# Lab: Team Builder

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

## Create Code First Model

Your task is to implement Team Builder console application. The application will consist of **users**, **teams, invitations** and **events**. Each event has several teams participating in it and each team has several users. Any **team member** or **creator** may send **invitation** to other user: let’s say we have two teams: **A** and **B** – and we are **members** of **A** but **not** of **B** – we can **send invitation** to other users **to join** team **A** **and** we **cannot** send invitations **for** team **B** because we are not simply part of it). The **invitation** **holds** information about the **team** which could be joined and who is the **invited user**, it also contains information if it is **active** or not.

The application consists of the following models:



**Here is information about each table:**

**Users**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Id | Integer from 0 to 2,147,483,647 | Unique table identificator |
| Username | String from 3 to 25 symbols | Unique, Required |
| FirstName | String up to 25 symbols |  |
| LastName | String up to 25 symbols |  |
| Password | String from 6 to 30 symbols | Should contain one digit and one uppercase letter, Required |
| Gender | Enumeration | Could be: '*Male*' or '*Female*' |
| Age | Integer from 0 to 2,147,483,647 |  |
| IsDeleted | Bool |  |

**Teams**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Id | Integer from 0 to 2,147,483,647 | Unique table identificator, Identity |
| Name | String up to 25 symbols | Unique, Required |
| Description | String up to 32 symbols |  |
| Acronym | String with exactly 3 symbols | Must be 3 symbols long, Required |

**Events**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Id | Integer from 0 to 2,147,483,647 | Unique table identificator, Identity |
| Name | String up to 25 symbols, Unicode | Required |
| Description | String up to 250 symbols, Unicode |  |
| StartDate | DateTime in format {dd/MM/yyyy HH:mm} |  |
| EndDate | DateTime in format {dd/MM/yyyy HH:mm} | Must be after StartDate |
| CreatorId | Integer from 0 to 2,147,483,647 | Relationship with table Users |

**Invitations**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| Id | Integer from 0 to 2,147,483,647 | Unique table identificator, Identity |
| InvitedUserId | Integer from 0 to 2,147,483,647 | Relationship with table Users |
| TeamId | Integer from 0 to 2,147,483,647 | Relationship with table Teams |
| IsActive | Boolean |  |

**UserTeams**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| UserId | Integer from 0 to 2,147,483,647 | Relationship with table Users, Unique table identificator |
| TeamId | Integer from 0 to 2,147,483,647 | Relationship with table Teams, Unique table identificator |

**TeamEvents**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** |
| TeamId | Integer from 0 to 2,147,483,647 | Relationship with table Teams, Unique table identificator |
| EventId | Integer from 0 to 2,147,483,647 | Relationship with table Events, Unique table identificator |

### Application Summary

**User** can **create** **event** or **team** – becoming their creator. **One** **event** may have **several teams** while **single team** can participate **in multiple events**. **Team** consists of **users** which also can be part of **other teams**.

Anyone from a team can **invite** people to join. Only the **creator** may **remove** **users** or to **disband** the whole **team**.

In order for a team to successfully participate in event – team’s creator must apply for it and later on to be approved by the creator of the event.

### Application Functionality

Team Builder contains the following functionality:

* **RegisterUser <username> <password> <repeat-password> <firstName> <lastName> <age> <gender>**Registers a new user.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] was registered successfully! | None |
| Username is not valid | Username [username] not valid! | ArgumentException |
| Password is not valid | Password [password] is not valid! | ArgumentException |
| Age is not in valid format or is non-positive number | Age not valid! | ArgumentException |
| Gender is not valid | Gender should be either “Male” or “Female”! | ArgumentException |
| Passwords do not match | Passwords do not match! | InvalidOperationException |
| Username is taken | Username [username] is already taken! | InvalidOperationException |
| There is currently logged in user | You should logout first! | InvalidOperationException |

\*Validation on first/last name is removed for the sake of simplicity – you are not obligated to perform any validation checks.

* **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 or user is deleted | Invalid username or password! | ArgumentException |
| There is currently logged in user | You should logout first! | InvalidOperationException |

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

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

* **DeleteUser**Deletes currently logged in user and then logs out.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] was deleted successfully! | None |
| There is no user logged in. | You should login first! | InvalidOperationException |

* **CreateEvent <name> <description> <startDate> <endDate>**Creates an event (**currently logged user is it’s creator)**. Keep in mind when parsing dates that there should be additional spaces between them.

**\***There might be several events with the same name. **Always pick the one with the latest start date!**

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Event [eventName] was created successfully! | None |
| Either start date or end date is in invalid format | Please insert the dates in format: [dd/MM/yyyy HH:mm]! | ArgumentException |
| Start date is after end date | Start date should be before end date. | ArgumentException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **CreateTeam <name> <acronym> <description>**Creates a team (currently logged user is it’s creator). Description is optional.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Team [team] successfully created! | None |
| Team does exist | Team [team] exists! | ArgumentException |
| Acronym is not valid | Acronym [acronym] not valid! | ArgumentException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **InviteToTeam <teamName> <username>**Sends an invite to the specified user to join given team. If the user is actually the creator of the team – add him/her directly!

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Team [teamName] invited [username]! | None |
| If the current user is not creator of the team nor part of it or user to invite is alredy a member | Not allowed! | InvalidOperationException |
| Either user or team does not exist | Team or user does not exist! | ArgumentException |
| There is an already active invite | Invite is already sent! | InvalidOperationException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **AcceptInvite <teamName>**Checks current user’s active invites and **accepts** the one from the team specified.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] joined team [teamName]! | None |
| Team does not exist | Team [teamName] not found! | ArgumentException |
| There is no invite from that team | Invite from [teamName] is not found! | ArgumentException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **DeclineInvite <teamName>**Checks current user’s active invites and **declines** the one from the team specified.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Invite from [teamName] declined. | None |
| *\*Look in above command to see other cases.* | | |

* **KickMember <teamName> <username>**Removes specified user member from given team. Only the creator of the team can kick other members.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | User [username] was kicked from [teamName]! | None |
| Team does not exist | Team [teamName] not found! | ArgumentException |
| User does not exist | User [username] not found! | ArgumentException |
| User is not a member in team | User [username] is not a member in [teamName]! | ArgumentException |
| Current user is not creator of the team | Not allowed! | InvalidOperationException |
| User to be kicked is the creator of the team | Command not allowed. Use DisbandTeam instead. | InvalidOperationException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **Disband <teamName>**Deletes given team. Allowed for the team’s creator only.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | [teamName] has disbanded! | None |
| Team does not exist | Team [teamName] not found! | ArgumentException |
| Current user is not creator of the team | Not allowed! | InvalidOperationException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **AddTeamTo <eventName> <teamName>**Adds given team for event specified. If there are more than one events with same name pick the latest start date.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | Team [teamName] added for [eventName]! | None |
| Event does not exist | Event [eventName] not found! | ArgumentException |
| Team does not exist | Team [teamName] not found! | ArgumentException |
| Current user is not creator of the event | Not allowed! | InvalidOperationException |
| Team is already added to event | Cannot add same team twice! | InvalidOperationException |
| There is no logged in user | You should login first! | InvalidOperationException |

* **ShowEvent <eventName>**Shows details for given event.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | [eventName] [eventStartDate] [eventEndDate]  [description]  Teams:  -[teamName]  … | None |
| Event does not exist | Event [eventName] not found! | ArgumentException |

* **ShowTeam <teamName>**Show details about given team.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | [teamName] [teamAcronym]  Members:  --[username1]  …  --[usernameN] | None |
| Team does not exist | Team [teamName] not found! | ArgumentException |

* **Exit**Exits application.

|  |  |  |
| --- | --- | --- |
| **Case** | **Message** | **Exception** |
| Success | *None* | None |

If a command’s name is **different** from any of the commands above, throw a NotSupportedException with message: “Command [commandName] not valid!”.

