# Workshop: Forum

## Overview

In the workshop we shall create the same Forum console application, but this time we are going to follow the SOLID principles and you will see how to do some things easily with Reflection. Again, you are provided with some source code, this time with all of the projects that you’ll need.

## Get familiar with the Project

Introduction

In this part you should look around the project that you are kindly provided. ☺

### StartUp

The StartUp.cs’s main job is just to instantiate a MenuController for the engine to use and run the **Engine**... for now. There is another method ConfigureServices but we’ll get to it later.

### Engine

The Engine.cs has a private IMenuConttroller that we are injecting trough the constructor and a **Run** method which is basically the while loop that is waiting for user input.

### ForumViewEngine

This class holds the functionality about displaying our interface on the console.

### MenuController

Holds the functionality of navigating through menus.

### Contracts

Those are all of the interfaces you are about to use in your application, so you don’t have to break a sweat defining them.

### Factories

Some factories you are going to use, partly implemented…

### Menus

These are the different menus all of them holding information about their views and the functionality to execute commands, change pages, etc. All of them inherit from the abstract class Menu.

### Models

The models that the app is using. The interesting one is **Session** that you’re about to implement.

### Services

Here we are going to put the services which connect our business logic to our data.

## Create Factories

Introduction

In this part we are going to implement a CommandFactory we’re going to use in the menus and a MenuFactory we’re going to use in commands. Fun, right? Let’s get to it.

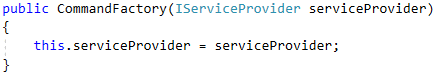
### CommandFactory

Now, when we open CommandFactory.cs we can see that it only implements the ICommandFactory interface and throws a not implemented exception.

First of all, we shall create a private field of type IServiceProvider which is the interface behind ServiceProvider, which, on the other hand is the built-in implementation of the [Service Locator Pattern](https://msdn.microsoft.com/en-us/library/ff648968.aspx?f=255&MSPPError=-2147217396).

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_18-06-06.png

Now, you might get the red squiggly line but just add the **using** **System** statement and it will be ok. Next, we need to inject our service provider trough the constructor. As you might have thought, the constructor should look like this:

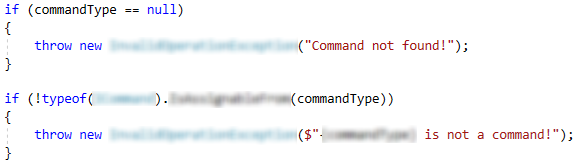


The CreateCommand method should implement finding the **command** with the **given** **name** from the **executing** **assembly**, perform **validations**, get the **parameters** **needed** for the command, and finally **create an** **instance** and **return** it.

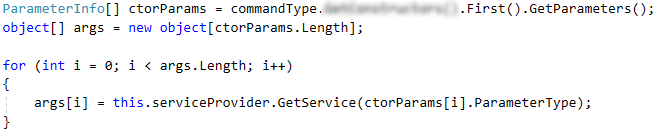
First, we need to fetch the executing assembly and find in it a type with the given command name plus “Command” suffix. The implementation should look like this:

devenv_2018-04-09_18-26-05

Next you need to perform a null check that throws an InvalidOperationException with “Command not found!” message and you have to check if the type inherits ICommand and throw exception with “{commandType} is not a command!” message if it doesn’t. The implementation will look a lot like this:



The next step is to get the **parameters** that the **constructor** of the command **needs** and create an **array** of **objects** we are going to take **from** the **service** **provider**. It should look like this:



Finally, you have to create an instance of that type with the given argument and return it.

devenv_2018-04-09_18-54-31

### MenuFactory

Since you’ve got the CommandFactory done create a new class called MenuFactory.cs implement IMenuFactory and implement it by yourself. The exception messages you have to use are: **“**Menu not found!**”** and **“**{menuType} is not a menu!**”**. Good luck!

## Create Session

Introduction

If you take a close look at the MenuController (Main functionality, remember?) there is a **private** **field** called **session** which is not used. That is because it’s not yet implemented. That **Session** is located in the **Models** folder and its job is to keep information about whether the user is logged in or not, current menu, history and some corresponding functionality.

### Private Fields

The Session should contain a **private** **user** of type **User** and **history** which implementation is going to be a **Stack** of IMenu. Looking like this:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_19-37-46.png

### Constructor

Create an empty constructor that only initializes the history field that you’ve just defined.

