# Forum Documentation

This document is a guide to create a fully-functional **Forum System** in ASP.NET MVC with SQL Server database using Entity Framework and MVC.

# Overview

[**ASP.MVC**](https://en.wikipedia.org/wiki/ASP.NET_MVC) – this is a [**web application framework**](https://en.wikipedia.org/wiki/Web_framework) developed by Microsoft, which implements the **model–view–controller** ([**MVC**](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)) pattern, with which you should already be familiar with. In other words, this gives you a **bare bone working web app** (you will see that you can start it immediately after creating the project) out of the box, on top of which you can **build your own app**. Consider it our **foundation**.

[**Entity Framework**](https://en.wikipedia.org/wiki/Entity_Framework) – basically, this gives you a way to **interact with a database** by making you see database objects ([**tables**](https://en.wikipedia.org/wiki/Table_(database))) as classes. Once familiar with **object-oriented programming** you should appreciate how handy this is.

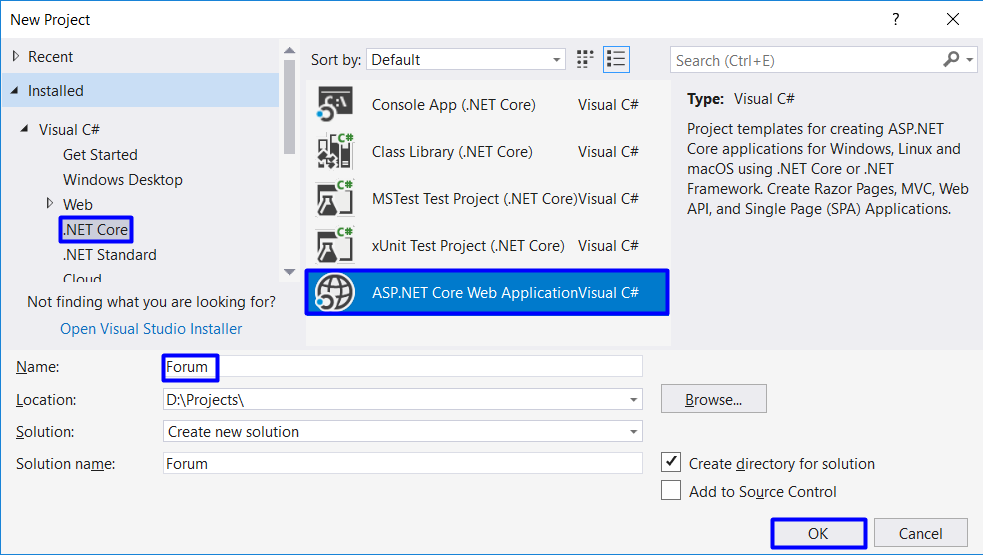
[**SQL**](https://en.wikipedia.org/wiki/SQL) – query language used for **managing a database**. In our case, Entity Framework will take care of this.

**NOTE:** If you are using the **skeleton** start from **chapter VI**.

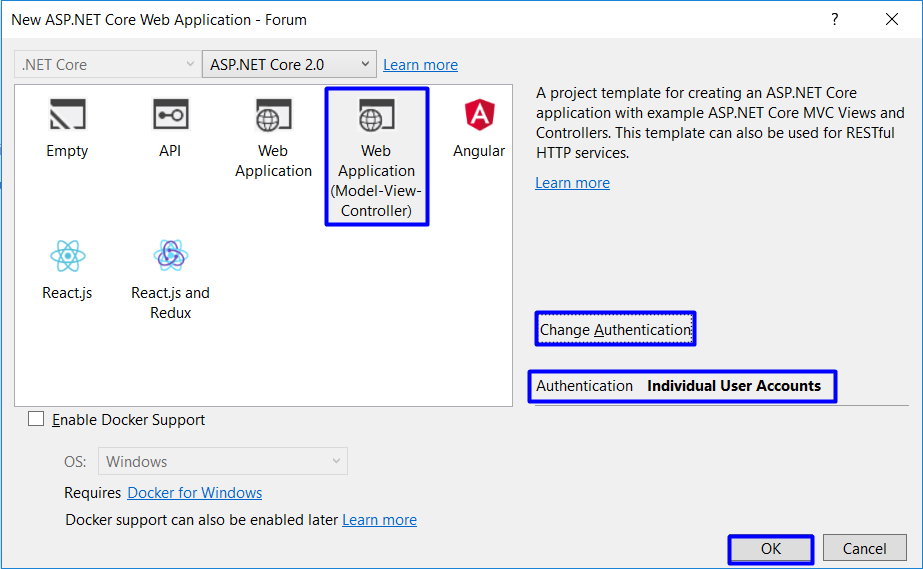
# Initial Setup

In this section we will setup our project and lay the foundations.

## Create a New ASP.NET Core MVC Application

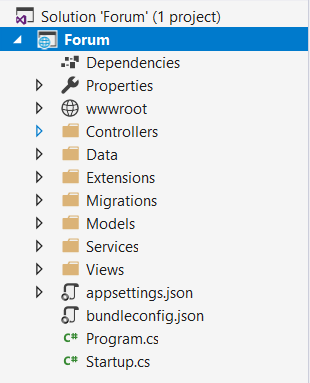
First, let's start by creating an **ASP.NET Core MVC Application** in Visual Studio. Don't forget to name the project appropriately, as if you leave this for later, you can encounter major problems. All code in the guide is made in a project with the name "**Forum**": 

In the next window, choose "**MVC**". Also change the authentication method to "**Individual User Accounts**":



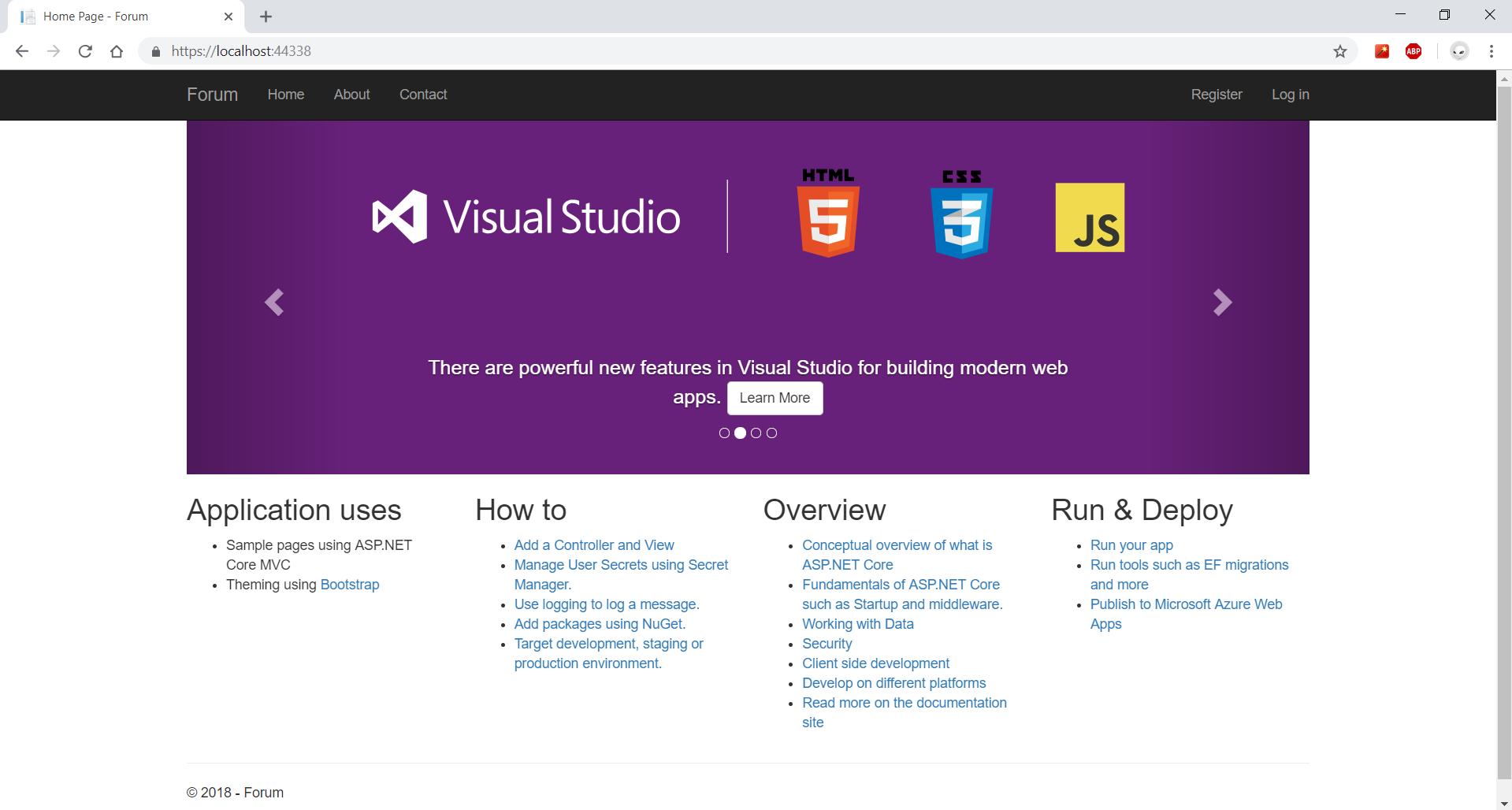
**Make sure you are creating ASP.NET Core 2.0 Project!**