If format of the command is not valid (invalid number or arguments) throw FormatException with message: “Invalid arguments count!”

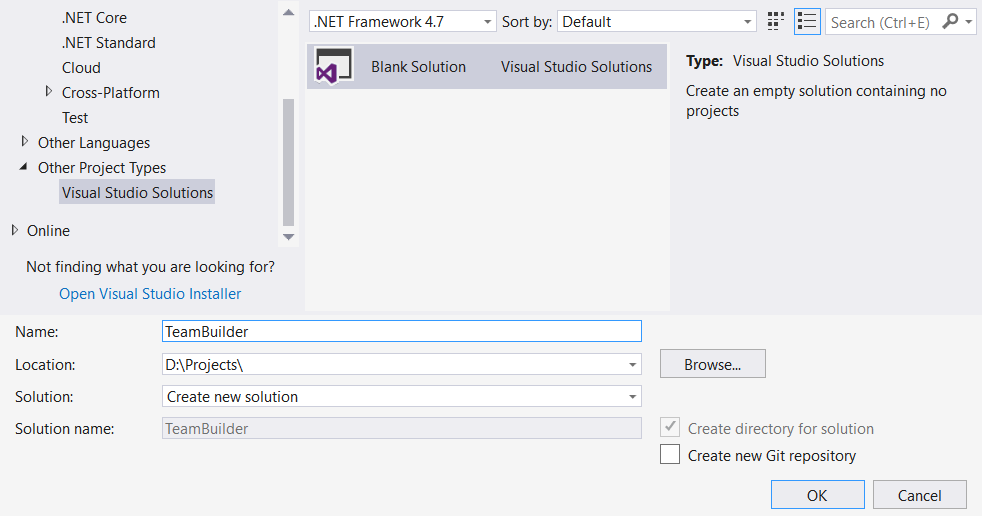
### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RegisterUser johny j0hny j0hny John Smith 22 Male  RegisterUser johny Inval1d Inval1d John Smith 22 Male  Login johny Invalid  Login johny Inval1d  Logout  Logout j0hny  Login johny Inval1d  DeleteUser  Logout  Login johny Inval1d  Exit | Password j0hny not valid!  User johny was registered successfully!  Invalid username or password!  User johny successfully logged in!  User johny successfully logged out!  Invalid arguments count!  User johny successfully logged in!  User johny was deleted successfully!  You should login first!  Invalid username or password! |
| RegisterUser daniel Dan123 Dan123 Daniel Trevor 22 MMale  RegisterUser daniel Dan123 Dan123 Daniel Trevor 22 Male  Login daniel Dan123  CreateEvent TEDexSofia Inovation 01-01-2012 12:00 02-01-2012 22:00  CreateEvent TEDexSofia Inovation 01/01/2012 12:00 02/01/2012 22:00  CreateTeam Band BND  CreateTeam BitColns BCS  AddTeamTo TEDexSofia Band  AddTeamTo TEDexSofia BCS  AddTeamTo TEDexSofia Band  AddTeamTo TEDexSofia BitColns  ShowEvent TEDexSofia  Exit | Gender should be either “Male” or “Female”!  User daniel was registered successfully!  User daniel successfully logged in!  Please insert the dates in format: [dd/MM/yyyy HH:mm]!  Event TEDexSofia was created successfully!  Team Band successfully created!  Team BitColns successfully created!  Team Band added for TEDexSofia!  Team BCS not found!  Cannot add same team twice!  Team BitColns added for TEDexSofia!  TEDexSofia 01/01/2012 12:00 02/01/2012 22:00  Inovation  Teams:  -Band  -BitColns |
| RegisterUser gordon Ham123 Ham123 Gordon Hamilton -2 Male  RegisterUser gordon Ham123 Ham123 Gordon Hamilton 32 Male  RegisterUser terrydom Terry123 Terry123 Terry Molina 32 Female  Login gordon Ham123  CreateEvent CrackIT ITHardware 22/10/2013 12:00 22/10/2013 22:00  CreateEvent CrackIT ITHard 13/08/2015 12:00 15/08/2015 22:00  CreateTeam Crackers CKS  CreateTeam Balder BLD  InviteToTeam Crackers terry-dom  InviteToTeam Crackers terrydom  InviteToTeam Balder terrydom  Logout  Login terrydom Terry123  AcceptInvite CrackIT  AcceptInvite Crackers  DeclineInvite Balder  Disband Balder  Logout  Login gordon Ham123  ShowTeam Balder  Disband Balder  ShowTeam Crackers  KickMember Crackers terry-dom  KickMember Crackers terrydom  AddTeamTo CrackIT Crackers  ShowEvent CrackIT  Exit | Age not valid!  User gordon was registered successfully!  User terrydom was registered successfully!  User gordon successfully logged in!  Event CrackIT was created successfully!  Event CrackIT was created successfully!  Team Crackers successfully created!  Team Balder successfully created!  Team or user does not exist!  Team Crackers invited terrydom!  Team Balder invited terrydom!  User gordon successfully logged out!  User terrydom successfully logged in!  Team CrackIT not found!  User terrydom joined team Crackers!  Invite from Balder declined.  Not allowed!  User terrydom successfully logged out!  User gordon successfully logged in!  Balder BLD  Members:  Balder has disbanded!  Crackers CKS  Members:  --terrydom  User terry-dom not found!  User terrydom was kicked from Crackers!  Team Crackers added for CrackIT!  CrackIT 13/08/2015 12:00 15/08/2015 22:00  ITHard  Teams:  -Crackers |

