# Exercises: Best Practices and Architecture

This document defines the **exercise assignments** for the ["Databases Advanced – Entity Framework" course @ Software University.](https://softuni.bg/trainings/1442/databases-advanced-entity-framework-october-2016)

# Best Practices and Architecture

## Photo Share System

You are given a project [skeleton](http://svn.softuni.org/admin/svn/DB-Fundamentals/DB-Advanced-EntityFramework/Feb-2017/08.%20DB-Advanced-EntityFramework-Best%20Practices-and-Architecture/08.%20DB-Advanced-EntityFramework-Best-Practices-and-Architecture-PhotoShare-Skeleton.zip) of a **Photo Sharing System**. Take a look at it to get more familiar with that project. The database is modeled by code first approach and the models are fine (in other words there is nothing to modify in PhotoShare.Models project).

But the other project PhotShare.Client is poorly written. Your task is to **improve the architecture of the project.** Seed some data in the database.

Then **implement the missing commands** by the hints given in each command class and **fix any bugs** in the already implemented code.

### System functionality

The photo share system contains the following commands:

* **RegisterUser <username> <password> <repeat-password> <email>**Registers a new user.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] was registered successfully! | None |
| Passwords do not match | Passwords do not match! | ArgumentException |
| Username is taken | Username [username] is already taken! | InvalidOperationException |

Any other validation is already implemented and it should stay as is.

* **AddTown <town Name> <country Name>**

Adds new town. Town names must be unique.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Town [town] was added successfully! | None |
| Town already exists | Town [town] was already added! | ArgumentException |

Any other validation is already implemented and it should stay as is.

* **ModifyUser <username> <property> <new value>**Modifies current user.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [user] [property] is [value]. | None |
| User does not exist | User [username] not found! | ArgumentException |
| Property not found | Property [property] not supported! | ArgumentException |
| Value not valid for that property | Value [value] not valid.  [DetailedMessage] | ArgumentException |

The above command may be executed in the following formats:

ModifyUser <username> Password <NewPassword>

* Password must contain at least one lowercase letter and digit. If not:
  + Detailed Message - “Invalid Password”

ModifyUser <username> BornTown <newBornTownName>

* Town must exist in database. If not:
  + Detailed Message – “Town [town] not found!”

ModifyUser <username> CurrentTown <newCurrentTownName>

* Town must exist in database. If not:
  + Detailed Message – “Town [town] not found!”

Any other format different than from all the above formats is invalid.

* **DeleteUser <username>**Deletes a user.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] was deleted successfully! | None |
| User does not exist | User [username] not found! | ArgumentException |
| User is already deleted | User [username] is already deleted! | InvalidOperationException |

* **AddTag <tag>**Adds a tag. Tag names should be unique.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Tag [tag] was added successfully! | None |
| Tag already exists | Tag [tag] exists! | ArgumentException |

Tag validation is already implemented and should stay as is.

* **CreateAlbum <username> <albumTitle> <BgColor> <tag1> <tag2>...<tagN>**Adds an album. Album names should be unique.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Album [Album] successfully created! | None |
| User does not exist | User [username] not found! | ArgumentException |
| Album does exist | Album [album] exists! | ArgumentException |
| Background color does not exist | Color [color] not found! | ArgumentException |
| If any tag is not found in database | Invalid tags! | ArgumentException |

* **AddTagTo <album> <tag>**Adds a tag.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Tag [tag] added to [album]! | None |
| Album or tag does not exist | Either tag or album do not exist! | ArgumentException |

* **MakeFriends <username1> <username2>**Makes the first user [friend](http://orig13.deviantart.net/006e/f/2013/297/c/6/lol_amumu_by_enzanblues456-d6rnrta.jpg) with the second one. :)

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Friend [username2] added to [username1] | None |
| Any of the users do not exist. | [user\*] not found! *\*Pick the first not existing possible user* | ArgumentException |
| They are already friends | [username2] is already a friend to [username1] | InvalidOperationException |