### Properties

Since we have to provide Username and UserId where **user** might be **null** the “**null-coalescing**” and the “**null-conditional**” operators will come in handy.

The **Username** property should look like this:

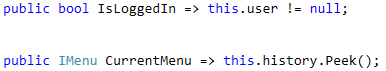
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_19-50-56.png

Then there is the UserId property:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_19-51-56.png

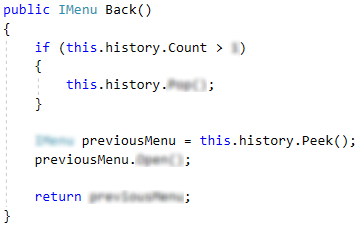
If you are having trouble with the syntax above read [this](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-conditional-operator).

Here are the LoggedIn check and CurrentMenu:



### Back

The **Back()** method’s job is simple: If there are more than one **Menus** in history, **pops** the **last** **one**, then **peeks** the **previous**, calls its **Open()** method and returns the **menu**. The implementation should look like this:



### LogIn

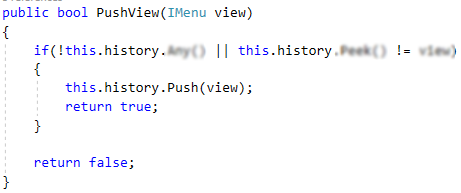
Just assign the given parameter to the corresponding field.

### LogOut

Just assign the field to null.

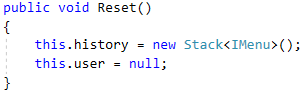
### PushView

As the name describes this **method** is going to **push** a **new** **menu** to the **history** **stack**. The **conditions** under which you have to push are whether **history** is **empty** or the **last** **menu** is **different** from the one you are about to **push**. In other words, you have to **make** **sure** that you **don’t** **push** **two** **same** **menus** in a **row**. Finally, you should return **true** if the push succeeded and **false** if not. The implementation:



### Reset

The reset method surprisingly resets the session setting the user to null and assigning new Stack of IMenu to history. Like so:



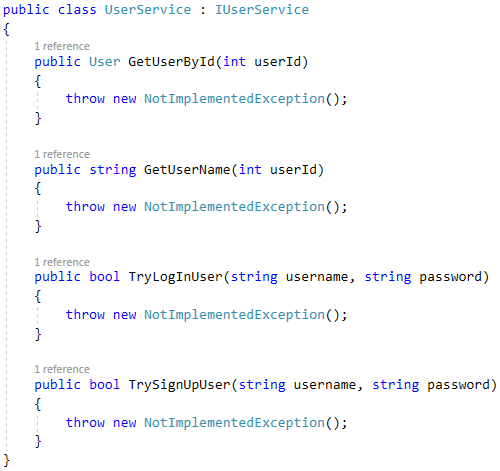
## Create Services

Introduction

In this part we are going to just define the UserService and PostService and their constructors and we will keep coming back to them to implement the functionality we need.

### UserService

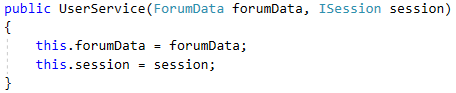
In the Services folder create a UserService class that implements IUserService and leave the methods throwing NotImplementedException for now. The class should look something like this:



Next, you need to define two private fields:

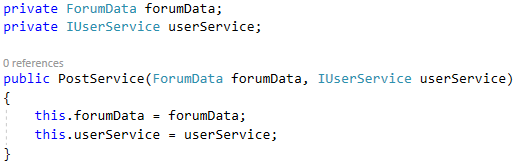
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_21-20-31.png

Next, you need to inject those two trough the constructor and it should look like this:



### PostService

In the same folder create a PostService class that implements IPostService interface. Leave the methods just as you did above. Create the private fields and inject them, like this:



That’s all for now, we will be back implementing methods shortly.

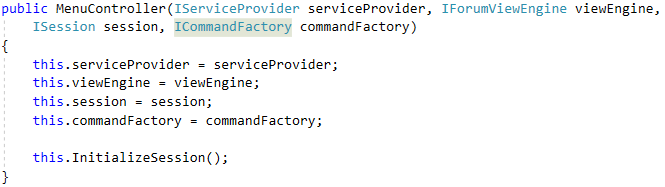
## MenuController

Introduction

In this part we are going to inject the unused dependencies and set the CurrentMenu property point to the session’s CurrentMenu.

### Inject the Dependencies

You just need to inject the dependencies and call the InitializeSession method, like so:



### Set CurrentSession

Just delete the current property and uncomment the second commented line.