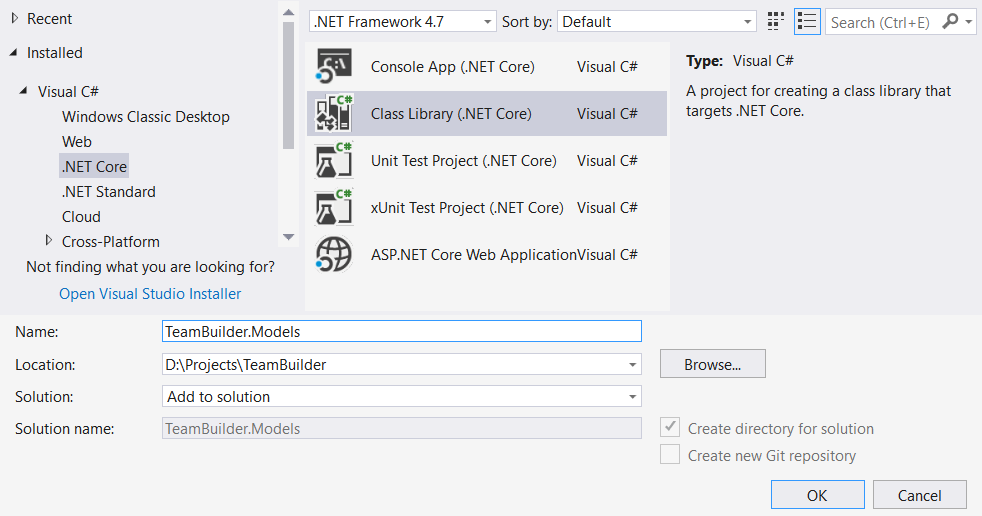
## Configure Models and Relations

### Create Entity Classes

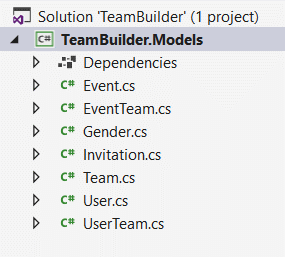
Let’s start with creating a simple blank solution:



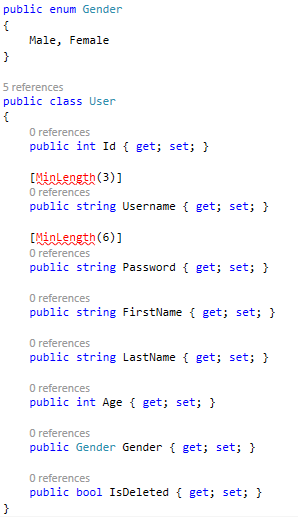
Now let’s start with creating project for our models. Create Class Library project named TeamBuilder.Models:



Inside the models’ project create **empty** classes for every independent entity (including **join entities**). In the end, you should have something like this:

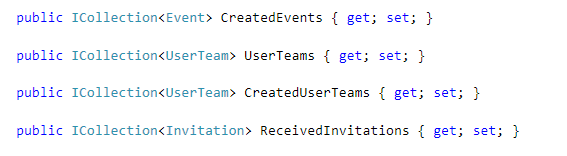


Let’s start with defining properties for our models, the first one is the User:



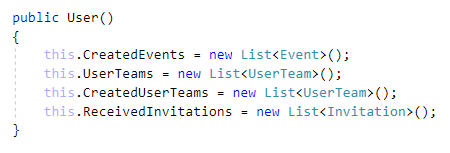
However for the MinLength attribute we will have to reference the System.ComponentModel.DataAnnotations package for our project.

We will use EntityTypeConfiguration class for any relation configuration etc. Talking about relations? Let’s add all navigation properties for the **User** entity:

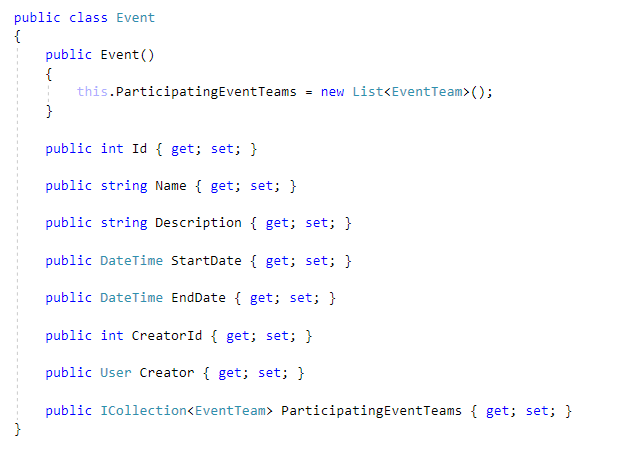


As you can see we have different collections (some with the same model).

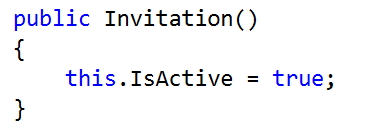
One last step for the User model – **initialize** all the above **collections** in the constructor:



Now we are done with our User, let’s continue with our Event:



So what we left to configure is the Invitation and the Team models. Well it’s up to you to do it but here is small hint on the Invitation model – make Invitation **initially active** (set this.IsActive = true):



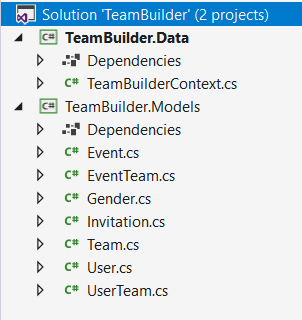
The Team will have reference to it’s **creator**, **members**, **events** which the team is participating and collection of **invitations** send from any member (or creator) of the team.

### Set up Entity Relations

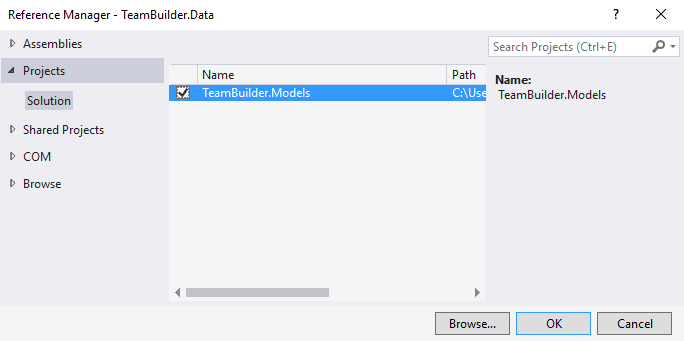
Now let’s move on the next part – creating our data model alongside with configuration of the relations.

Create a new Class Library project called TeamBuilder.Data. In it, delete the generated class ”Class1.cs” and add a new DbContext **class**. Name it TeamBuilderContext. Make sure to install Microsoft.EntityFrameworkCore.SqlServer package beforehand.

