# Workshop: Forum

## Overview

In this workshop we shall create a functional **Forum** console application. You are given a source code which contains a solution with a single project in it **“Forum.App”**. This is the main project of your application and your business logic.

## Get Familiar with the Project

### Introduction

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

### StartUp

The StartUp.cs’s Main does nothing but instantiate Engine class and start it.

### Engine

Things here are pretty straightforward too. The class holds a forumViewer of type ForumViewer and a menuController of type MenuController (whose methods you need to implement) and IEnumerable<IController> as fields. The Run method holds a while loop that listens for user input.

### MenuState

An enum holding all the states of the menu.

### MenuController

This is the main logic of your menu, and it calls the controllers (you are about to implement) depending of the current state of the menu.

### InvalidCommandException

Custom exception that inherits Exception class with an overridden message.

### UserInterface

The folder contains a ForumViewEngine class and all the components it works with. The most significant ones are the Views. They store labels, buttons and logic for displaying the information given to them. That’s all you need to know for now, you’ll get familiar with the ForumViewEngine API in the process of creating the controllers.

### Controllers

You have a Contracts folder that holds the interfaces your controllers should implement, and a single controller called MainController which renders the main view and executes commands. The fun of implementing the other controllers is on you. Good luck!

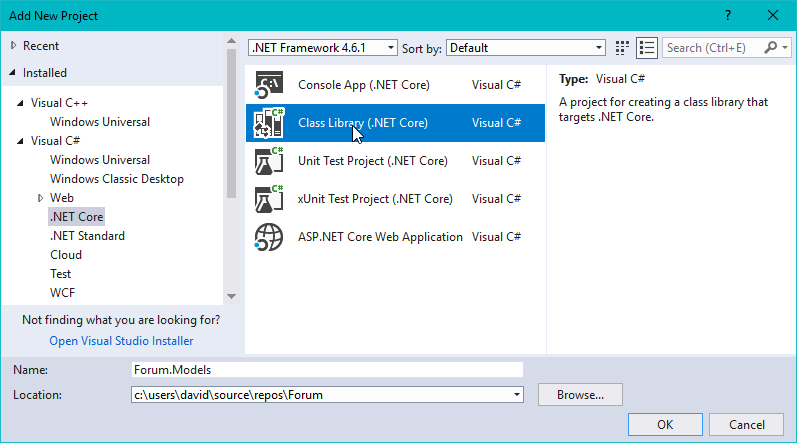
## Create Object Models

### Introduction

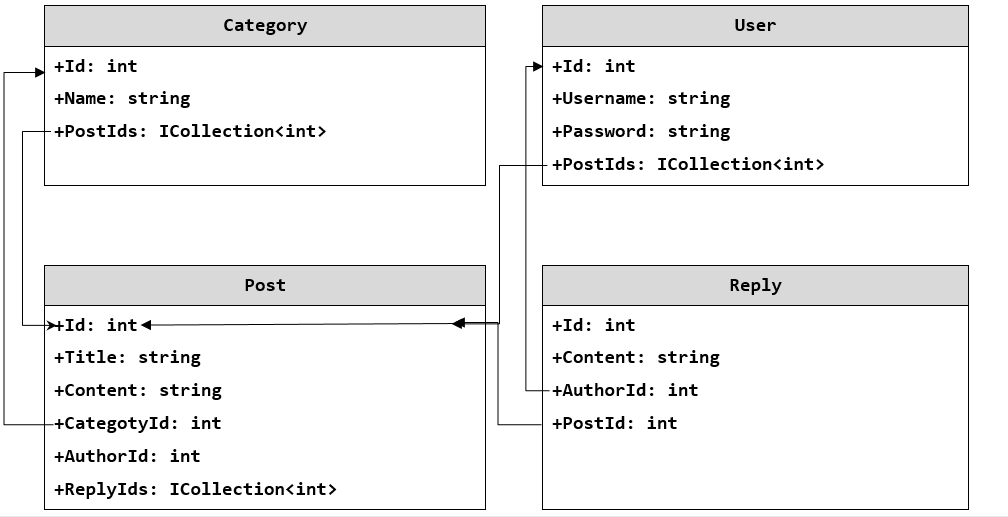
Before stepping into implementing we need to implement the models that hold the information for our entities: Category, User, Post, and Reply.

### Create Forum.Models Project

Create a new project called “**Forum.Models**” and choose the “Class Library” for its output type.



