank name** and **SWIFT code**.

Solve the task in 3 different ways. Use the following 3 approaches to make 3 different models of the classes and the database tables:

- Table per Hierarchy

- Table per Subclass
- Table per Concrete Class

Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

13. University System

Your task is to create a database for **University System**, using the **Code First** approach. In the database, we should keep information about students, teachers and courses.

- **Student** - first name, last name, phone number, average grade, attendance
- **Teacher** - first name, last name, phone number, email, salary per hour
- **Course** – name description, start date, end date, credits

Each student can be enrolled in many courses and in each course many students can be enrolled. A teacher can teach in many courses but one course can be taught only by one teacher.

Use class hierarchy to reduce code duplication. **Solve the task in 3 different ways**. Use the following 3 approaches to make 3 different models of the classes and the database tables.

- Table per Hierarchy
- Table per Subclass
- Table per Concrete Class

Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

14. Vehicles

Your task is to create a database for **Vehicles Info System,** using the **Code First** approach. In the database, we should keep information about different kind of vehicles. Each vehicle has **manufacturer**, **model**, **price** and **max speed**. There are two main types of vehicles: **motor** and **non-motor vehicles**. There is only one type of non-motor vehicles – **bike**. Bike has **shifts count** and **color**. All motor vehicles have **number of engines**, **engine type** and **tank capacity**. There are several types of motor vehicles:

- **Car** – number of doors, information about having insurance
- **Train** – locomotive, number of carriages, list of carriages
- **Plane** – airline owner, color, passengers’ capacity
- **Ship** – nationality, captain name, size of ship crew
    - **Cargo Ship** – max load kilograms
    - **Cruise Ship** – passengers’ capacity

All carriages have passengers’ seats capacity. There are three types of carriages

- **Passenger** – standing passengers capacity
- **Restaurant** – tables count
- **Sleeping** – beds count

Locomotive has **model** and **power**. Each locomotive can pull one train and one train can be pulled only by one locomotive.

Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

15. Bank System

Your task is to create a database for **Bank System,** using the **Code First** approach. In the database, we should keep information about banking accounts. There are two types of bank accounts:

- **Saving account** – account number, balance, interest rate
- **Checking account** – account number, balance, fee

The **operations** that can be performed with those accounts are:

- **Savings account** – deposit money, withdraw money, add interest
- **Checking account** – deposit money, withdraw money, deduct fee

16. Bank System Console Client

Extend the database from the previous exercise to support keeping information about **users**. A user has **username, password, email** and can have **many bank accounts**. Design a console application that uses that database and support the following commands:

Commands that can be executed when there is **no currently logged in user**:

- **`Register <username> <password> <email>`** - That command add new user to the database in case username, password and email are valid. Otherwise print appropriate message informing why the user cannot be registered. The requirements for valid parameters are:
  - **Username** – can contain only letters [a-Z] and numbers. Cannot start with number. Cannot be less than 3 symbols long
  - **Password** – must contain at least 1 lowercase letter, 1 uppercase letter and 1 digit. Also, must be more than 6 symbols long
  - **Email** – must be in format **<user>@<host>** where:
    - **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**_**' can appear between them.
    - **<host>** is a sequence of at least two words, separated by dots '**.**'. Each word is sequence of letters and can have hyphens '**-**' between the letters.
- **`Login <username> <password>`** - That command set the current logged in user if exists. Otherwise print appropriate message.

Commands that can be executed when there is **currently logged in user**:

- **`Logout`** – log out the user from the system. If there is no logged in user print appropriate message.
- **`Add SavingAccount <initial balance> <interest rate>`** - add saving account to the currently logged in user. Also, set the account number to random combination of 10 uppercase letters and digits. For example: “PX234ADG56”, “90M09JKE73”, etc.
- **`Add CheckingAccount <initial balance> <fee>`** **-** add checking account to the currently logged in user. Also, set the account number to random combination of 10 uppercase letters and digits.
- **`ListAccounts`** – prints a list of overall information for all accounts of currently logged in user in format:

  ```
  Saving Accounts:
  --{Account Number} {Current Balance}
  Checking Accounts:
  --{Account Number} {Current Balance}
  ```

  Order them **by account number ascending**.
- **`Deposit <Account number> <money>`** - adds money to the account with given number
- **`Withdraw <Account number> <money>`** - subtracts money from the account with given number
- **`DeductFee <Account number>`** - deduct the fee from the balance of the account with given number
- **`AddInterest <Account number>`** - add interest to the balance of the account with given number

After each command **print appropriate message** telling whether the command was successfully executed or not. If it is not print appropriate message telling what was the error. Use all of the **best practices** in programming and **suitable design patterns**.

Example

Input	Output

`Register vl Tsepesh89 vlad@rom.ro`
`Register vlad123 tspesh vlad@rom.ro`
`Register vlad123 Tsepesh89 -v-@-rom.ro`
`Register vlad123 Tsepesh89 vlad@rom.ro`
`Logout`
`Login vlad321 Tsepesh89`
`Login vlad123 smallPussyCat`
`Login vlad123 Tsepesh89`
`Add SavingsAccount 1000 0.2`
`Add CheckingAccount 100 4.20`
`Deposit A8234JDG9M 10.42`
`Withdraw A8234JDG9M 5`
`Deposit PO8FHH34GM 200`
`Withdraw PO8FHH34GM 45.2`
`AddInterest A8234JDG9M`
`DeductFee PO8FHH34GM`
`ListAccounts`
`Logout`
	`Incorrect username`
	`Incorrect password`
	`Incorrect email`
	`vlad123 was registered in the system`
	`Cannot log out. No user was logged in.`
	`Incorrect username / password`
	`Incorrect username / password`
	`Succesfully logged in vlad123`
	`Succesfully added account with number A8234JDG9M`
	`Succesfully added account with number PO8FHH34GM`
	`Account A8234JDG9M has balance of 1010.42`
	`Account A8234JDG9M has balance of 1005.42`
	`Account PO8FHH34GM has balance of 300.00`
	`Account PO8FHH34GM has balance of 254.80`
	`Added interest to A8234JDG9M. Current balance: 1206.50`
	`Deducted fee of PO8FHH34GM. Current balance: 250.60`
	`Accounts for user vlad123`
	`Saving Accounts:`
	`--A8234JDG9M 1206.50`
	`Checking Accounts:`
	`--PO8FHH34GM 250.60`
	`User vlad123 successfully logged out`

17. ***Bank System GUI

Use the logic and database of the previous exercise and replace the console client with graphical user interface with technology of your choice.