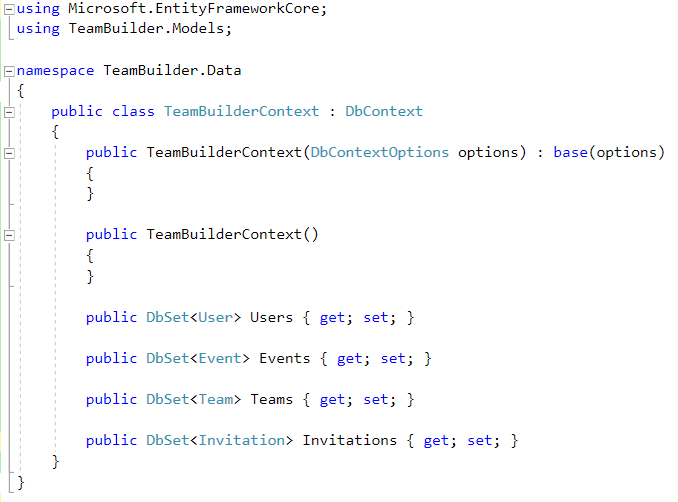
When you’re done, your **project structure** should look like this:



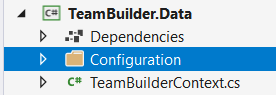
Make sure to reference TeamBuilder.Models project to TeamBuilder.Data:



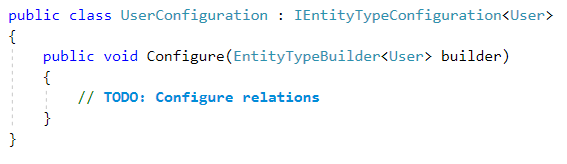
Now go to TeamBuilderContext.cs file and reference all models that we have already created:



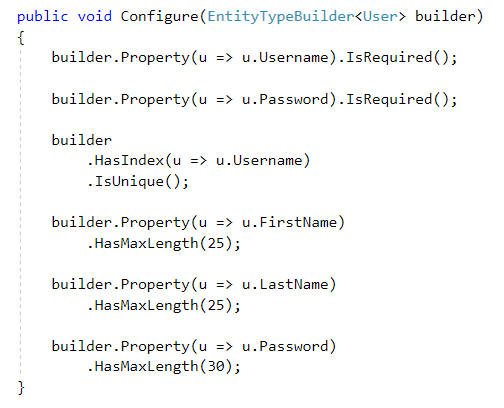
Now in the TeamBuilder.Data project add new folder named **Configuration** – in it we will put all model configurations:



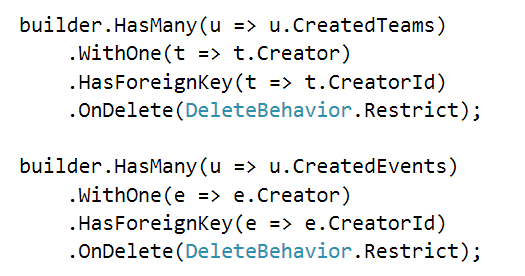
Now in it, let’s create a UserConfiguration class – make that class inherit (ouch) EntityTypeConfiguration<User>:



Now let’s configure the simple properties of the User model:

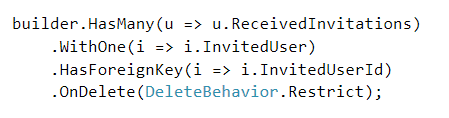


Now let’s start setting up the **relations**. First begin with created **Teams/Events**:

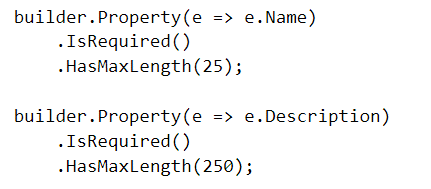


This will set up the one-to-many relation between **User-Team** and **User-Event**.

**User-Invitation** relation:

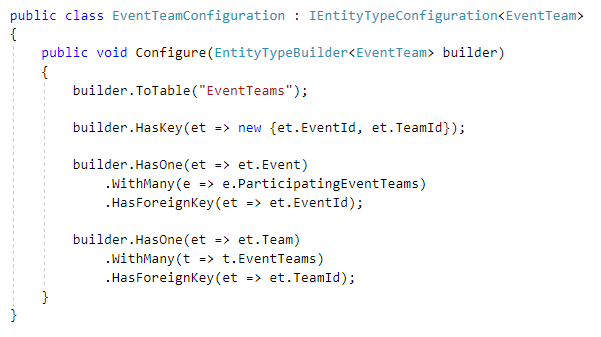


We will need one more configuration class before moving on – it will be EventConfiguration:

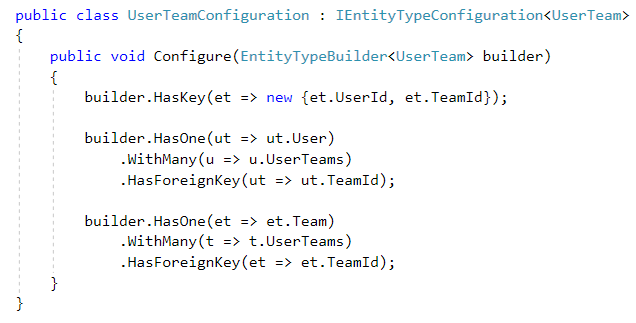


Before we move on to configuring these relations in the OnModelCreating method, let’s create **two more classes** which will configure our **many-to-many** relations:

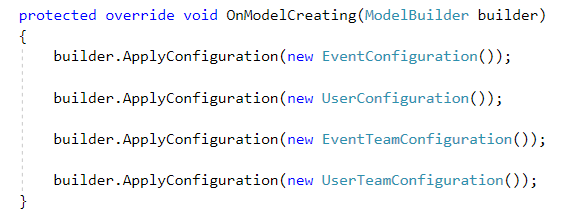
EventTeamConfiguration.cs:



UserTeamConfiguration.cs:



Now, let’s go to TeamBuilderContext.cs, override the **OnModelCreating** method and include all the configurations in the model builder:



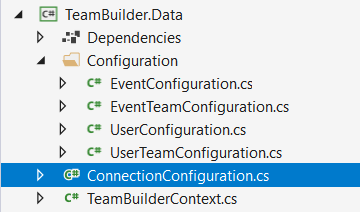
We are done with setting up the relations – what’s left is to configure the other models and the additional constraints (like .IsRequired(), .HasMaxLength() and so on). This part is left to you.😊

Reminder: add those configurations in the **ModelBuilder** as well.

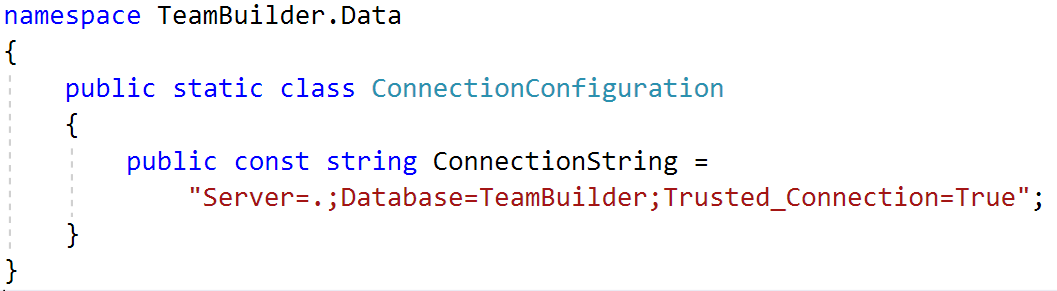
### Configure Connection String

We’ve set up almost everything. All that’s left is to tell Entity Framework Core which **SQL Server instance** and **database** to target.