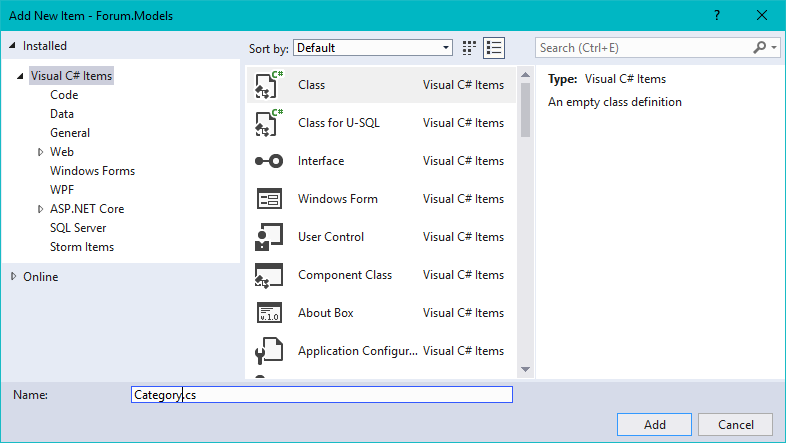
### Create Model Classes

Create the models with the following data: 

For each entity we need to create a class, holding properties show above. As you can see each entity holds information about itself (id, username, etc.) and collection of or a single id of the object that is related with. For example every category has Id, Name, and a collection of integers, representing the ids of the posts it contains.

#### Create Category Class

First off we need to create a public class called Category.cs

****

Then we need to add the properties shown in the diagram above.

First create the property holding the id

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-22-02.png

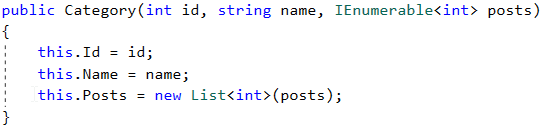
Second, create the property holding the name of the category

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-23-34.png

Third, create the property holding ICollection of numbers representing post ids

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-25-10.png

Last, create a constructor that takes int id, string name, IEnumerable<int> posts as parameters and assigns them to the corresponding properties.



Note: We don’t need private fields behind the properties, because **we aren’t going to perform any validation** on this level.

#### Finish the Model classes

Do the same to create the other three models using the information form the diagram shown above.

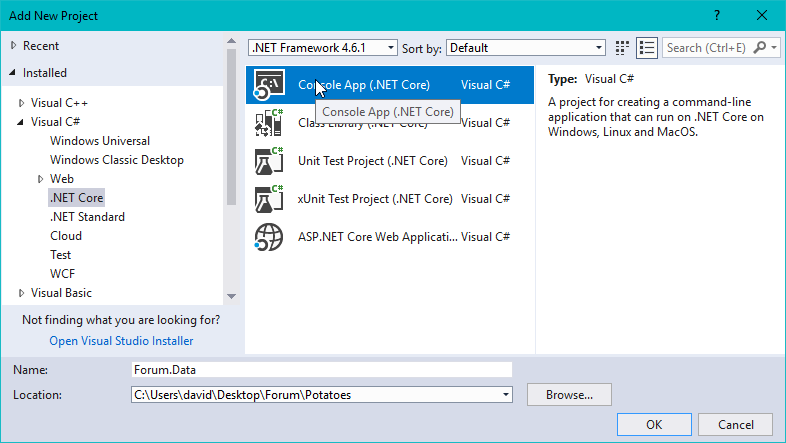
## Create Data Manager

### Introduction

In this part we’re going to implement a [Data Mapper](https://en.wikipedia.org/wiki/Data_mapper_pattern) that reads a row from [.csv file](https://en.wikipedia.org/wiki/Comma-separated_values), converts it into a model and vice-versa. After that we’re going to create a class that holds the collections of all our entities and speaks with the Data Mapper.

### Create Forum.Data Project

Create a project called “Forum.Data” and choose Console App so you can test your classes in it.



### Create a DataMapper class

#### Fields

The first property we are going to initialize is DATA\_PATH. It will store the directory path were we will store our data (.csv files).

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-31-52.png

The second one is called CONFIG\_PATH and it stores the configuration file name. DATA\_PATH and CONFIG\_PATH are user concatenated to specify where the file should be.

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-33-10.png

The third constant is called DEFAULT\_CONFIG and it stores the content of the configuration file.