Click on **[OK]** and you should see the following project structure:



## Run the Application

Run the application to see what was generated by the Visual Studio MVC application template. Press **[Ctrl+F5]**.

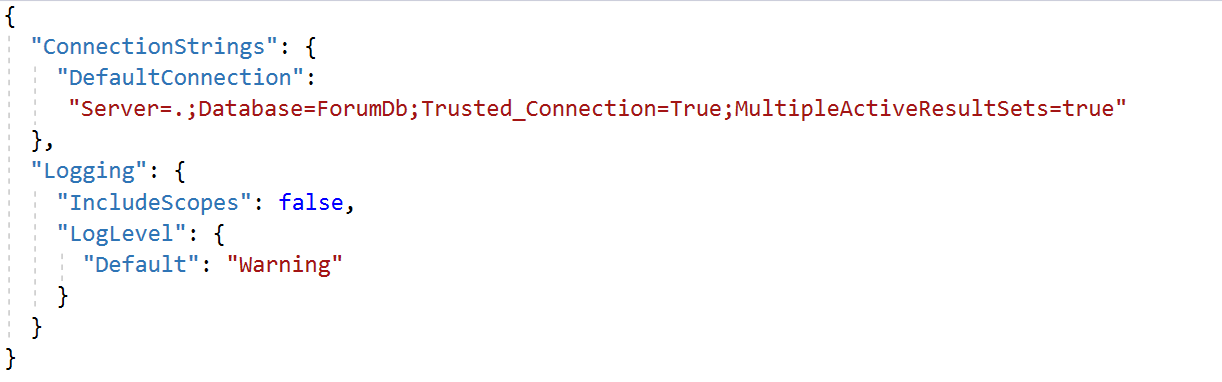


## Configuration

Before we continue with the registration of our first user, we have to make some configurations.

Open **appsettings.json** and change the **DefaultConnection** like the image below.

**Note:** The Server part may not be the same. (**.\SQLEXPRESS, (LocalDb)\MSSQLLocalDB**)



## Register a User

Click on the button in the upper right corner and **register a user**. **If you register** a user successfully **proceed with the next steps**.

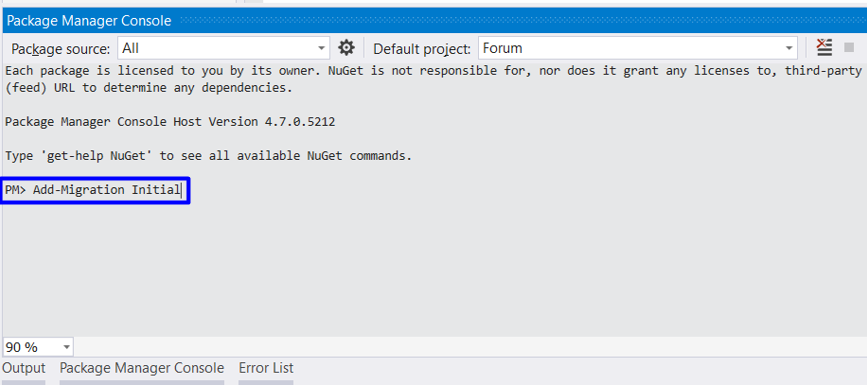
**If you get an exception**, you need to **check the next step**.

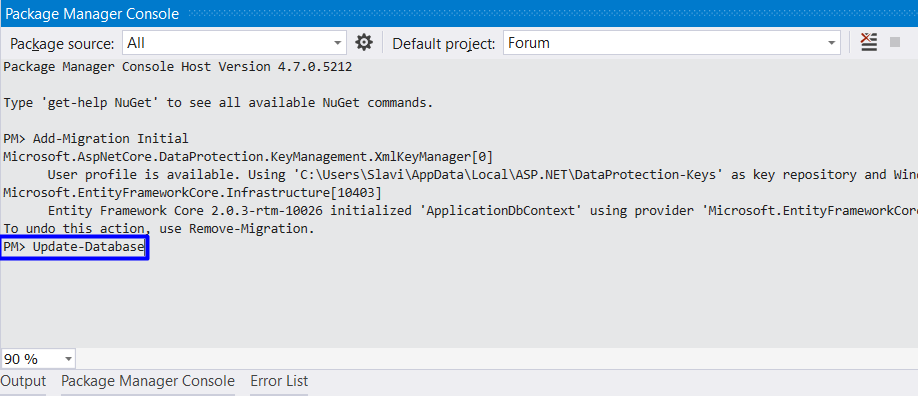


First, open the package manager. In the top left corner of the screen click "**View -> Other Windows -> Package Manager Console**":