* **ListFriends <username>**List usernames of all friends for given user sorted alphabetically.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Friends:  -[username1]  …  -[usernameN] | None |
| User does not have any friends. | No friends for this user. :( | None |
| User does not exist | User [user] not found! | ArgumentException |

* **ShareAlbum <albumId> <username> <permission>**Makes specified user to be part of given album.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Username [user] added to album [album] ([permission]) | None |
| Album does not exist | Album [album] not found! | ArgumentException |
| User does not exist | User [user] not found! | ArgumentException |
| Permission not valid | Permission must be either “Owner” or “Viewer”! | ArgumentException |

* **UploadPicture <albumName> <pictureTitle> <pictureFilePath>**Creates picture and atteches it to specified album

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Picture [picture] added to [album]! | None |
| Album does not exist | Album [Album] not found! | ArgumentException |

* **Exit**Exits application.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Good Bye! | None |

If command is **NOT** in format specified above (either command name is wrong or input arguments count) throw InvalidOperationException with message: “Command <CommandName> not valid!”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RegisterUser admin abc123 abc123 user@softuni.bg  ModifyUser admin BornTown Sofia  AddTown Sofia Bulgaria  ModifyUser admin BornTown Sofia  UploadPicture Nature Vitosha C://Pictures/Vitosha.png  CreateAlbum admin Nature Magenta nature  AddTag nature  CreateAlbum admin Nature Magenta nature  UploadPicture Nature Vitosha C://Pictures/Vitosha.png  Exit | User admin was registered successfully!  Value Sofia not valid! Town Sofia not found!  Town Sofia was added successfully!  User admin BornTown is Sofia.  Album Nature not found!  Invalid tags!  Tag #nature was added successfully!  Album Nature successfully created!  Picture Vitosha added to Nature!  Good Bye! |
| RegisterUser peter pan123 pan123 peter@pan.com  RegisterUser peter aaa a123 pesho@pan.com  RegisterUser capt hook123 hook123 capitan@hook.com  MakeFriends peterr capp  MakeFriends peter capp  MakeFriends peter capt  RegisterUser jack jack123 jack123 jack@j.com  MakeFriends peter jack  ListFriends peter  RegisterUser user  Exit | User peter was registered successfully!  Username peter is already taken!  User capt was registered successfully!  User peterr not found!  User capp not found!  Friend capt added to pan!  User jack was registered successfully!  Friend jack added to pan!  Friends  -capt  -jack  Command RegisterUser not valid!  Good Bye! |
| RegisterUser tomCat tom123 tom123 tom@t.com  RegisterUser jerry jerry123 jerry123 jerry@j.com  RegisterUser butch butch123 butch123 butch@b.com  AddTag #cartoon  CreateAlbum tomCat Tom&Jerry Blue cartoon  ShareAlbum 1 jerry Owner  AddTag #childhood  AddTagTo Tom&Jerry childhood  Exit | User tom was registered successfully!  User jerry was registered successfully!  User jerry was registered successfully!  Tag #cartoon was added successfully!  Album Tom&Jerry successfully created!  Username jerry added to album Tom&Jerry (Owner)  Tag #childhood was added successfully!  Tag #childhood added to Tom&Jerry!  Good Bye! |

## Extend Photo Share System

Extend the **Photo Share System** by implementing two more commands:

* **Login <username> <password>**Logs a user into the system and keep a reference to it until the “Logout” command is called.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] successfully logged in! | None |
| Either user does not exist or password does not match | Invalid username or password! | ArgumentException |
| User has logged in already | You should logout first! | ArgumentException |

* **Logout**Logs out a user from the system.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] successfully logged out! | None |
| There is no user logged in. | You should log in first in order to logout. | InvalidOperationException |

**Modify all commands accordingly**.

Logged user can:

* Modify his own profile
* Add friends to himself
* Add tag
* Add town
* Share an album (only if he is owner of the album)
* Upload picture (only if he is owner of the album that picture is uploaded to)
* Add tag to picture (which is part of at least one album which he is owner of)
* Delete user (can delete only himself/herself)
* List friends of any user
* Logout