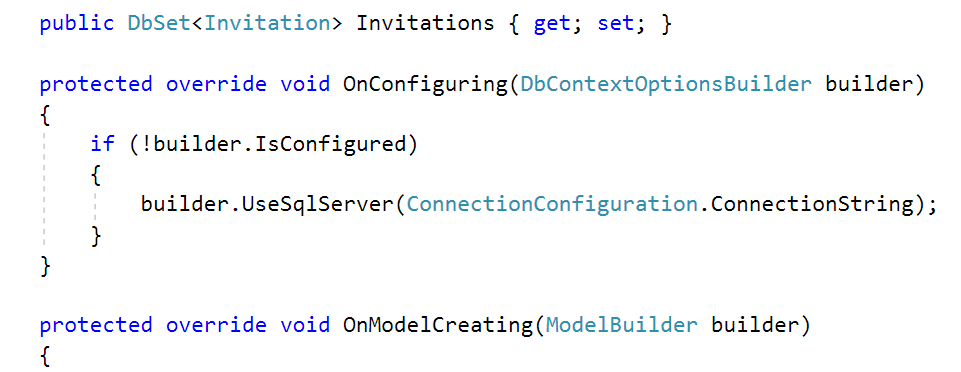
Create a new class, called ConnectionConfiguration.cs:



Inside it, define your connection string:

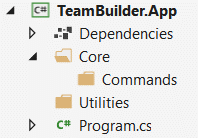


Then, in your TeamBuilderContext class, override the OnConfiguring method:



## Define Application Structure

Great, we have created the models and their relations. Next, we have to start implementing the console application. Create new Console Application project named TeamBuilder.App. And it will have the following hierarchy:



We will use the so called **Command Pattern**, used by some kind of **Engine** class. Every command may use **helper methods** and **classes**.

Rename **Program.cs** to **Application.cs**.

Reference the TeamBuilder.App to Teambuilder.Data and TeamBuilder.Models as well.

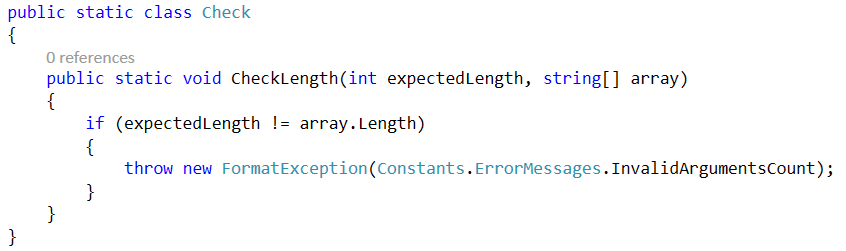
## Implement Utilities

Before we go to implementing the **core logic** of the application, let’s write some **helper methods** which will help us later.

Create a **public static class Constants**. Inside of it – there will be several **constants** for performing **validation checks** or even **error messages**. For the sake of simplicity, you won’t need to implement this one yourself, it’s ready to copy-paste:

|  |
| --- |
| public static class Constants  {  public const int MinUsernameLength = 3;  public const int MaxUsernameLength = 25;  public const int MaxFirstNameLength = 25;  public const int MaxLastNameLength = 25;  public const int MinPasswordLength = 6;  public const int MaxPasswordLength = 30;  public const string DateTimeFormat = "dd/MM/yyyy HH:mm";  public static class ErrorMessages  {  // Common error messages.  public const string InvalidArgumentsCount = "Invalid arguments count!";  public const string LogoutFirst = "You should logout first!";  public const string LoginFirst = "You should login first!";  public const string TeamOrUserNotExist = "Team or user does not exist!";  public const string InviteIsAlreadySent = "Invite is already sent!";  public const string NotAllowed = "Not allowed!";  public const string TeamNotFound = "Team {0} not found!";  public const string UserNotFound = "User {0} not found!";  public const string EventNotFound = "Event {0} not found!";  public const string InviteNotFound = "Invite from {0} is not found!";  public const string NotPartOfTeam = "User {0} is not a member in {1}!";  public const string CommandNotAllowed = "Command not allowed. Use {0} instead.";  public const string CannotAddSameTeamTwice = "Cannot add same team twice!";  // User error messages.  public const string UsernameNotValid = "Username {0} not valid!";  public const string PasswordNotValid = "Password {0} not valid!";  public const string PasswordDoesNotMatch = "Passwords do not match!";  public const string AgeNotValid = "Age not valid!";  public const string GenderNotValid = "Gender should be either “Male” or “Female”!";  public const string UsernameIsTaken = "Username {0} is already taken!";  public const string UserOrPasswordIsInvalid = "Invalid username or password!";  public const string InvalidDateFormat =  "Please insert the dates in format: [dd/MM/yyyy HH:mm]!";  // Team error messages.  public const string InvalidAcronym = "Acronym {0} not valid!";  public const string TeamExists = "Team {0} exists!";  }  } |

Now add new class called **Check.cs**. It will have one simple method in it which will check if array’s length is equal to expected amount:



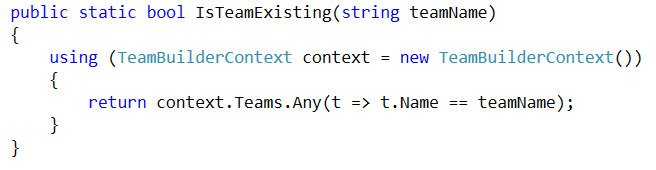
You may add other checker methods here if you want (make sure they are **static** though).

One last helper named CommandHelper before we continue with our core logic. The helper class will make queries to the database checking whether certain things **exist** (for example - check if a **town exists** by a given **town name** and so on).

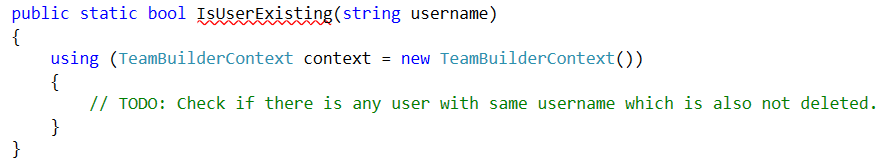
It will contain the following methods:

* **bool** **IsTeamExisting**(**string** **teamName**)
* **bool IsUserExisting**(**string** **username**)
* **bool IsInviteExisting**(**string** **teamName**, **User** **user**)
* **bool IsUserCreatorOfTeam**(**string** **teamName**, **User** **user**)
* **bool IsUserCreatorOfEvent**(**string** **eventName**, **User** **user)**
* **bool IsMemberOfTeam**(**string** **teamName**, **string** **username**)
* **bool IsEventExisting**(**string** **eventName**)

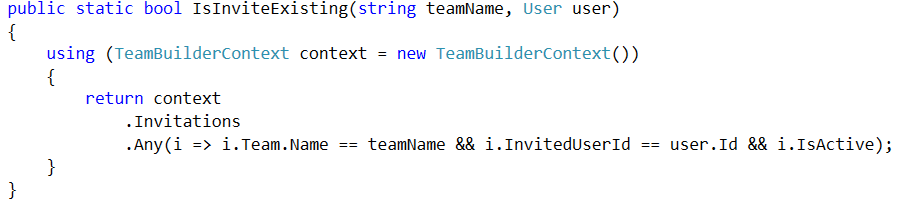
Let’s implement those one by one:



Here is how the IsUserExisting() should be implemented:

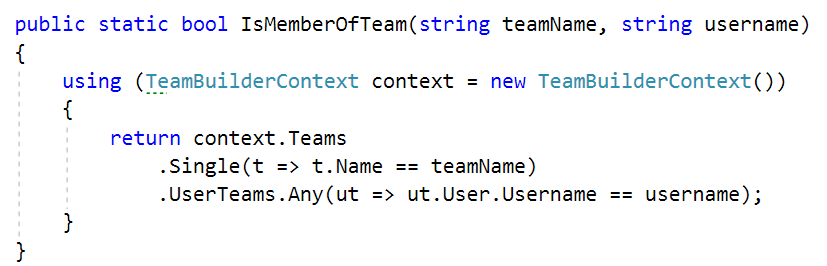


And here’s how the IsInviteExisting() method should be implemented:



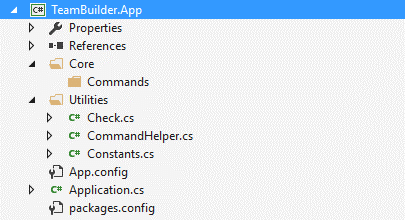
\* Note that we are using the **user’s** **Id**, keep that in mind when passing the **user** to that method (they should already be loaded from the database).

Here’s how to implement the IsMemberOfTeam() method:



It is your turn to implement IsEventExisting(), IsUserCreatorOfEvent()and IsUserCreatorOfTeam().

After all you should have three classes in your Utilities folder:

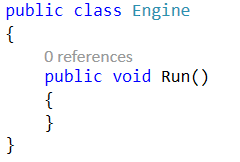


We are [done](https://media.giphy.com/media/1bHdnX1QMeQTe/giphy.gif) with this section.

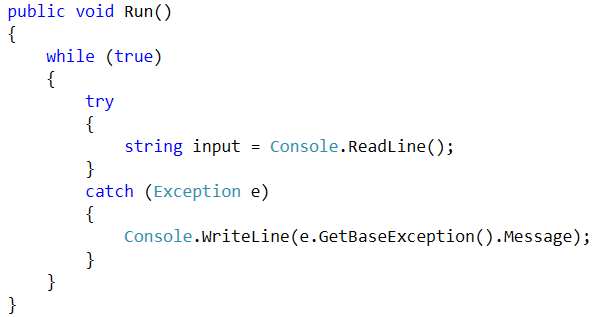
## **Implement Core Structure**

Our application will rely on three major elements: Engine, CommandDispatcher and Commands bundle classes.

First we will take on the Engine class. In the **Core** folder add new Engine.cs class with simple Run() method:



Inside the run method create new **endless loop** inside that loop put **try-catch block**:

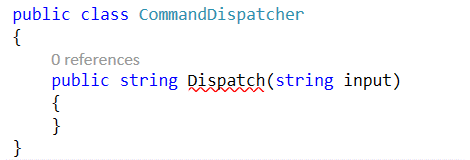


Note that inside the exception we get the base exception (the initial exception) and we print it on the console.

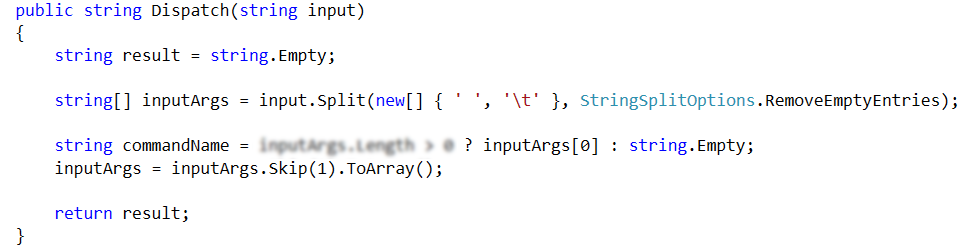
We have a neat application here, but it **does not do anything**. We have to make it **parse input** from the **console**. Then, based on the **input**, find a **specific command** and **execute** it. The **result** of the command should be **printed back** on the **console**.

For this part, we will need to make a CommandDispatcher class. Its task is to **parse the input**, **find** the specified **command** (**if any**) and **execute** that command while **giving the command the input** from the user.

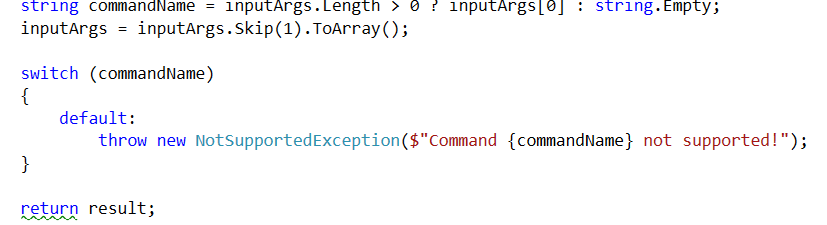
Create a CommandDispatcher class in **Core** folder. **Make** one **method** which **receives string** and **returns string** called Dispatch():



Now **split** the **input** (split by any whitespace character), **take** the **first** **argument** as the name of the command and create new array (or **overwrite** the old one) which will have all other arguments from the input except the **name** of the command. Something like this:

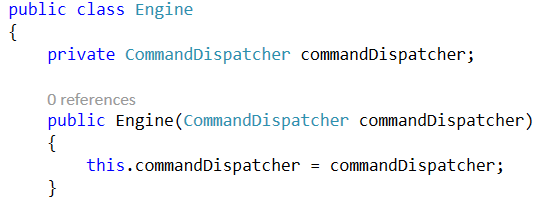


Now, write a **switch case** on the **command name** and set the default behavior to **throw an exception**:

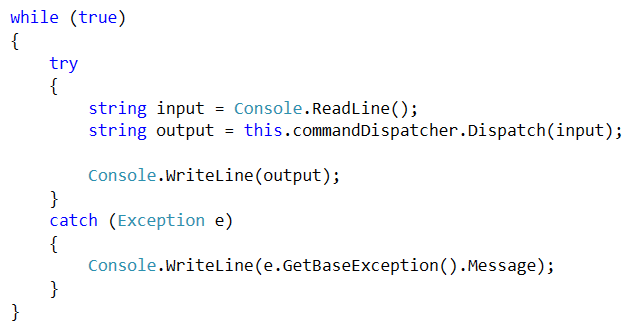


Now that we have configured the basic logic turn back to the **Engine** class.