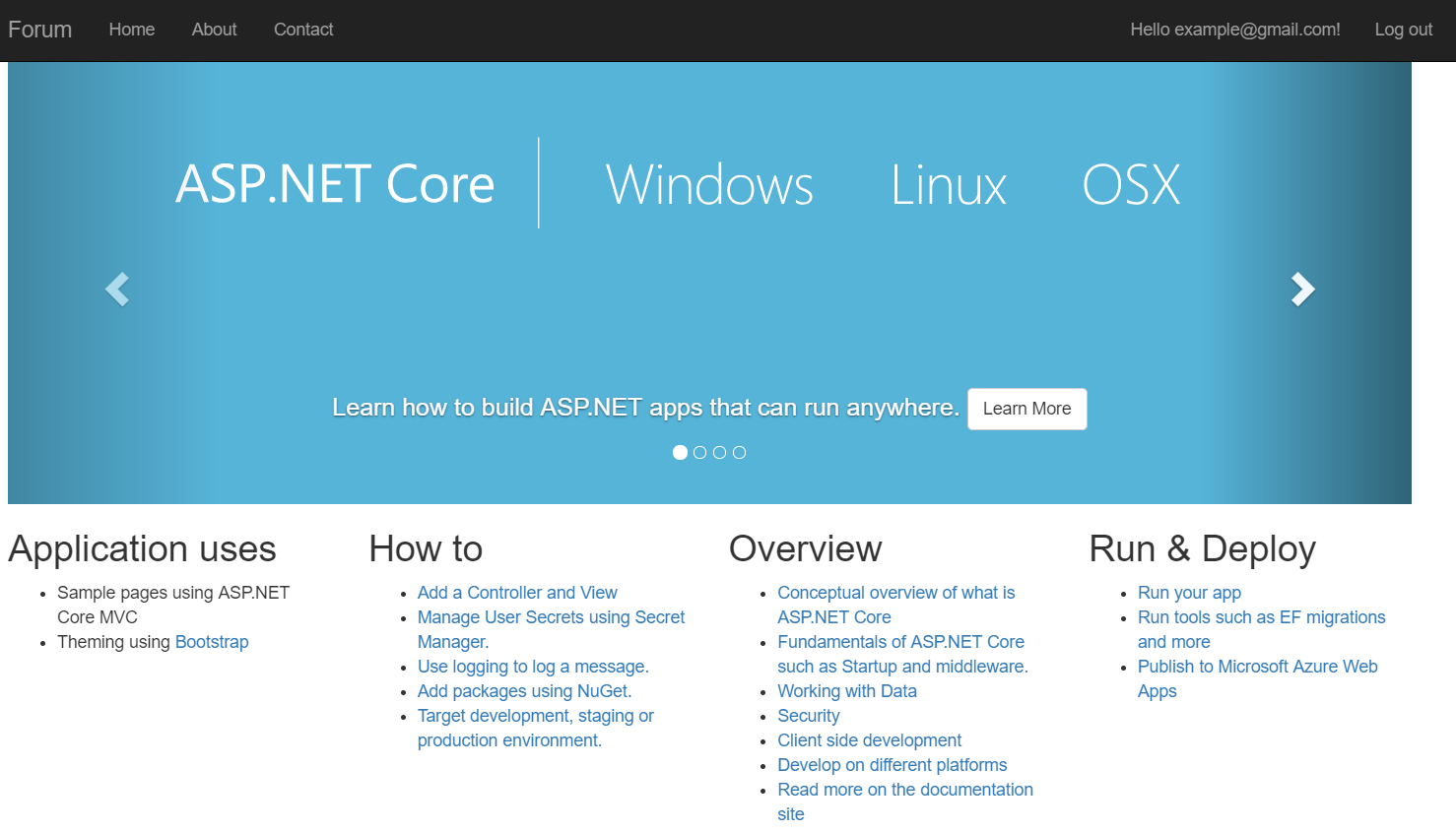


Write "Add-Migration Initial" and click Enter. After that write "Update-Database" and again click Enter.





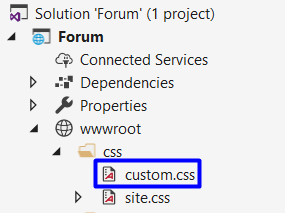
Now refresh browser and you must successfully register user:



After you are done, **proceed with forum creation**.

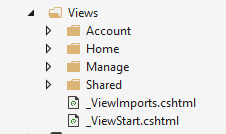
## Plug in Custom Bootstrap Files

We need to **insert our bootstrap file** in the project so we can start using it. This can be done by adding "custom.css" file:



## Setup the General Layout of the Forum

Now we need to set the **general layout**. Just head to the "**Views"** folder. This is the folder that **holds all the views** for the project. Inside you can see that there are some other folders:



* **Account** - This folder holds all views that are related to **user registration** and **login**
* **Home** - Contains the **home page views**
* **Manage** - Views related to **managing user accounts** (changing password, adding a phone number, etc.)
* **Shared** - folder containing views that are **common for all of the project** (general layout, contact page, etc.)

As you probably guessed we need the "**Shared**" folder. In it you will find a file named "\_Layout.cshtml". It begins with an underscore because it is a **partial view** (more on that later).

In it just **delete all code** and **paste the provided** one:

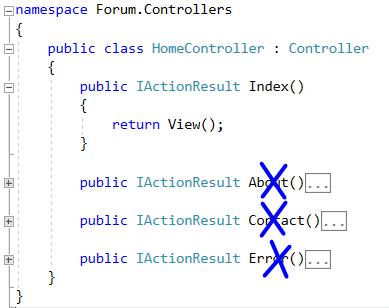
|  |
| --- |
| \Views\Shared\\_Layout.cshtml |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <meta name="viewport" content="width=device-width, initial-scale=1.0" />  <title>@ViewData["Title"] - Forum</title>  <**environment** **include**="Development">  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />  <link rel="stylesheet" href="~/css/site.css" />  <link rel="stylesheet" href="~/css/custom.css" />  </**environment**>  <**environment** **exclude**="Development">  <**link** rel="stylesheet" **href**="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"  **asp-fallback-href**="~/lib/bootstrap/dist/css/bootstrap.min.css"  **asp-fallback-test-class**="sr-only" **asp-fallback-test-property**="position" **asp-fallback-test-value**="absolute" />  <**link** rel="stylesheet" **href**="~/css/site.min.css" **asp-append-version**="true" />  </**environment**>  </head>  <body>  <nav class="navbar navbar-inverse navbar-fixed-top">  <div class="container">  <div class="navbar-header">  <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">  <span class="sr-only">Toggle navigation</span>  <span class="icon-bar"></span>  <span class="icon-bar"></span>  <span class="icon-bar"></span>  </button>  <**a** **asp-area**="" **asp-controller**="Home" **asp-action**="Index" class="navbar-brand">Forum</**a**>  </div>  <div class="navbar-collapse collapse">  @await Html.PartialAsync("\_LoginPartial")  </div>  </div>  </nav>  <div class="container body-content">  @RenderBody()  <hr />  <footer>  <p>&copy; 2018 - Forum</p>  </footer>  </div>  <**environment** **include**="Development">  <script src="~/lib/jquery/dist/jquery.js"></script>  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>  <**script** **src**="~/js/site.js" **asp-append-version**="true"></**script**>  </**environment**>  <**environment** **exclude**="Development">  <**script** **src**="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"  **asp-fallback-src**="~/lib/jquery/dist/jquery.min.js"  **asp-fallback-test**="window.jQuery"  crossorigin="anonymous"  integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">  </**script**>  <**script** **src**="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"  **asp-fallback-src**="~/lib/bootstrap/dist/js/bootstrap.min.js"  **asp-fallback-test**="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"  crossorigin="anonymous"  integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">  </**script**>  <**script** **src**="~/js/site.min.js" **asp-append-version**="true"></**script**>  </**environment**>  @RenderSection("Scripts", required: false)  </body>  </html> |

## Delete Unnecessary Actions and Views

Now if you head to the Home Controller (**"Controllers/HomeController.cs"**) you should see that there are four methods (actions - **Index()**, **About()**, **Contact()** and **Error()**). You don’t need **About()**, **Contact()** and **Error()**.

Also, if you look at the home page **we don't have links** to them anymore. You can **safely delete them**:

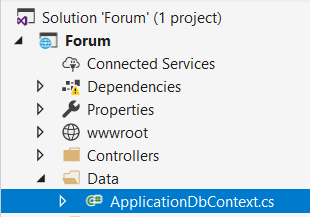
This also means that **you can safely delete the views** that are linked to them. You can find them at "**Views/Home**":



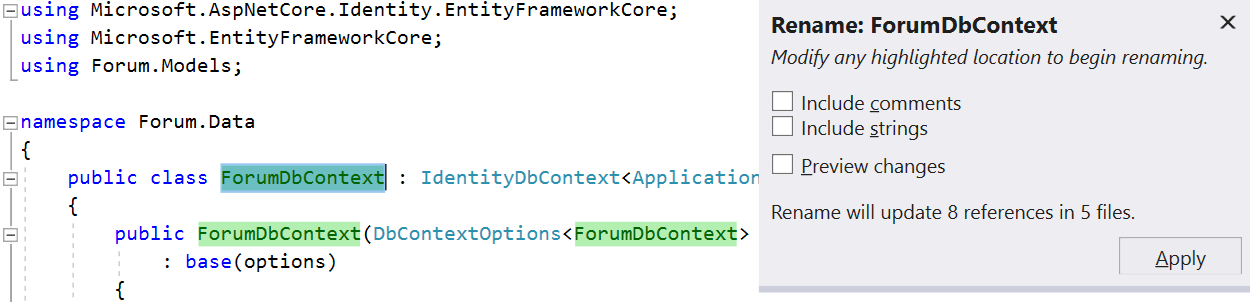
# Edit Class Structure

## Rename ApplicationDbContext

First, rename the file "**ApplicationDbContext.cs**" to "**ForumDbContext.cs**", because now it holds exactly that, the **forum database context**:



And then **rename the class** (Rename with **[Ctrl + R, R]** after selecting the name of the class):