Non-logged user can:

* List friends of any user
* Register
* Login

If any of these rules are violated print: “Invalid credentials!” and throw InvalidOperationException.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RegisterUser thor tor123 tor123 tor@t.com  Login thor tor1234  Login thor tor123  Login thor tor123  AddTag thunder  RegisterUser loki loki123 loki123 loki@l.com  Logout  RegisterUser loki loki123 loki123 loki@l.com  Login loki loki123  Exit | User thor was registered successfully!  Invalid username or password! User tor successfully logged in!  Invalid Credentials!  Tag #thunder was added successfully!  Invalid Credentials!  User tor successfully logged out!  User loki was registered successfully!  User loki successfully logged in!  Good Bye! |

## \*Bus Tickets System

Your task is to design a database for **Bus Tickets System** using the **code first** approach. The database should keep information about:

* **bus companies** - name, nationality, rating
* **tickets** - price, seat, customer, trip
* **customers** - first name, last name, date of birth, gender, home town
* **trips** - departure time, arrival time, status, origin bus station, destination bus station, bus company
* **bus stations** - name, town
* **towns** - name, country
* **reviews** - content, grade, bus station, customer, date and time of publishing
* **bank accounts** - account number, balance, customer

Each ticket is bought from a customer for certain trip. A customer has only one home town. Every trip has origin and destination bus station and it is organized by only one bus company. A bus station is located in only one town and one town can have many bus stations located in it. Reviews are written for a certain bus company and a bus company can have many reviews. One customer can write many reviews but a single review can be written only by one customer. Bank account can be owned by one customer and each customer can own only one bank account.

Gender can be only male, female or not specified. The status of the trip can be departed, arrived, delayed or cancelled. Grade of a review can any be a floating-point number in range [1, 10].

When database is up and running **seed** it with some **sample records in each table**.

Now let’s **make a command line application** that would **use that database and provide the following functionalities**:

* **print information for trips for a given bus station –** Print a list of arrivals and departures buses for given bus station id in the format provided below
* **buy ticket –** Insert new ticket and reduce the balance from customers’ bank account. Consider managing all cases such as the customer does not have enough money in his/her bank account.
* **publish review –** insert new review from given user into the database
* **print reviews –** print a list of reviews for a given bus company in the format provided below

|  |  |
| --- | --- |
| **Command** | **Successful Output** |
| print-info {Bus Station ID} | {Bus Station Name}, {Town}  Arrivals:  From {origin station} | Arrive at: {Arrival Time} | Status: {status}  Departures:  To {destination station} | Depart at: {Departure Time} | Status {status} |
| buy-ticket {customer ID} {Trip ID} {Price} {Seat} | Customer {Full Name} bought ticket for trip {Trip ID} for {Price} on seat {Seat} |
| publish-review {Customer ID} {Grade} {Bus Company Name} {Content} | Customer {Full Name} published review for company {Company Name} |
| print-reviews {Bus Company ID} | {Id} {Grade} {Date and Time}  {Customer Full Name}  {Content} |

The application should **receive and execute unlimited number of commands** until the **“exit”** command is received.

Use all of the **best practices** you know to design and write your application. Finally **write unit tests** to make **sure all these functionalities** work correctly.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| print-info 1  exit | Sofia Central Station, Sofia  Arrivals:  From: Burgas | Arrive at: 14:30 | Status: Departed  From: Svishtov | Arrive at: 07:30 | Status: Arrived  From: V.Tarnovo | Arrive at: 14:30 | Status: Departed  Departures:  To: Varna | Depart at: 14:40 | Status: Delayed  To: Plovdiv | Depart at: 15:30 | Status: Cancelled |
| buy-ticket 2 323 14.20 A4  buy-ticket 3 333 -12.00 B8  buy-ticket 9 874 6.90 A1  exit | Customer John Doe bought ticket for trip 323 for $14.20 on seat A4  Invalid price  Insufficient amount of money for customer Harry Potter with bank account number BGR33133235 |
| publish-review 2 10 BusTrip2000 Excellent trip! Look forward to travel again.  publish-review 3 2 BusCompany2001 Awful and dirty bus. Cannot recommend it to anyone.  exit | John Doe’s review was succesfully published  Chuck Norris’ review was successfully published |
| publish-review 2 8.5 BusTrip2000 Would recommend it but the driver needs to stop smoking while driving.  print-reviews 1  exit | John Doe’s review was succesfully published  1 10 2016/11/15 10:46:18  John Doe  Excellent trip! Look forward to travel again.  2 8.5 2016/11/18 12:20:00  John Doe  Would recommend it but the driver needs to stop smoking while driving. |