## Create ServicePovider

Introduction

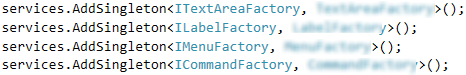
In this part we are going to implement a **service** **provider** that is going to provide us with the **objects** (**services**) we **require**.

### ConfigureServices

Go to StartUp.cs. There you have a ConfigureServices method that returns an IServiceProvider. First, we are going to instantiate a ServiceCollection, like so:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_23-11-59.png

Then we are going to add all of the **factories** as **singletons**:



Next, we will add ForumData as a **singleton** and the two services as **transients**:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_23-22-24.png

Finally, we have to add a **Session**, ForumViewEngine and MenuController as singletons:

devenv_2018-04-09_23-26-20

And a ForumConsoleReader as transient:

devenv_2018-04-09_23-28-47

After all that is done, just build the service provider and return it:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_23-39-14.png

Now, go to the Main method, instantiate an IServiceProvider and get a MenuController from it:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-09_23-43-52.png

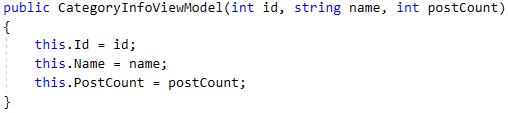
## Create ViewModels

Introduction

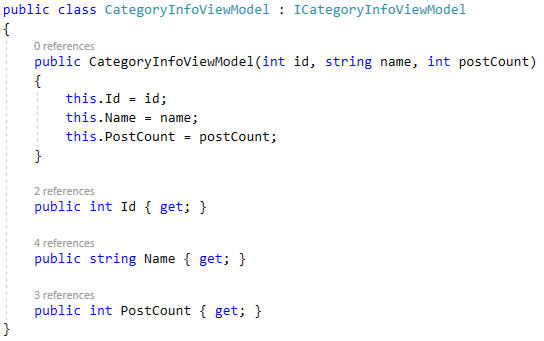
In this part we are about to create **view** **models** in order to work with **information** easily and not have to use the whole data models, so let’s get to it.

### CategoriesViewModel

In the ViewModels folder create a class called CategoryInfoViewModel which implements ICategoryInfoViewModel. Add a constructor to assign the properties, like so:



The whole thing should look like this:



### ContentViewModel

This class is responsible for storing and wrapping our content.

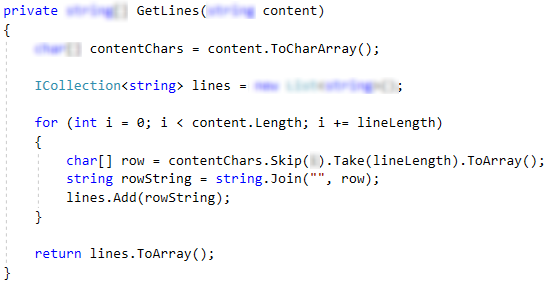
First create a class in the ViewModels folder called ContentViewModel. That view model should know how long the text line length will be, so we are going to add a private constant integer that will hold the number **37** like so:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_07-57-25.png

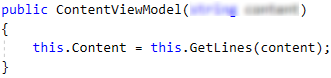
Next, we are going to generate an automatic **property** with just a **getter** and it’s going to represent text lines as an array of strings:



After that there is method we need to implement called GetLines. The parameter it takes is a **single** **string** and its return type is an array of strings. As you might have guessed it **converts** a whole **string** into **lines**:



The last piece of code you need to add is a public constructor that takes string content as a parameter and simply assigns the result of the method above to the Content property:



### PostInfoViewModel

Similar to CategoriesViewModel you just have to implement the IPostInfoViewModel interface and add a constructor, so I’m going to leave it to you.

### PostViewModel

This is a bit more interesting. You need to implement the IPostViewModel interface and inherit ContentViewModel which is quite easy actually. Your class should look a lot like this:



### ReplyViewModel

Again, I’m leaving this up to you. The interface you have to implement is IReplyViewModel and you will need to inherit ContentViewModel.

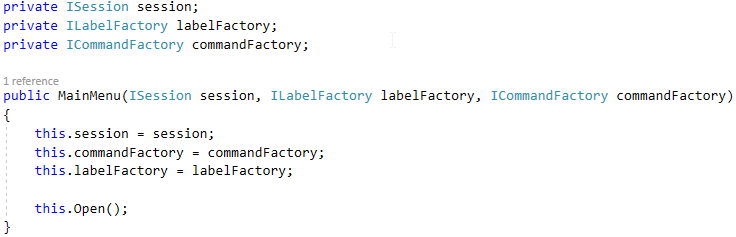
## MainMenu

Introduction

In this part we are going to implement the ExecuteCommand in MainMenu.cs and its corresponding commands.