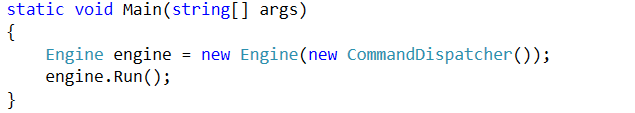
Add new private field of CommandDispatcher which must be initialized in the constructor:



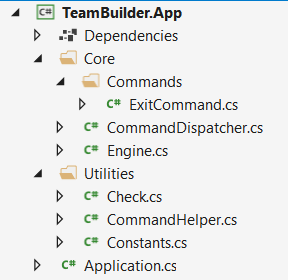
Now use it in the Run() method:

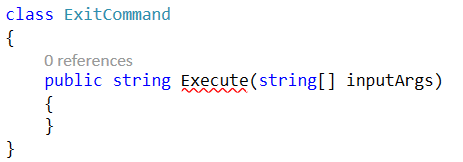


We are done with our Engine class for now. Let’s instantiate in our Application:

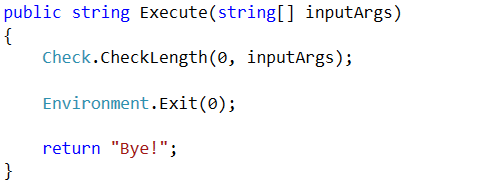


Let’s create one simple command. Inside the **Commands** folder create new ExitCommand class with   
Execute(string[] inputArgs) method which returns string:

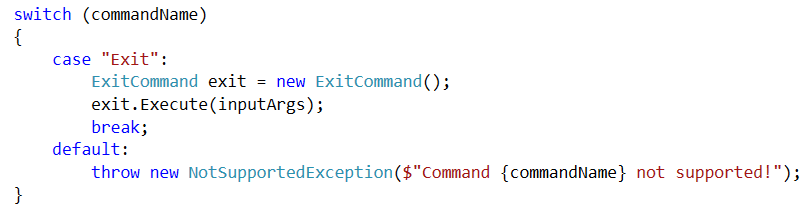




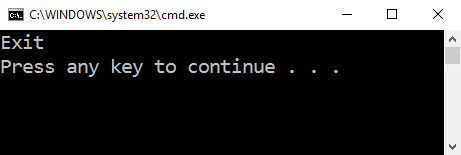
The function must check if there are any input arguments and throw exception if there are or else to exit the program:



One last thing before we move on. Include that command in the CommandDispatcher:



Set current project as start up project, check for any errors and if there are not – start the program.

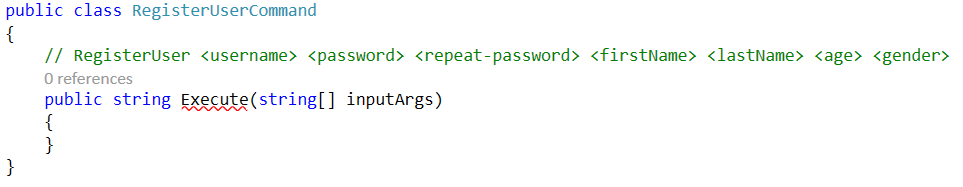


If everything is fine, the result should be like the one above.

## **Implement Base Logic**

In this part we will take on implementing: RegisterUser, Login, Logout and DeleteUser.

First things first – create a new command class, named RegisterUserCommand. Again, make an Execute method just like the one we created in the ExitCommand:



We have several cases here so go back to the [Application Functionality](#_Application_Functionality) section and see how the command should behave.

Now after that we know what the command is expected to do is time to put some code to work:

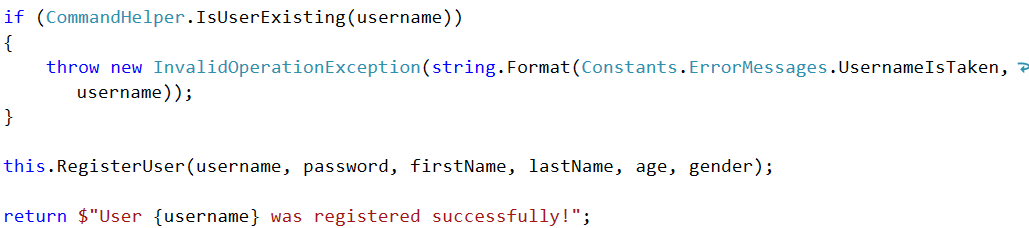


Note that every message for the exception is taken from our static helper class.

There are some more validations on the input:

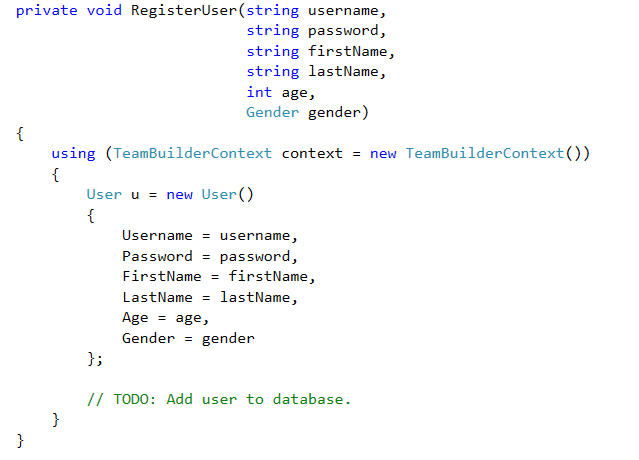


We are almost done with validation, we have to check if the given username is taken and if not to register the new user:

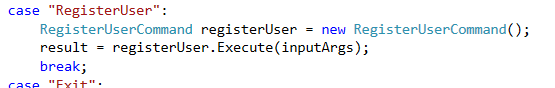


\*Note that we are using the CommandHelper class to make the check.

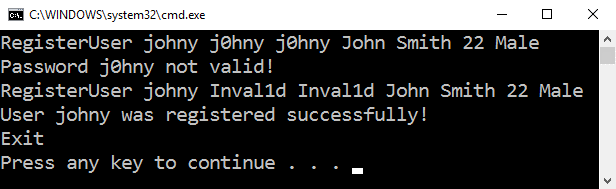
Here is how we can the RegisterUser method may look like:



After we are done come back to the command dispatcher and add new case:



Start the application and run sample register user command(take one from the [Examples](#_Examples) section). Something like this should happen:



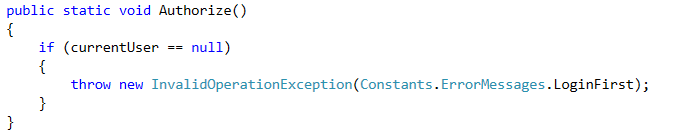
See if the user is really saved in the database. You can play around with the corner cases if you want.

The next two commands to implement (Login and Logout) require an additional helper class which will have the logic behind authenticating a user in our application.