**Rebuild the solution** and if there are any errors, fix them by adding the missing libraries.

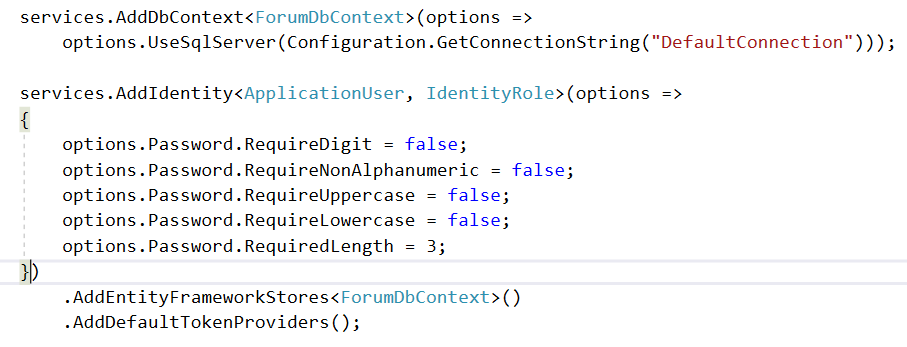
## Modify Default Password Requirements

**ASP.NET MVC** has really complex password requirements. We can change this by editing three separate files:

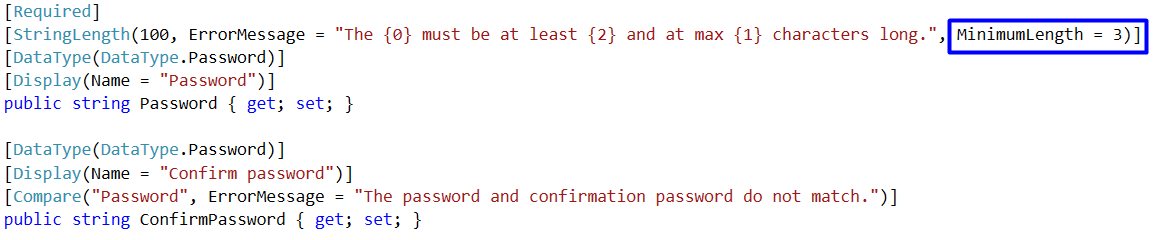
First go to **"Startup.cs"** and look for this code:



Make the **required length equal to 3** and **the rest equal to false**. This will be much more comfortable for testing.

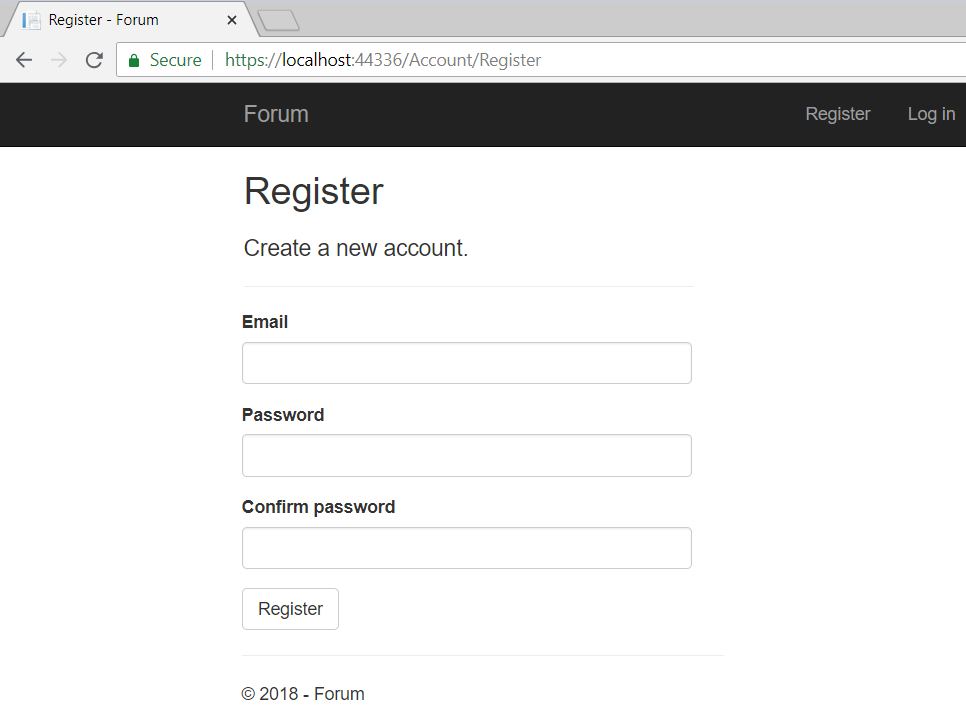


Then go to "**Models/AccountViewModels/RegisterViewModel.cs**" and find these and **edit them to be equal to 3**:



## Test Identity

You can test if the authentication module actually works. Just **start the application** and **click on register**. Register a new user and try to log off and to log in again:



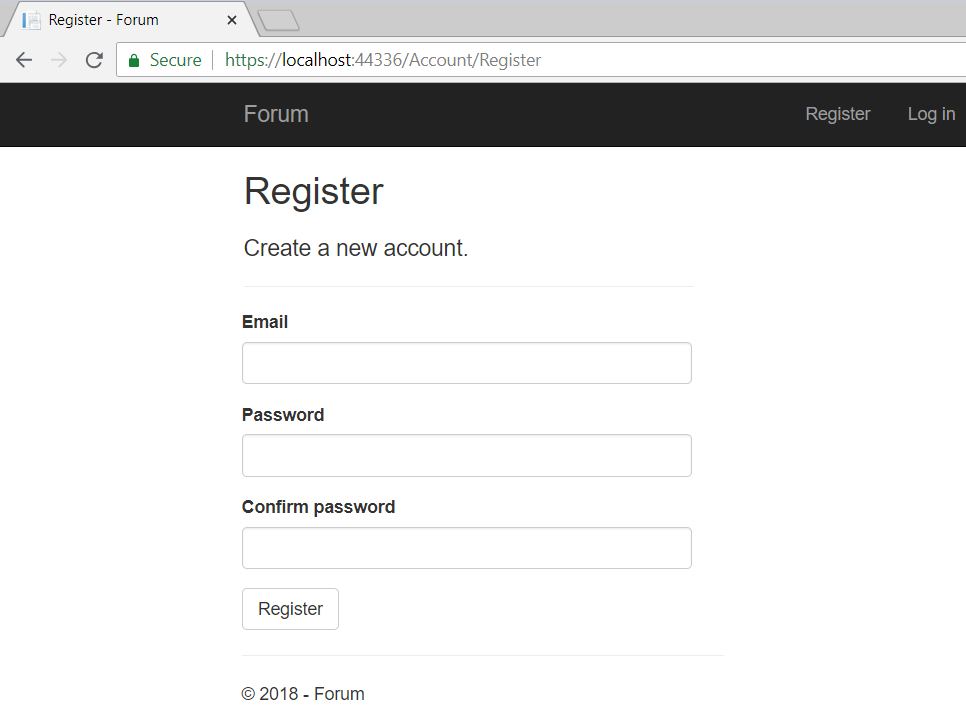
# Get Familiar with Razor Views

## Inspect the Register View

Go to file "**Views/Account/Register**". This is the file that renders the view for Action "**Register**" in the "**AccountController**":



So, the above code renders this:



The first thing to notice is the first line:



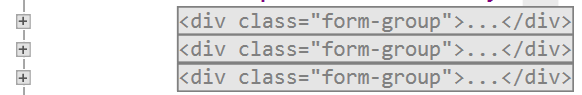
This means that this view works with a class called **RegisterViewModel**, located in the models folder (the view's [**View Model**](http://stackoverflow.com/questions/11064316/what-is-viewmodel-in-mvc)). More on that later.

There is a single **HTML** **form** in the view (**forms** are used to **gather information** from the user):

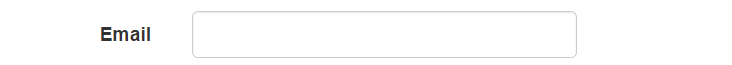


It is used to "**Post**" information and has some **classes** for styling.