### Dependencies

Define a private CommandFactory field and inject it through the constructor:



### ExecuteCommand

This method will extract a command name by splitting the current option text and adding the suffix “Menu”, like so:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_00-02-16.png

Then create a command using the factory:

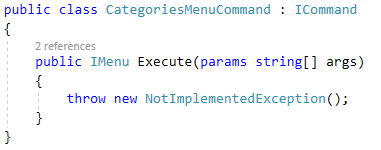
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_00-08-10.png

Finally call the **Execute** method of the command and return the view.

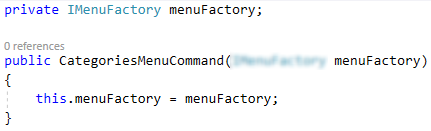
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_00-10-36.png

### CategoriesMenuCommand

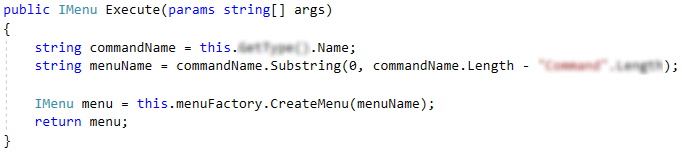
In the **Commands** folder create a new class called CategoriesMenuCommand and implement the ICommand interface.



First of all, define a private IMenuFactory and inject it through the constructor:



Next, in the **Execute** method you should get the type name of the current object, remove the “Command” suffix, create a menu with that name and return it. The implementation should look like this:



### LogInMenuCommand & SignUpMenuCommand

Having the previous command implementation those two will be no match to you. ☺

## CategoriesMenu

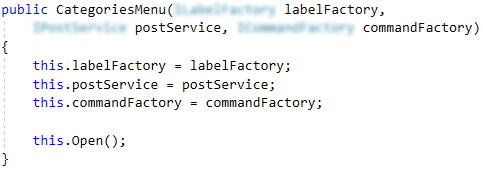
Introduction

In this part we are going to implement DI, ExecuteCommand method changing pages and loading categories functionality.

### Dependencies

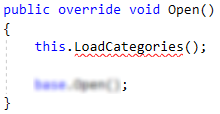
First add IPostService and ICommandFactory private fields and inject them through the constructor along with the ILabelFactory:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_01-07-52.png



### Open

Create an override of the Menu’s Open method which calls LoadCategories method you are about to implement and then calls the base implementation of Open:



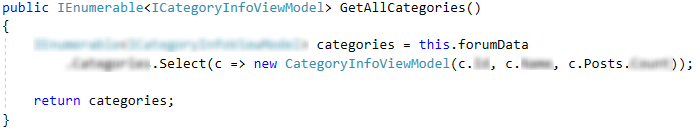
### LoadCategories

This method’s only job is to assign the result of PostService’s GetAllCategories result to categories field:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_01-19-56.png

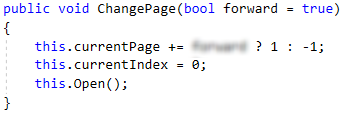
### PostService’s GetAllCategories

Go to post service and add the following:



### ChangePage

This method is responsible for **changing** **pages**, **resetting** the **index** and **reloading** the new page categories.  
Implementation:



### ExecuteCommand

In this method we need to **calculate** the **actual** **index** depending on the page we are and based on that **create** a **command**.

First, we will define an ICommand and set it to null for now.

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_08-59-22.png

**Actual** **index** is calculated when you **multiply** the **current** **page** by the **number** of **categories** shown **per** **page** and add the **current** **index** of this page, like so:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_09-04-07.png

Now if the currentIndex is between 0 and 10 (you are on a category) you should create a command called “ViewCategoryMenuCommand” like so:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_09-09-22.png

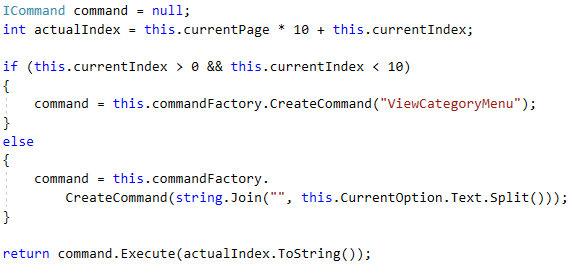
Else, you need to **create** a **command** from the **current** **option** text like you did in the previous menu:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_09-11-12.png