## \*\*Extend Bus Tickets System

Implement one additional command **change-trip-status**. That command would change the status of a given trip

|  |  |
| --- | --- |
| **Command** | **Successful Output** |
| change-trip-status {Trip Id} {New Status} | Trip from {Origin Bus Station Town} to {Destination Bus Station Town} on {Trip Departure Date and Time}  Status changed from {Old Status} to {New Status} |

Add **new table to the database** to keep information about **arrived trips** (actual arrival time, origin bus station, destination bus station, passengers count).

In case a **trip status is changed to Arrived, automatically add new record** in the **arrived trips table** with the required information.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| change-trip-status 2 Departed  change-trip-status 3 Cancelled  change-trip-status 134 Arrived  exit | Trip from Burgas to Sofia on 2016/11/15 10:45:00  Status changed from Cancelled to Departed  Trip from Sofia to Varna on 2016/11/15 10:00:00  Status changed from Delayed to Cancelled  Trip from Plovdiv to Varna on 2016/11/14 15:30:00  Status changed from Departed to Arrived  On 2016/11/14 22:32:43 - 34 passengers arrived at Varna from Plovdiv |

# Performance

This part will concentrate on the performance side when executing queries. For this part of the exercise you should **NOT** submit anything. Use database “Employees”. Download it [here](http://svn.softuni.org/admin/svn/DB-Fundamentals/DB-Advanced-EntityFramework/Feb-2017/08.%20DB-Advanced-EntityFramework-Best%20Practices-and-Architecture/Employees-Database.sql).

Execute the script given above, create **new** console **project** and in it **add** new **ADO.NET** entity model **from database** “**Employees**”, name it EmployeeContext.

Before we start we have to setup our environment in order to test the performance of the queries generated by Entity Framework:

|  |
| --- |
| EmployeesContext context = new EmployeesContext();  Stopwatch stopwatch = new Stopwatch();  long timePassed = 0L;  int testCount = 10; // Amount of tests to perform  for (int i = 0; i < testCount; i++)  {  // Clear all query cache  context.Database.ExecuteSqlCommand("CHECKPOINT; DBCC DROPCLEANBUFFERS;");  stopwatch.Start();  // **TODO: Method to execute query**  stopwatch.Stop();  timePassed += stopwatch.ElapsedMilliseconds;  stopwatch.Reset();  }  TimeSpan averageTimePassed = TimeSpan.FromMilliseconds(timePassed / (double)testCount);  Console.WriteLine(averageTimePassed); |

Put the code above inside your Main() method.

Keep in mind that when testing/analyzing performance there are **multiple factors** that we should consider beforehand. In our case: **size of data** (how many tables there are, how many rows they have and how they are related), the **structure** of our C# **query** (query that we make using C# code which later on EF transforms to SQL), the **complexity** of our C# **queries** (having multiple conditions in .Where() clause etc.), computer power and so on.

The following tasks should only give **estimated** judgement about how EF generates SQL code and on what to concentrate when looking for performance optimization.

## Lazy vs Eager (Performance case #1)

In “**Employees**” database we have employees and certain information about them. Your task is to collect (without printing) all **employees**: their first **name**, the name **department** name, their **address** text.