Inside you can see three **<div>**s:



A **form group** represents some **grouped elements inside a form**. There are two elements in it -a **label** anda **text box**. The above div represents this:

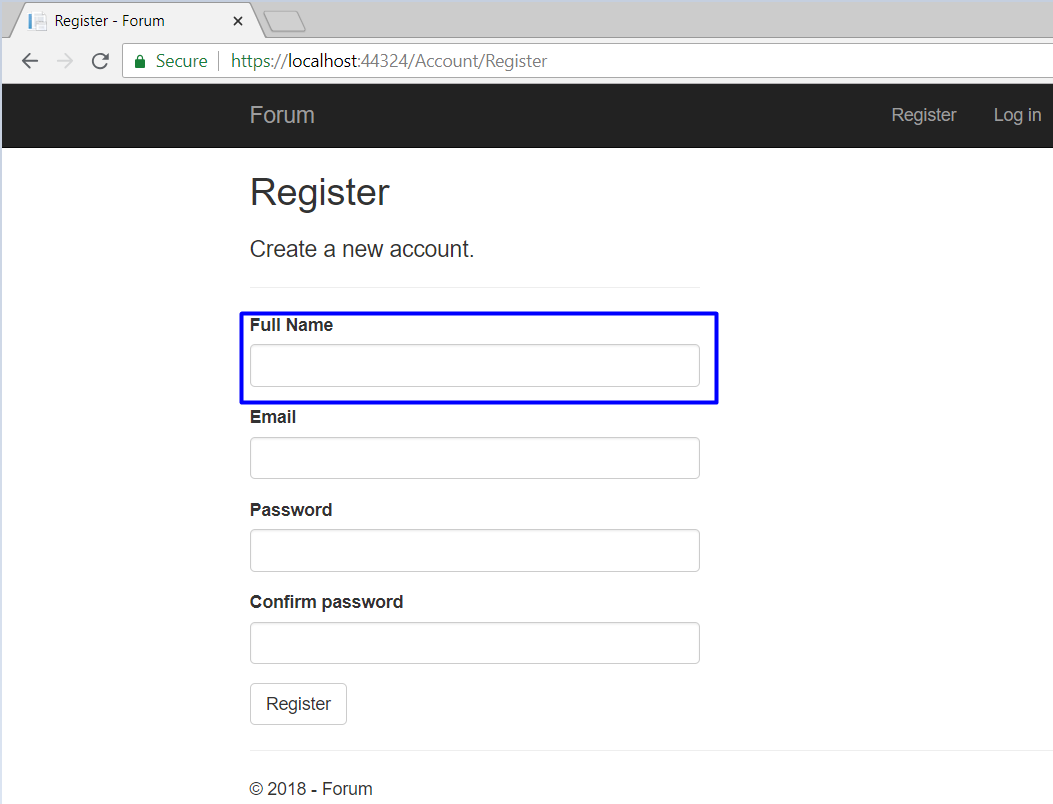


The other **<div>**s are analogous.

The fourth one has a **button** in it. This button submits the form along the information gathered.

# Start Working with Project Models

There is a problem with the application user. The model has no full name. We need to add this:



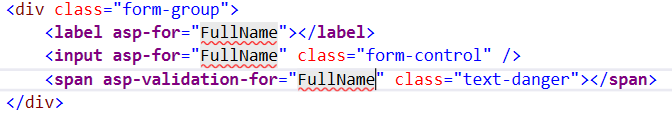
In order to have a full name saved for every user in the database we need to:

1. Edit the **view**
2. Edit the **model**
3. Edit the **controller**

As we recently worked with the views, let's start from there.

## Edit the Register View

Go back to "**Views/Account/Register.cshtml**". Remember that there were some divs **representing every element on the register page**. We need to **add additional one** for the full name. Insert one **before** the **email**:



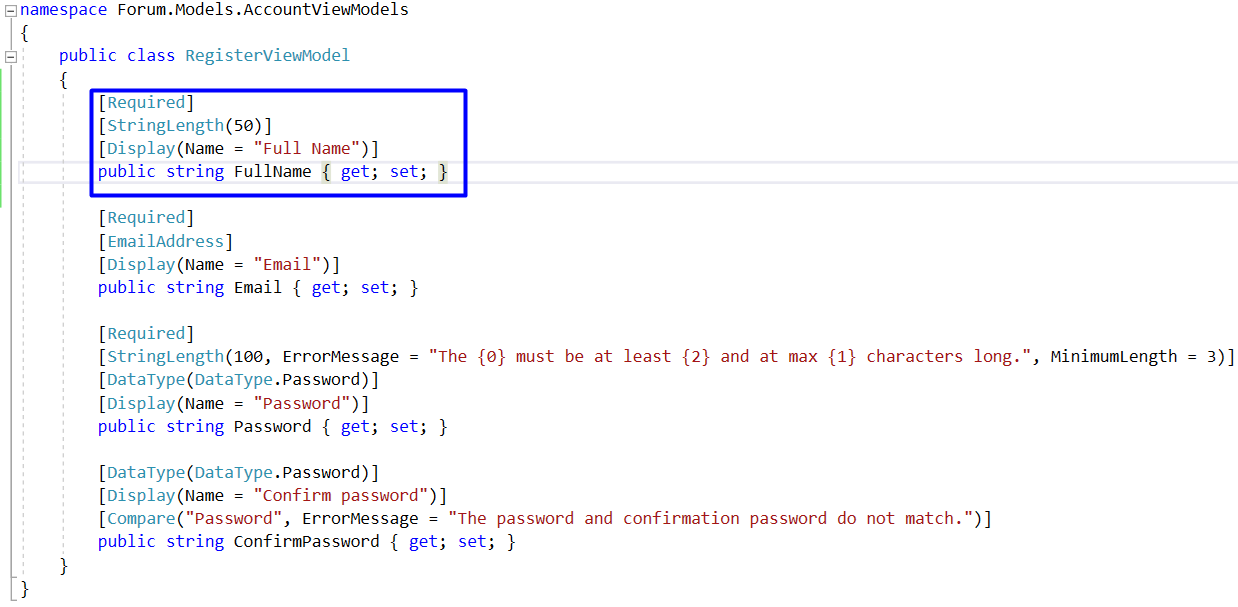
But there is a problem, our model has no **property** for "**FullName**". That's why you don't get any autocompletion and "**FullName**" has red underlining.

## Edit the Register View Model

There are **two kinds of models** in ASP.NET. **Data models** and **View models**. In short, **data models are meant to interact with the controller** and **view models with the views**.

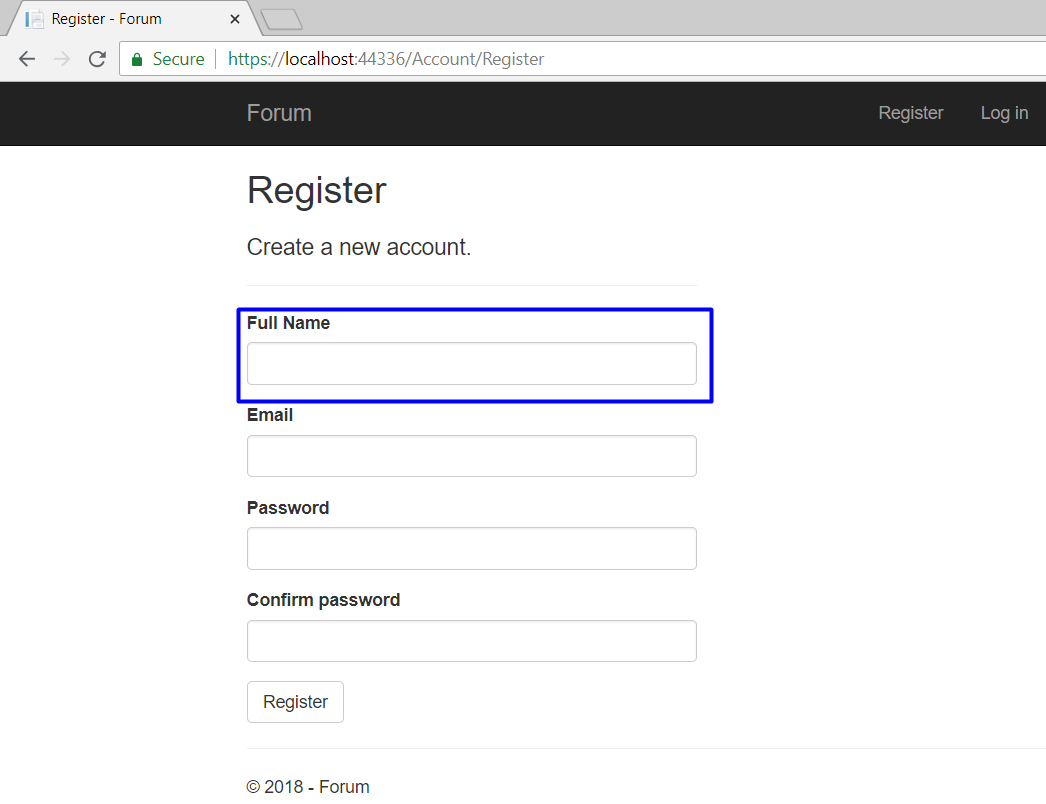
Go to "**Models/AccountViewModels.cs**". This is the model that our view works with.