In the Core folder, add a new class - AuthenticationManager. It consists of the following functionallities:

* void Login(User user) – saves given user as logged user until logout or exit of the application
* void Logout() – logs out currently logged in user, if there is none should throw exception (use the method below)
* void Authorize() – throws InvalidOperationException if there is no logged in user
* bool IsAuthenticated() – returns true if there is logged in user else returns false
* User GetCurrentUser() – gets currently logged in user if there is not throws exception

Let’s take a look at how the Authorize() method might look like:

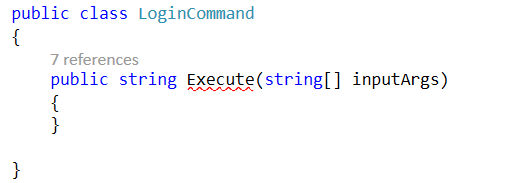


Where current user is private static **field**:



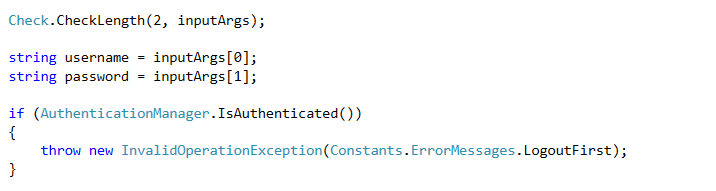
Other methods are left to you to implement them.

Now we are done with our AuthenticationManager (sort of). Let’s implement Login and Logout. Create LoginCommand first:

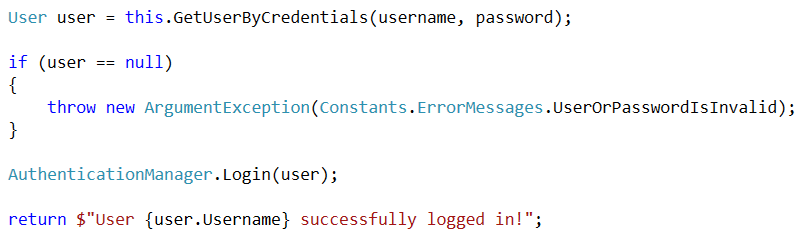


Again go to [Application Functionality](#_Application_Functionality) section and see the cases defined there.

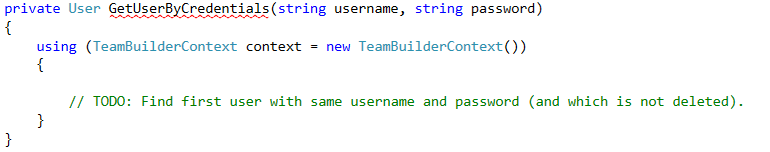
First check given arguments count then if there is currently logged in user:



If there is no logged in user try to find one based on the input given. If you don’t find one return null. Something like this:

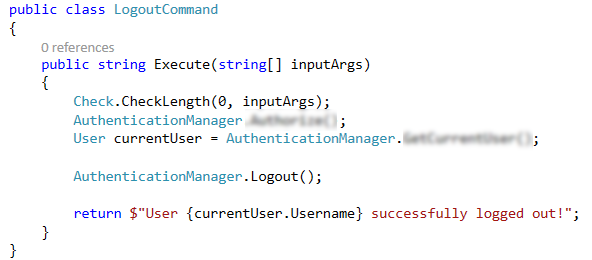


If you wonder what’s behind GetUserByCredentials:

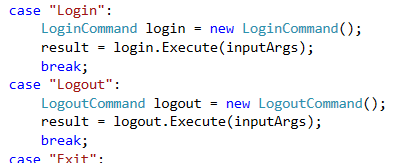


\*Hint: use .FirstOrDefault()

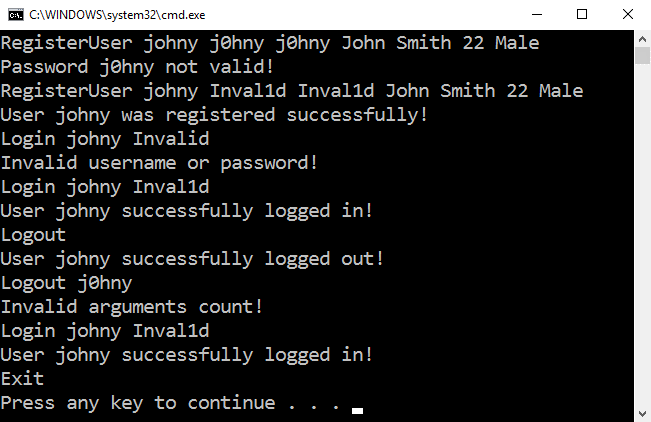
So we finished implementing the login command, now let’s create LogoutCommand:



Now go back to the CommandDispatcher and cases for login and logout:

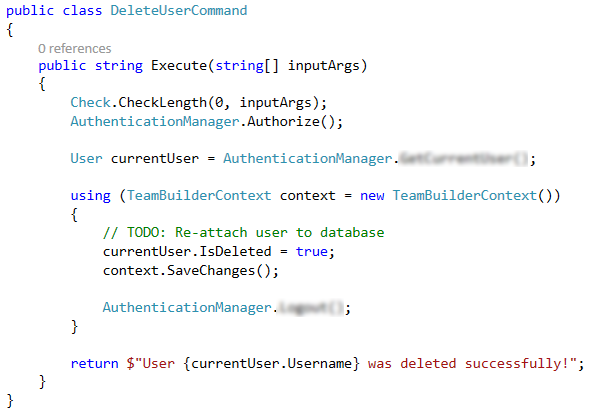


Start the application again and try to test it with some of the [Examples](#_Examples):



If everything is all right something like this should be displayed.

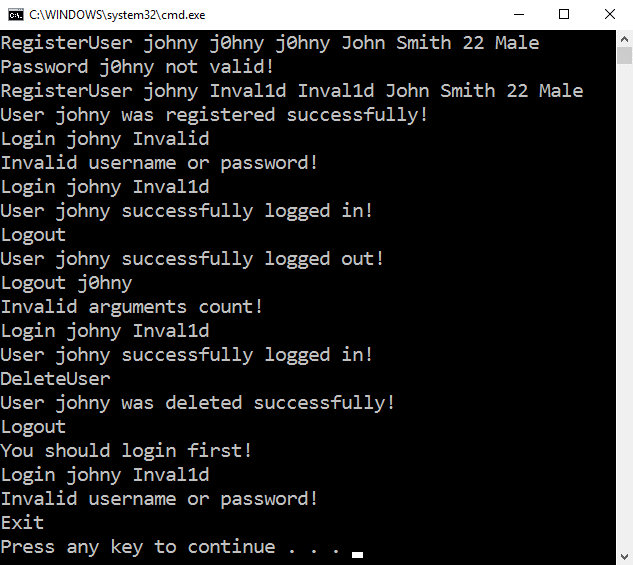
One last thing before we finish up this section – the DeleteUserCommand:



Now for one last [time](http://cdn.niketalk.com/5/50/900x900px-LL-506d59bf_not-this-shit-again_zpsb4456328.jpeg) go to the CommandDispatcher and add **case** for this command.

After doing that you are fully capable of testing the first example given in the [Examples](#_Examples) section.

Run the program, insert the input and something like this should happen:



And for the next part all you have to do is… relax. Chill buddy, grab a beer, talk to somebody – you don’t need that application to build a social team around you 😉.