Finally, you need to **return** the **result** of **execution** of the **command**:

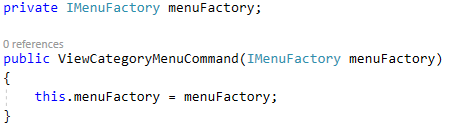
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_09-13-02.png

The whole method should look like this:

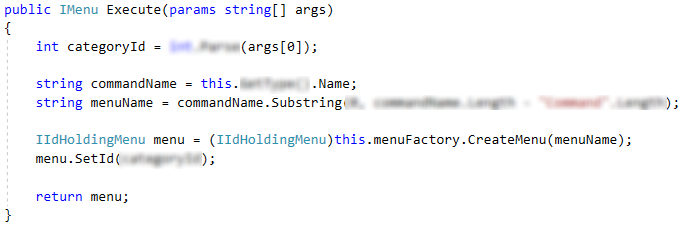


### ViewCategoryMenuCommand

Go to the commands folder and create ViewCategoryMenuCommand that implements the ICommand interface. The dependency you’ll need is an IMenuFactory:



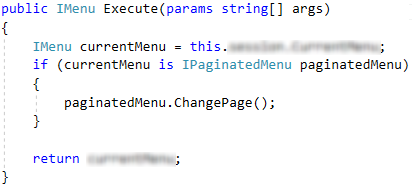
Execute method’s job is to parse the categoryId from form args array, extract the menu name and create a menu. Since the category has an id we will set it to the menu and return it.



### NextPageCommand

The only dependency you need is ISession session and don’t forget to inject it through the constructor.

Execute method does nothing but fetching current menu from session and calling the change page method with no arguments if IPaginatedMenu is assignable from current menu. At the end it returns the menu.



### PreviousPageCommand

Having the previous command implementation, this one is up to you.

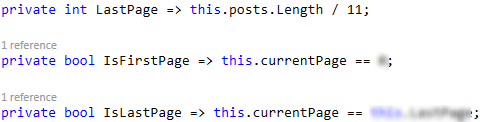
## ViewCategoryMenu

### Dependencies

Add an ICommandFactory field and inject it along with the labelFactory and the postService.

### Pagination properties

These properties will help us know when we are on the last page and when we are on the first one.



### Open

You should override the Open method like you did in the previous menu. It should call LoadPosts and then call the base implementation.

### LoadPosts

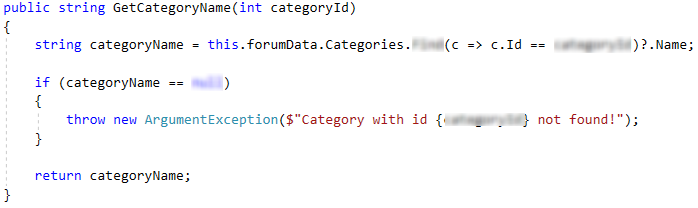
Implement a LoadPosts method with no parameters that calls PostService’s GetCategoryPostsInfo, like this:

C:\Users\CiB0rG\AppData\Local\Microsoft\Windows\INetCache\Content.Word\asd.png

### PostService’s GetCategoryPostsInfo

Go to the post service and having categoryId find all posts from that category and return them as an IEumerable<IPostinfoViewModel>.

### PostService’s GetCategoryName



### ChangePage

You can get the implementation from any IPaginatedMenu, but try implementing it by yourself. Just don’t forget to call Open so it can reload the categories on the new page.

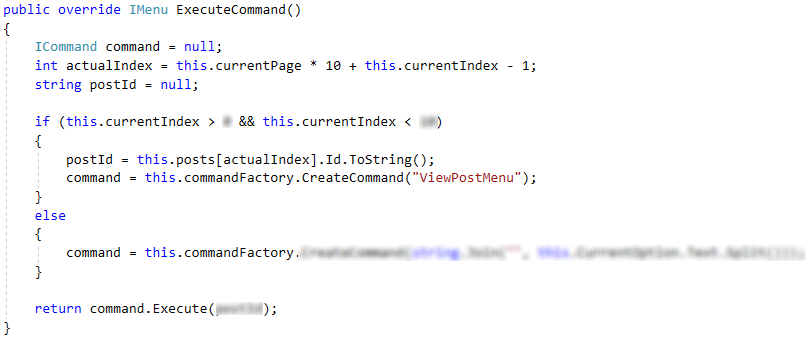
### SetId

This method pretty much speaks for itself, it sets the categoryId to the given id and reloads the posts:

devenv_2018-04-10_11-03-01

### ExecuteCommand

This method looks a lot like in the previous menu, but this time we need а postId in order to create a ViewPostMenu when the time comes. The implementation looks a lot like:



### ViewPostMenuCommand

This command looks just like ViewCategoryCommand. Just create it, implement ICommand, inject IMenuFactory and try hard not to copy/paste.

## SignUpMenu

### Dependencies

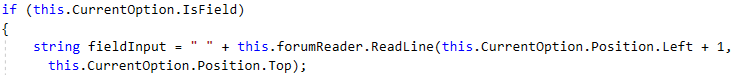
Define IForumReader reader and ICommandFactory command factory and inject them along with the existing labelFactory. Call the base class Open method in the constructor.



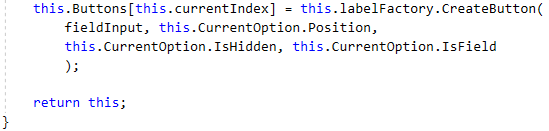
### ExecuteCommand

In this method you should check if the CurrentOption is an input **field** and read information about the user if it is. Otherwise you should try to create a command and execute it.

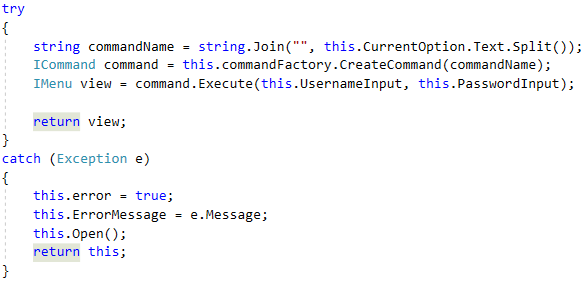
If CurrentOption.IsField is true we first need to read the input at a given location on the screen, like so:



Now, in order to visualize to the user what he just wrote, we need to create a new label an insert it into the same slot in the SignUpMenu’s Buttons array. And finally, we return the current state of the menu:



After we have done that, we need to implement logic about creating a command. In a try/catch block extract the command name form CurrentOption.Text, create a command, execute it and return the resulting menu. Otherwise, if the execution fails catch an **Exception** assigning the **exception** **message** to ErrorMessage, refresh the menu by calling Open() ant return its current state.



### SignUpCommand

Create a SignUpCommand class in the Commands folder, implement ICommand and define and inject IUserService and IMenuFactory.

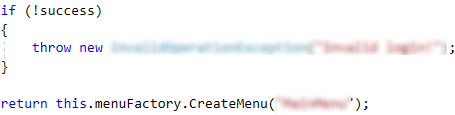
For the Execute method we will need to extract the username and the password from the args array:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_12-27-47.png

After that you need to assign a boolean with the result of userService.TrySignUpUser with the given parameters:

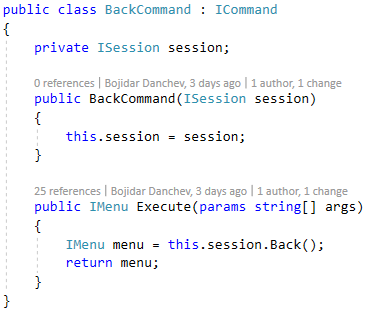
devenv_2018-04-10_12-29-59

Finally, if **success** is **false** throw an exception with message “Invalid login!” or return a new MainMenu if it’s true.



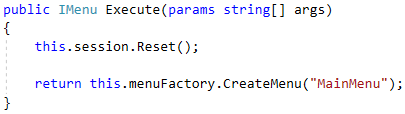
### BackCommand

This command is pretty straightforward:



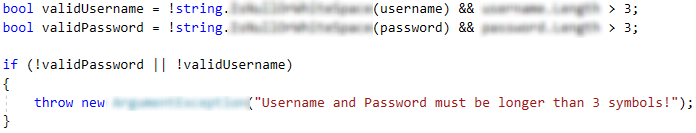
### LogOutCommand

LogOutCommand implements ICommand and has two dependencies ISession and IMenuFactory both injected through the constructor. Execute method looks like this:

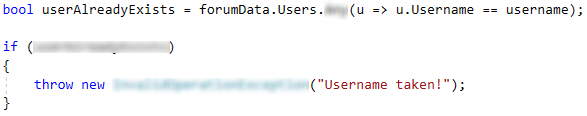


### UserService’s TrySignUpUser

The first thing you have to do in TrySignUpUser is to validate that neither username nor password is null or its length is less than 3. Throw an ArgumentException in any of those cases. The code should look a lot like this:



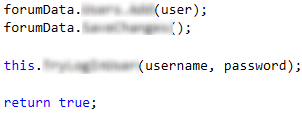
The next check we need to perform is whether a user with that same username exists. Throw an InvalidOperationException if it does. The code:



After that we need to generate an id for the new user to acquire:

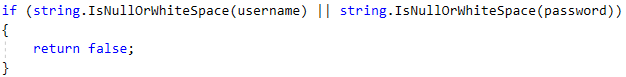
C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_12-53-55.png

Finally, we need to add the new user to forumData’s Users, saveChanges, TryLogInUser, and return true at the very end:

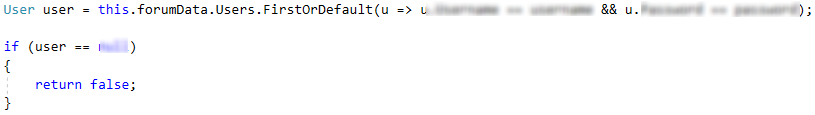


### UserService’s TryLogInUser

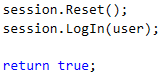
Just like in the previous method we will start with validations. Check if the given username or password is null or empty and return false if any of them are.



Then write a query to fetch a user with the same username and password and return false if it doesn’t pass the null check:



Finally, you need to reset session, log in the acquired user and return true at the end.



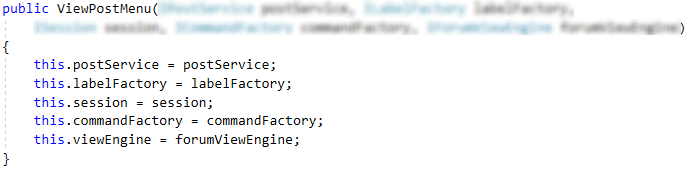
## LogInMenu

Having SignUpMenu implemented this should be boring, but don’t copy/paste.

## ViewPostMenu

### Dependencies

The dependencies you have to add are ICommandFactory and IPostService and constructor should look like this:



### SetId

This method sets the postId with the given parameter and calls the current implementation of the Open method.

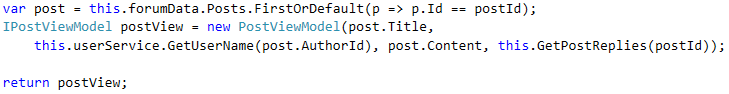
### LoadPost

This method’s job is to assign a post provided by the postService:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_13-46-14.png

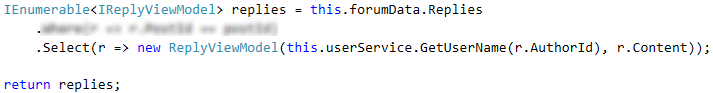
### PostService’s GetPostViewModel

This method returns PostViewModel that contains a **Title**, **Replies**, **Content** and a **Username** that is the author of the current post:



### PostService’s GetPostReplies

Just generate a method called GetPostReplies whose job is to fetch all of the replies for this post and return the as collection of IReplyViewModel

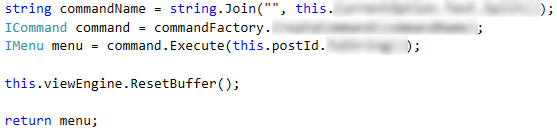


### UserService’s GetUserName

Implementation of this method is simple, just find the user with the **given** **id** and return their **username**.