Add the following lines of code:



This will **add a new property** called **FullName**, that is required and its maximum length is 50.

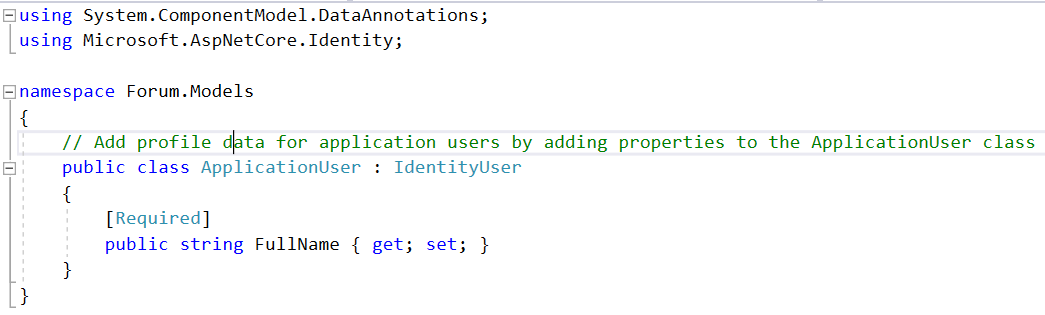
Now you can **compile the project** and see that you have a **new textbox on the register view**:



But **it is not functional**. It doesn’t save data in the database.

## Edit the Application User

Head to the **"Models/ApplicationUser.cs"** and **add a string property called FullName**:



The **property should be public**. Include an [attribute](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/) [**Required**]. In this case the attribute is **used for validation** that makes sure that a user without full name **cannot be inserted into the database**.

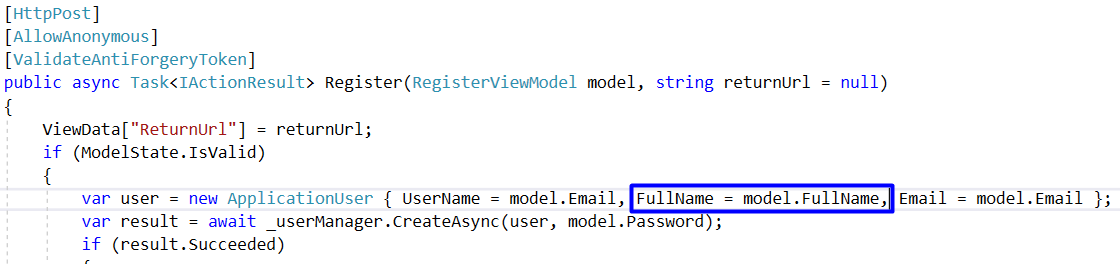
**Don’t forget** to **add missing libraries** for the attribute.

## Edit Register Action

The only thing left to do is to set the full name when the user is registered.

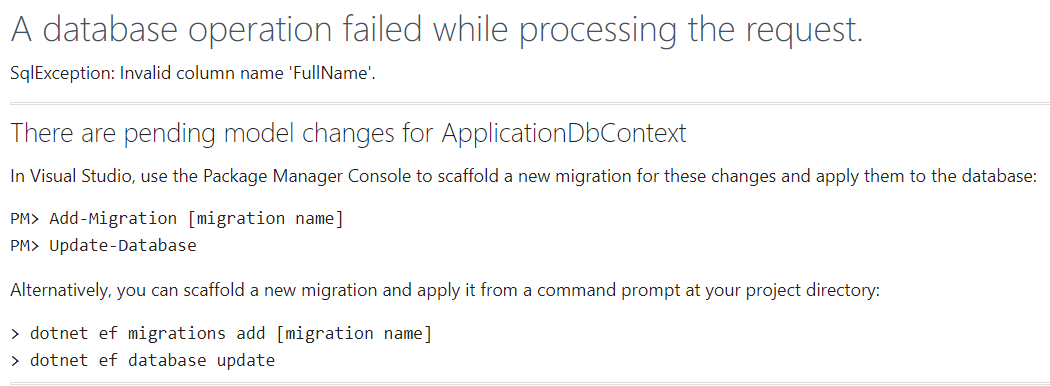
Go to "**Controllers/AccountController.cs**" and find the Register method.

Add the following code:



## Add Database Migrations

Now, if you try to register a new user, you should **get an error**:



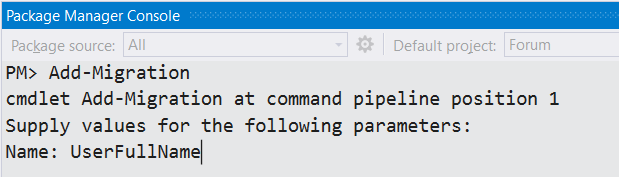
**This error tells us that the database has changed**. We need to explicitly state that we want the **database to be modified if any changes are made** to the entities. Or the easy way - drop your base.

First, open the package manager. In the top left corner of the screen click "**View -> Other Windows -> Package Manager Console**":

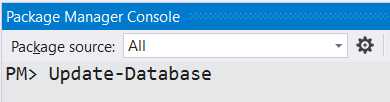


Type:

|  |
| --- |
| **Add-Migration**  **UserFullName** |



Now you can run in package manager console the following command: "Update-Database":



**Make sure** that you can **start the app**, **register a new user** and the User entity in the database has a full name column.

# Creating Topic Entities

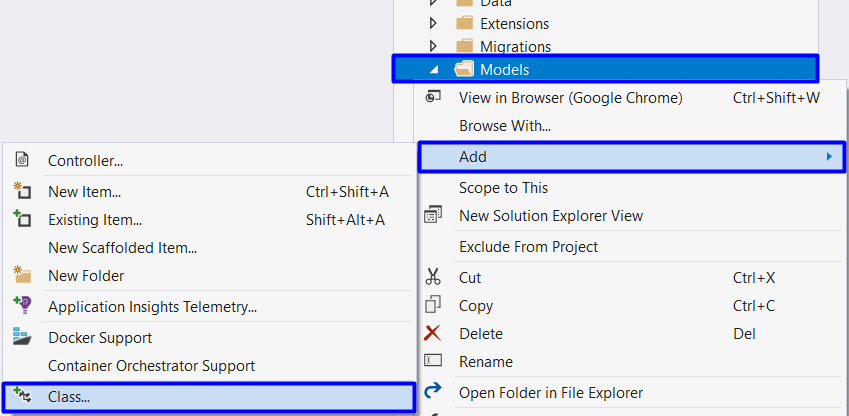
If you are using the skeleton that we provided, you have to **Update** your database.

This is the moment we can actually **create something**. Let's start with topics.

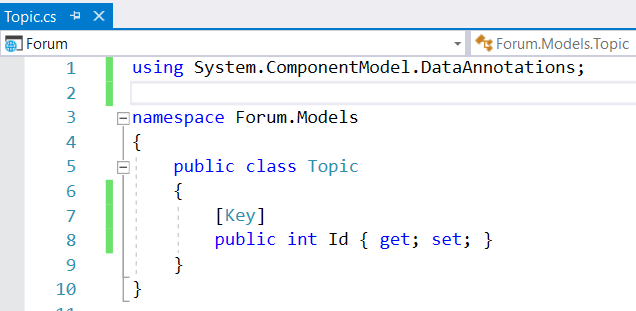
For the topics, we will need a **controller** and a **model**. In the controller, we will have action for **listing all topics**, action for **creating a topic**, action for **displaying a single topic** and actions for **editing** and **deleting topic**. For each action, we need a **view**.