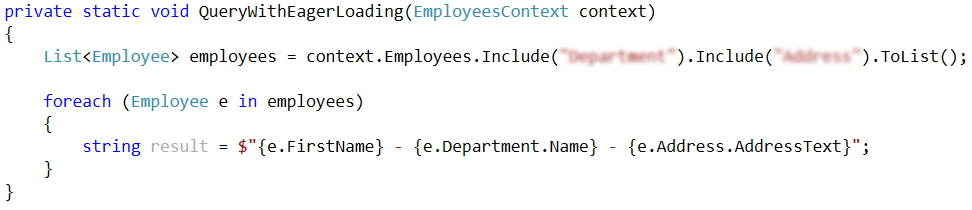
That’s not all, write **two** different **queries** to solve the problem: **first** using lazy loading and **second** using eager loading (Include). Compare with [SQL ExpressProfiler](https://expressprofiler.codeplex.com) (or a similar tool) how many sql queries have been made with each of the queries.

### Constraints

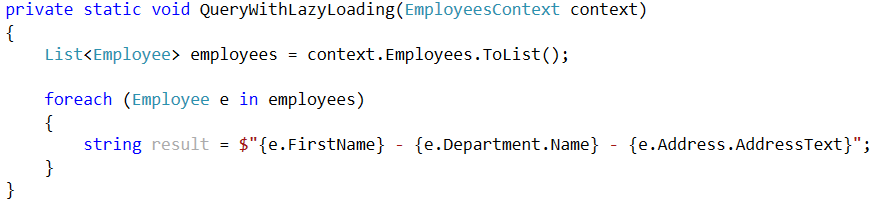
Do **not** use .Select() for any of the queries!

Create two methods: QueryWithLazyLoading and QueryWithEagerLoading. Let them accept EmployeeContext and they should not return any result (void). In them create your queries.

Here is how the solution with **eager** loading should look like:

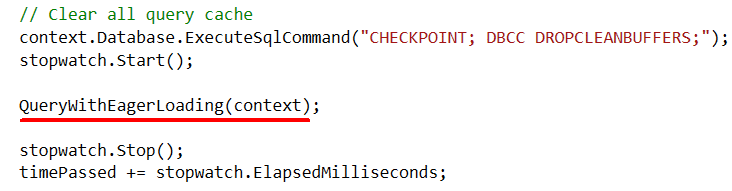


And here is how the “**lazy**” solution should look like:

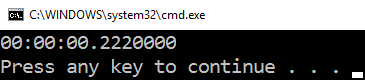


Note – do not print any results to the console (we are not testing how fast Console.WriteLine() is), but make sure that the needed data that is actually fetched. In the above examples if any of the departments or addresses is null – exception will be thrown (because we want on null department to get its name or on null address to get its address text).

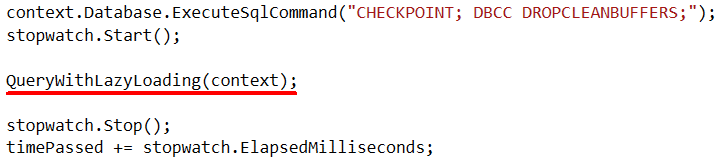
Then execute the program with both methods **SEPARATE**. Replace the //**TODO** with method call:



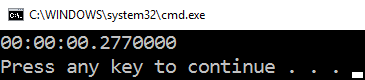
Which leads to result similar to this one (the result should be **near** this timespan):



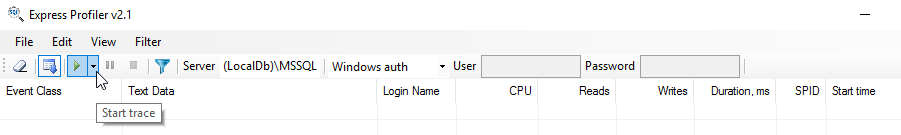
Now try the “**lazy**“ solution:



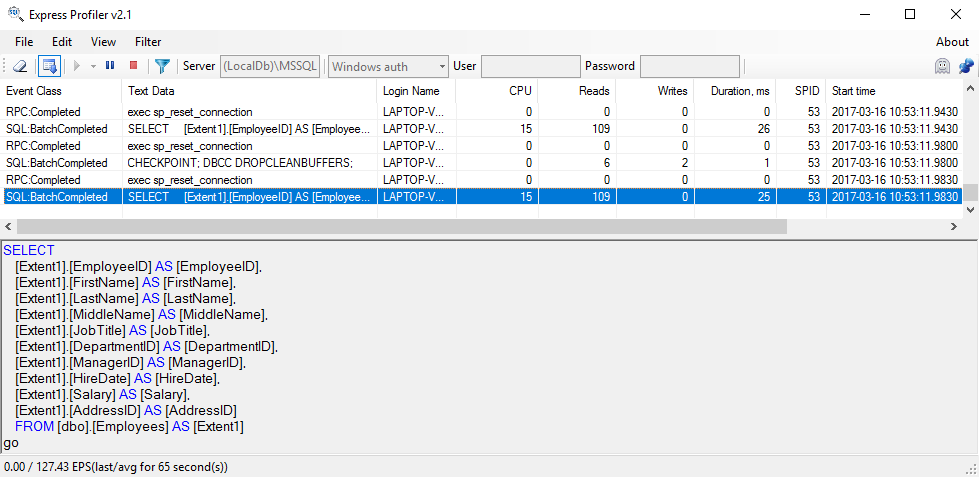
The result of executing the program should look like this:



Ok, now execute them again this time use the Profiler tool to see the generated SQL queries:



Remember to insert your own server name (replace: “**(LocalDB)\MSSQL**” if needed). Start tracing, then execute the program and see the result in the profiler:



Now do this for the “**eager**” solution.

Keep in mind that the test is ran at least 10 times, you can change that value to whatever you see fit. 😊

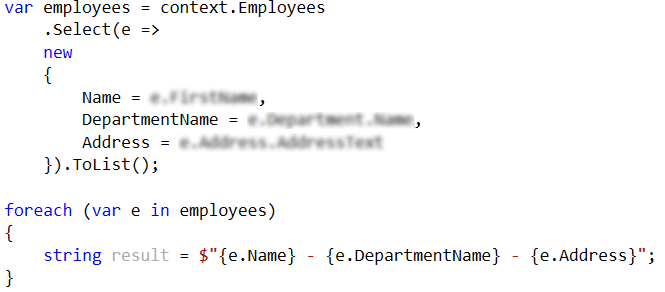
## Lazy vs Eager (Using Select)

Solve previous task again but this time use Select statements for both solutions. Execute them and check the generated SQL queries. Are they identical? Are they optimized compared to the ones in previous task? Is their performance equal?

Make separate methods and test it,

### Hints

Here is how the “lazy” solution might look like:



The eager one should be the same with a couple of Include()-s.😊

## Lazy vs Eager (Performance case #2)

Take the queries from the first task and modify them to collect data only for users which salary is below 3000.

How many queries made the “lazy” solution? Which one was faster?

## Lazy vs Eager (Performance case #3)

Show first name and department name of all employees participating in exactly one project. Use two queries: one with lazy loading and one with eager loading. Find the first SQL query generated (from the solution with lazy loading) and compare it to the query generated by the solution with eager loading. Which one is more optimal and why?

## Order by and ToList

Get all employees with their department ordered by department name and then by employee name. Again write two queries and in the first one use ToList followed by OrderBy and in the second – exchange their positions query (first use OrderBy then use ToList). Which one was faster? Why?

## Optimize Query

You are given a query which selects all managers whose employees live in town which name is starting with “B”. Your task is to optimize so it can pass for below 0.3 seconds:

|  |
| --- |
| var employees = context.Employees  .ToList()  .Where(e => e.Subordinates.Any(s => s.Address.Town.Name.StartsWith("B")))  .Distinct();  foreach (Employee e in employees)  {  string result = $"{e.FirstName}";  } |