"users=users.csv\r\ncategories=categories.csv\r\nposts=posts.csv\r\nreplies=replies.csv"

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-35-17.png

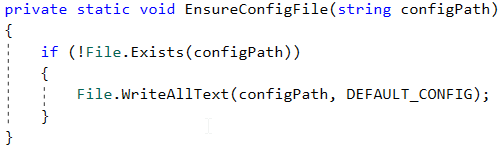
The last field is the object representation of the configuration file in you class and is called config. As you might have guessed it’s going to be a Dictionary<string, string> where the key represents the entity collection group and the value the name of the file it is stored.

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-37-08.png

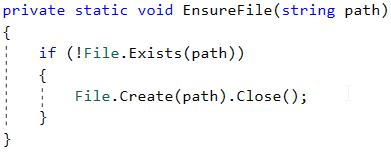
#### Helper Methods

Next, we should start implementing the helper methods, in order to simplify the code and avoid repeating it in the methods, who take care of parsing and reading/writing entity data.

The first method we are going to implement is called EnsureConfigFile(string configFilePath), his return type is void and as you can tell by its name, his job is to check whether a file form the path exists, and creating a new one with the string form **DEFAULT\_CONFIG** constant if it doesn’t. When you finish it will look something like this:

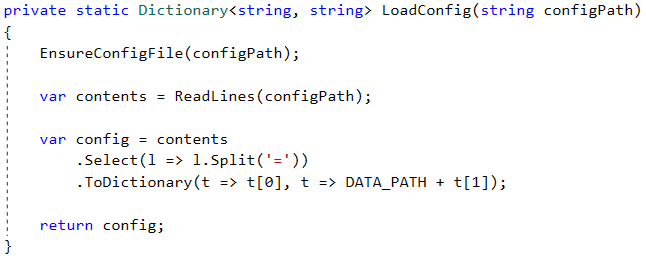


Next method is called EnsureFile(string path), again his return type is void and the logic in it is pretty similar to the previous method only if the file doesn’t exist it creates an empty one. The code of course is similar too:

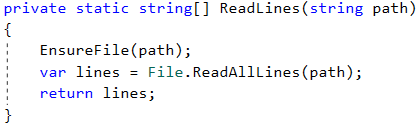


Note: Don’t forget to close the file.

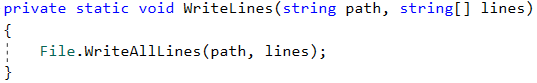
Next in line is the LoadConfig(string configFilePath) method. Since it’s going to load the config its return type is Dictinary<string, string>. The logic is simple: Call the EnsureConfigFile method you have implemented with the given configFilePath as a parameter. After that you just need to read the data, split it with delimiter '=' and convert it to Dictionary<string, string> where keys are names of the collection and values are DEFAULT\_PATH + names of the files:



Our mapper needs something that **reads all the lines** of the files. This is going to be the static ReadLines method, with string array as a return type and **string path**. Its job is simple, calls EnsureFile with the given path, reads all the lines from the file and returns them as an array of strings:

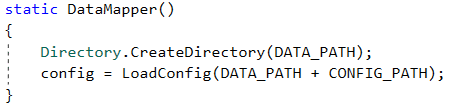


And finally something that **writes into files**, when our **data changes**. Since we’re just writing into the files, the return type of our method is going to be void, his name will be WriteLines and the arguments that it would take are **string** representing the **path**, and **string array** containing the **lines (entities)**. The only thing that it would do is to write all the data in the file with the given path.



#### Constructor

The thing we need to initialize in the **constructor** is a **directory** for the **configuration file** and the object config that represents **configuration file**.



#### Parsers

Now we have to create methods that read the information and converts them to a list of models and corresponding ones that take list of models and writes the to the file in specific format. The entities that we are dealing with will be in the following formats:

Category format: {Id};{Name};{postId1,postId2,postId3...}

User format: {Id};{Username};{Password};{postId1,postId2,postId3...}

Post format: {Id};{Title};{Content};{CategoryId};{AuthorId};{replyId1,replyId2,replyId3...}

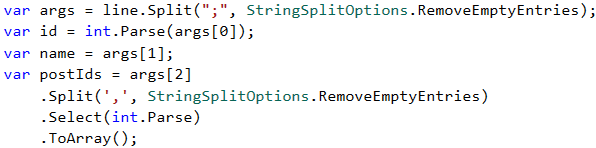
Reply format: {Id};{Content};{AuthorId};{PostId}

Each line will contain information about a single entity.