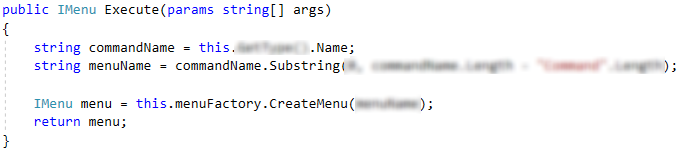
### ExecuteCommand

Don’t forget to call ViewEngine’s **ResetBuffer** after executing the command:



### AddPostMenuCommand

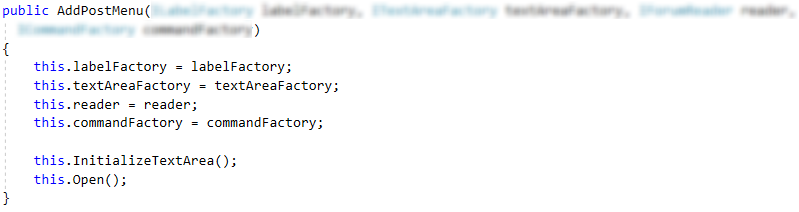
This command has only an IMenuFactory dependency and the execute method you have probably seen too many times:



## AddPostMenu

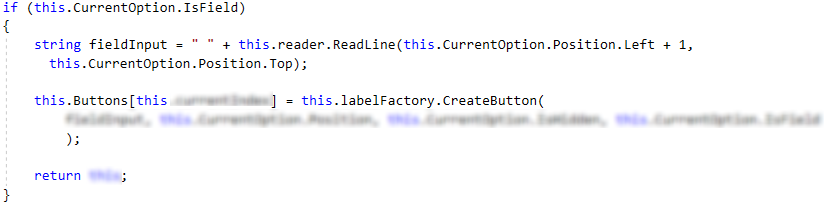
### Dependencies

Add a command factory, then rework your constructor to look something like this:

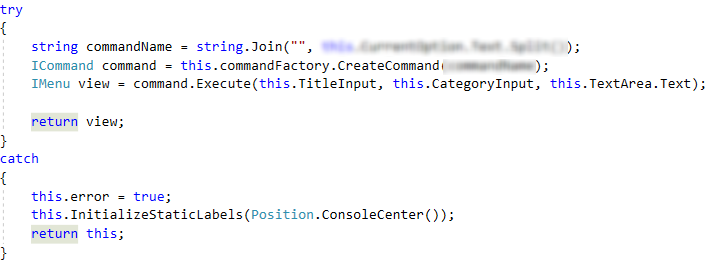


### ExecuteCommand

This is similar to the LogIn menu where we had to keep track if the position we are on is filed or not. The first part is exactly the same.

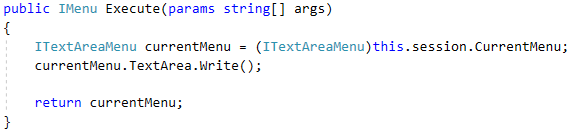


The try/catch differs a bit:



### WriteCommand

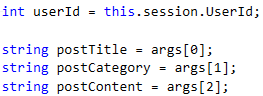
WriteCommand has only an ISession injected trough constructor and the Execute looks like this:



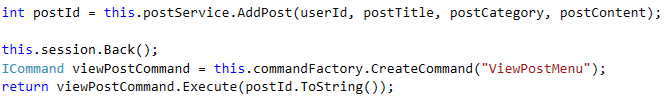
### PostCommand

PostCommand contains ISession session, IPostService postService, ICommandFactory commandFactory injected trough the constructor.

The **Execute** method retrieves the userId from the injected session and the other information needed from the args array:

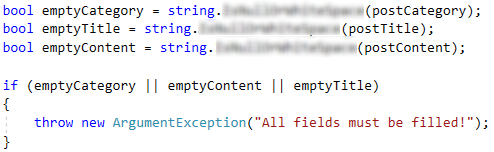


After that calls the post service to save the post, generates a viewPostCommand and returns its execution result:



### PostService’s AddPost

First, we begin with validation for any of the three strings and throwing an exception if any of those validations fail.



Next, we need to ensure that there is a category with such name:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_15-18-32.png

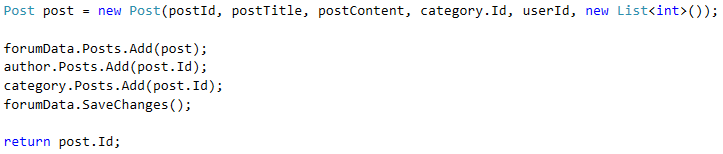
Next you should generate an id for the new post:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_15-19-59.png

Then you need to get the User representing the author using UserService:

C:\Users\david\Documents\ShareX\Screenshots\2018-04\devenv_2018-04-10_15-22-35.png

And finally, we are ready to create a new post and save it in all post collection, the authors posts, and the current category posts:



### PostService’s EnsureCategory

Check if such category exists and create one if it doesn’t.

### UserService’s GetUserById

Similar to the GetUsername but return the whole object instead.

## AddReplyMenu

😊