## Creating the Topic Model

This is a class that will **hold information about a single topic** and will be **saved in the database** (e.g. an **entity**). So, **create a new class** in the "**Models**" folder and name it "**Topic**":



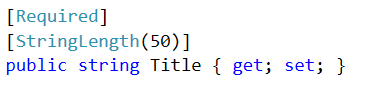
Add a public int property "**Id**" with an attribute [**Key**]:



The attribute [**Key**] specifies that this is the [primary key](https://www.techopedia.com/definition/5547/primary-key) and is used to **identify** a record in the table (think of it as an **id**).

Now we need the other properties of a topic: **Title**, **Description**, CreatedDate, LastUpdatedDate and **Author**.

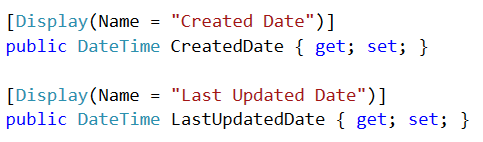
The **title** will be with a maximum length of 50 characters and will be **required**, so again, we can use an attribute to specify that:



The **description** won't have any restriction to length and it can be an **empty** **string**, so:



The Display tag gets or sets a value that is used for display in Views:



And we need an **author id**, which will be of type **string**:



An author for the topic, which will be of type ApplicationUser:



You should end up with something like this:

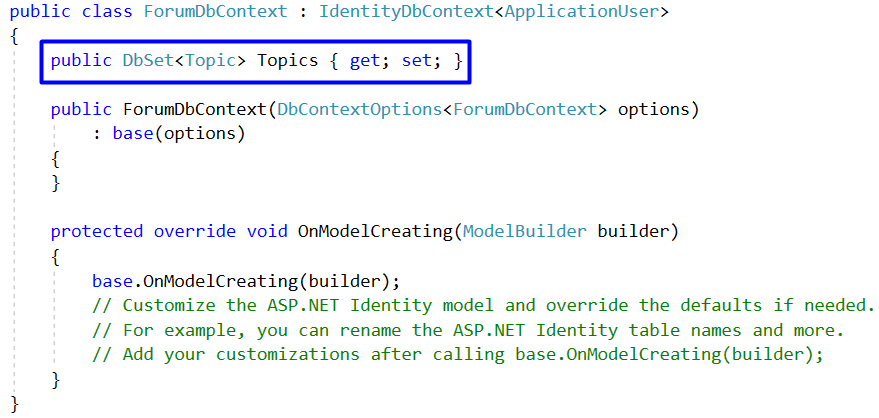


## Inserting Topic Manually

We want to test if the **entity is created properly** in the database. In order to check this, the database should know about topics. We can do this in the file "ForumDbContext.cs".

The **database context** is the layer in our application that **communicates with the database**.

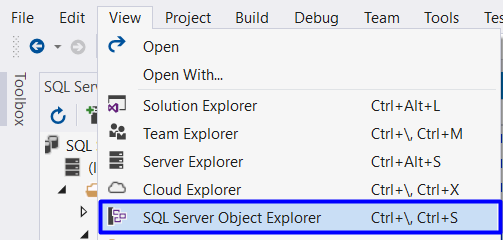
Just create a new property that is of type **public DbSet<Topic>** and name it Topics:



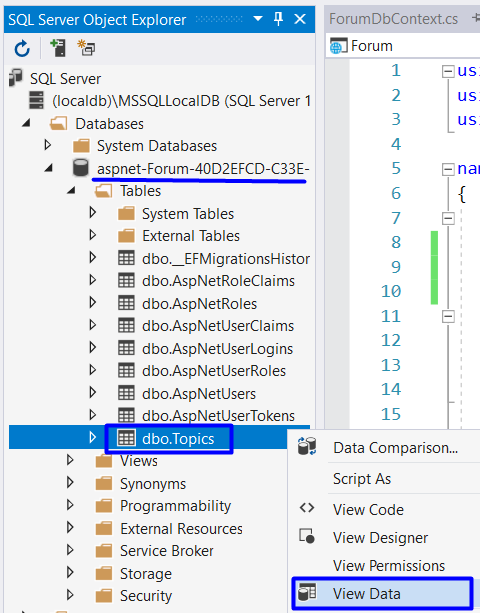
Apply another migration adding Topic.(Look section V-5)

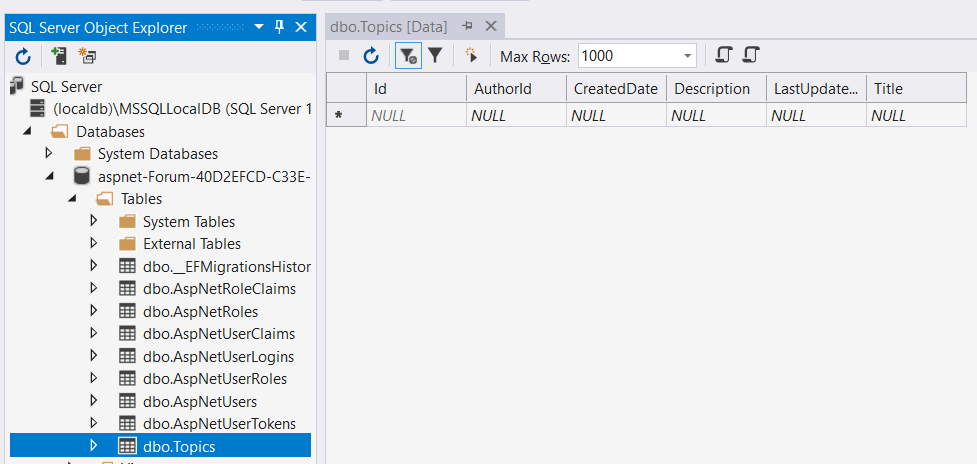
**Build the application** and then do something involving **modifying the database**, like registering a new user. After that you can see that in there is a new table called "Topics".

First open SQL Server Object Explorer:

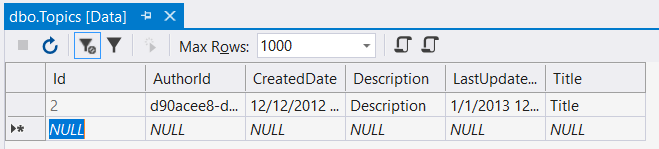


Find "aspnet-Forum…" database open Tables Folder, clickView Data and you will see:





So, try to **insert some topic**. Just click on a field and **type some information in**. Make sure you copy and paste the **GUID** of some existing user, otherwise it won't let you to insert a topic with an invalid or missing author.

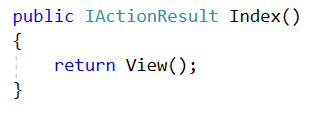


The only problem is that we don't have a place in the forum where we see the topic. For this we will need a **Topic Controller**.

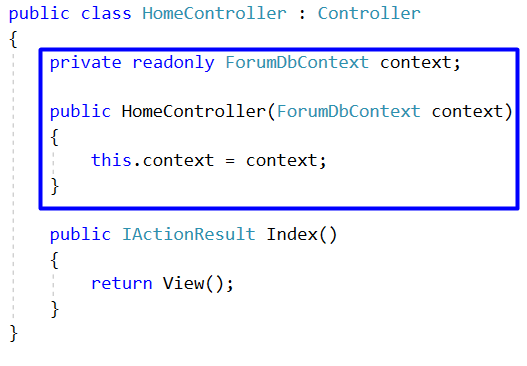
# List Topics

## Creating the Get Method

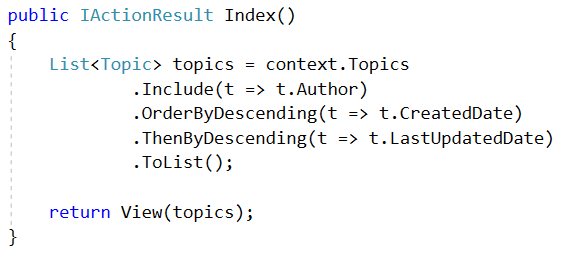
Now, go to "**Controllers/HomeController.cs"** head to the **Index()** action:



To do this, we need an instance of the class ForumDbContext. First create private readonly field of type **ForumDbContext**. **Readonly** indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. Then assign value to the context by the constructor:



Now, using **LINQ**, we can **get all topics** and then **pass them to the view**:



Notice the **".Include()**" statement. What it does is when we **materialize** the topics from the database, we need to **include information about their authors**. If we didn’t do that, the author would be left **null** and if we try to get information about the author inside the view, an **exception will be thrown**.