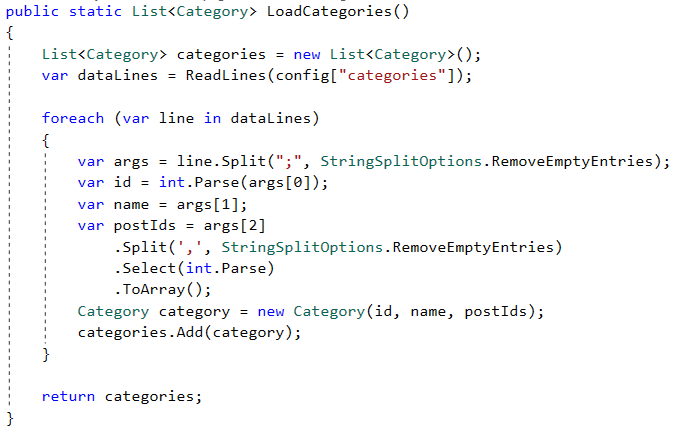
The fist method we are going to implement is LoadCategories() and its return type will be List<Category>. Again, the logic is not a big deal, you need to instantiate a new List<Category> and read the lines from the file using the **helper method** and the **path** in config (config["categories"]).

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_16-55-39.png

After that you have to **split** each line by delimiter ';' so you can get the id (first argument), the name (second argument) of the category and the post ids (third argument split by ',').



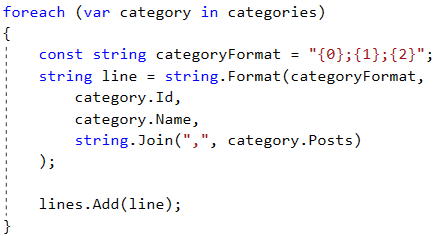
The next step is to **create** an instance of the Category **model** and add it to that list of yours. When you are done with **all the lines** just return the **list of categories**. After you finish it the **method** should look something like this:



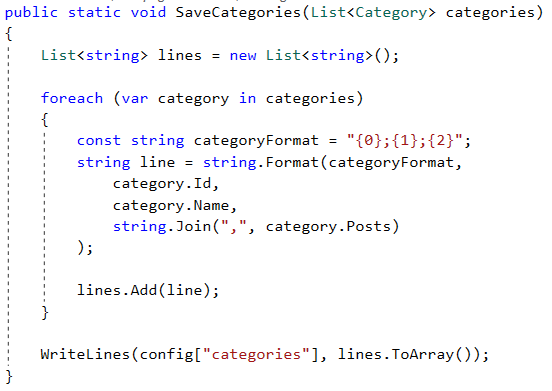
Now we need a **method** which does **the exact opposite** of the one we just finished. Its name will be SaveCategories, it’s return type – void and it will take a List<Category> as argument. You need to instantiate a list again, but this time, it will contains **strings (lines).**

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_17-02-18.png

For each **model** in the list **create a string**, which follows the **format** shown above and add it to the list.



At the end call the WriteLines method with the path from config and the **lines you’ve created**. When you finish it should look something like this:



Following those instructions and repeat that to create: LoadUsers, SaveUsers, LoadPosts, SavePosts, LoadReplies, SaveReplies**.**

### Create ForumData class

This is going to be the holder of all of our data. It will have properties for all the entity groups:

**Categories**:

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_17-43-00.png

**Users**:

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_17-42-43.png

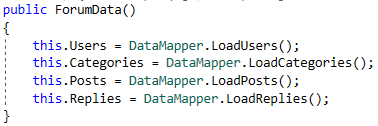
**Posts**:

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_17-43-10.png

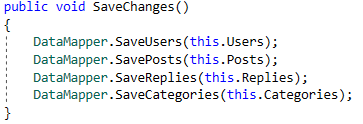
**Replies**:

C:\Users\david\Documents\ShareX\Screenshots\2018-03\devenv_2018-03-06_17-43-19.png

The next step is defining a constructor that calls the corresponding DataMapper methods in order to initialize collections. It should look something like this:



Finally we need a method that saves the changes we made to our collections.



#### Test Your Code

Since you’ve created a **console application** at first you’ll have a Program.cs file with Main method in it, where you can test. You don’t have logic to read the user input, **therefore** you need to **add the information** in the files manually (**following the formats!**) in order to see if the methods you just wrote work in **the way you expect**. After you are done testing you can delete Program.cs file, right-click on the project (alt + enter) and set Output type to be “**Class Library”** and you are good to go**.**

In the next part, we will implement the controller logic and everything else.