## Edit the Home Page View

We are just going to put a placeholder on the home page. Go to "**Views/Home/Index.cshtml**" and paste this:

|  |
| --- |
| @model IEnumerable<Topic>  @{  ViewData["Title"] = "Home Page";  }  <div class="container-fluid">  <div class="row forum-container">  <section class="col-md-9 col-sm-12 article-wrapper">  <p>  <**a** **asp-controller**="Topic" **asp-action**="Create" class="btn btn-default">Create New Topic</**a**>  </p>  @foreach (var topic in Model)  {  <article class="forum-post">  <div class="head-post">  <h2 class="post-title">  <**a** **asp-controller**="Topic" **asp-action**="Details" **asp-route-id**="@topic.Id">@topic.Title</**a**>  </h2>  <p>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Author: @topic.Author.UserName</span>  </span>  </span>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Created Date: @topic.CreatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  </span>  </p>  </div>  <div class="meta-info">  <div class="col-md-6 last-replay">  <div class="table-holder">  <div class="last-replay-inner cell">  <span class="cell">Last activity: </span>  <span class="cell replay-date">@topic.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </div>  </div>  </div>  </div>  </article>  }  </section>  </div>  </div> |

Examine the code.

The view uses a **model**. On the first line is the statement that specifies the **type of the model** (which is **IEnumerable<Topic>**). Then, every time the **@Model** or **Model** is used, it basically is a **IEnumerable<Topic>**. So, we can iterate through it with a **foreach loop** and use the properties of the topic to show them on screen.

**Notice also** that there is an **ActionLink** pointing to **Topic/Details** and Topic/Create. If you try to click it, it won’t work, because we haven't yet created neither the Details and Create actions, nor the views.

**While you are in the Index view, start the application** and see what happens:



If you start the application from somewhere else, just type "**/**" after the localhost and you should see the above screen.

If you click the title you should get an error. We will fix that in a moment.

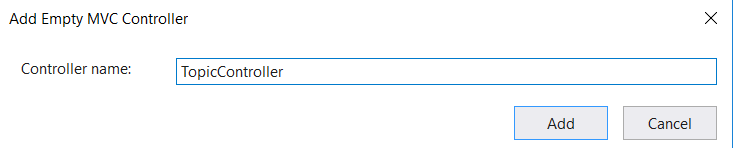
# Topic Details

## Creating the Topic Controller

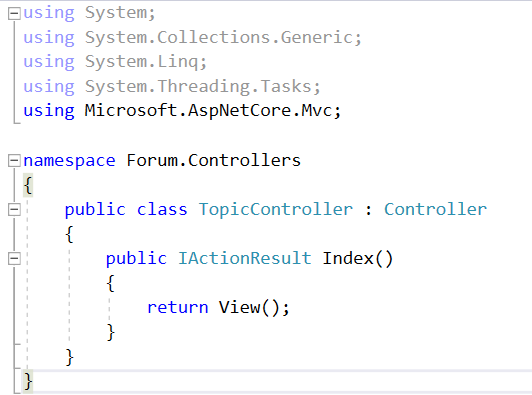
Head to the "**Controllers**" folder and create a new **Controller**:





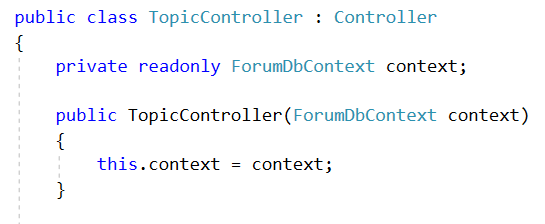


This is the result and after we will return and write some logic:



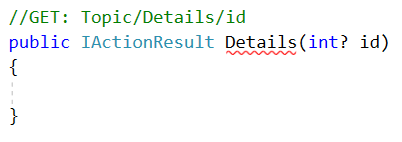
## Creating the Get Method

In the **Topic controller** now we need an instance of the database context like HomeController:

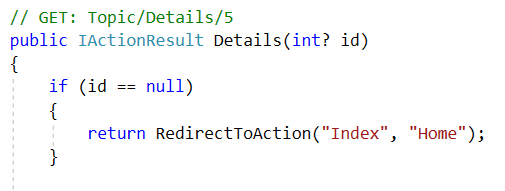


After that, create an action called "Details".

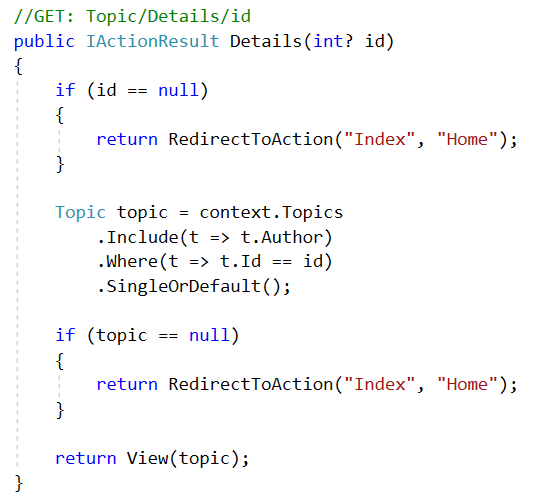
The integer parameter should be **nullable** (**int?**), e.g. it can have a null value (unlike a normal **int**), just like an object:



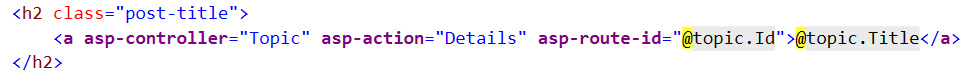
We should **check if it is null** redirect to the home view(e.g. something went wrong in the request):



and a query for the topic by its id:



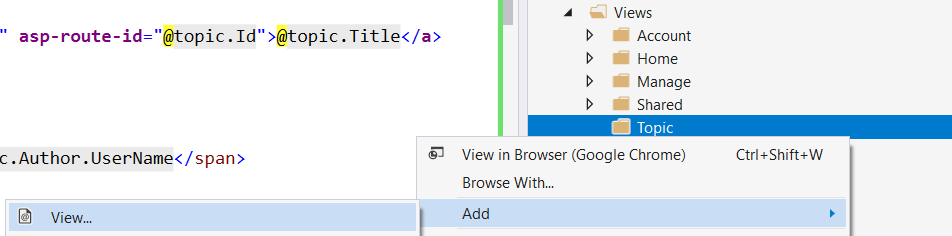
Fetch the topic using this **LINQ** query. You can see how the **id is passed** to the action inside the   
**Home/Index** view:

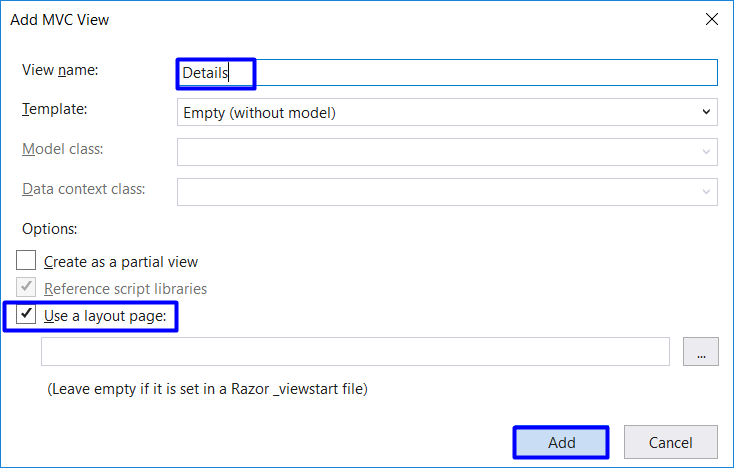


It is passed as a **parameter in the action link**.

## Creating the View

The above action gives the view an article. Now we need to show it in the browser. In Views Folder create new folder Topic and inside create new view called "Details":





Paste the following code:

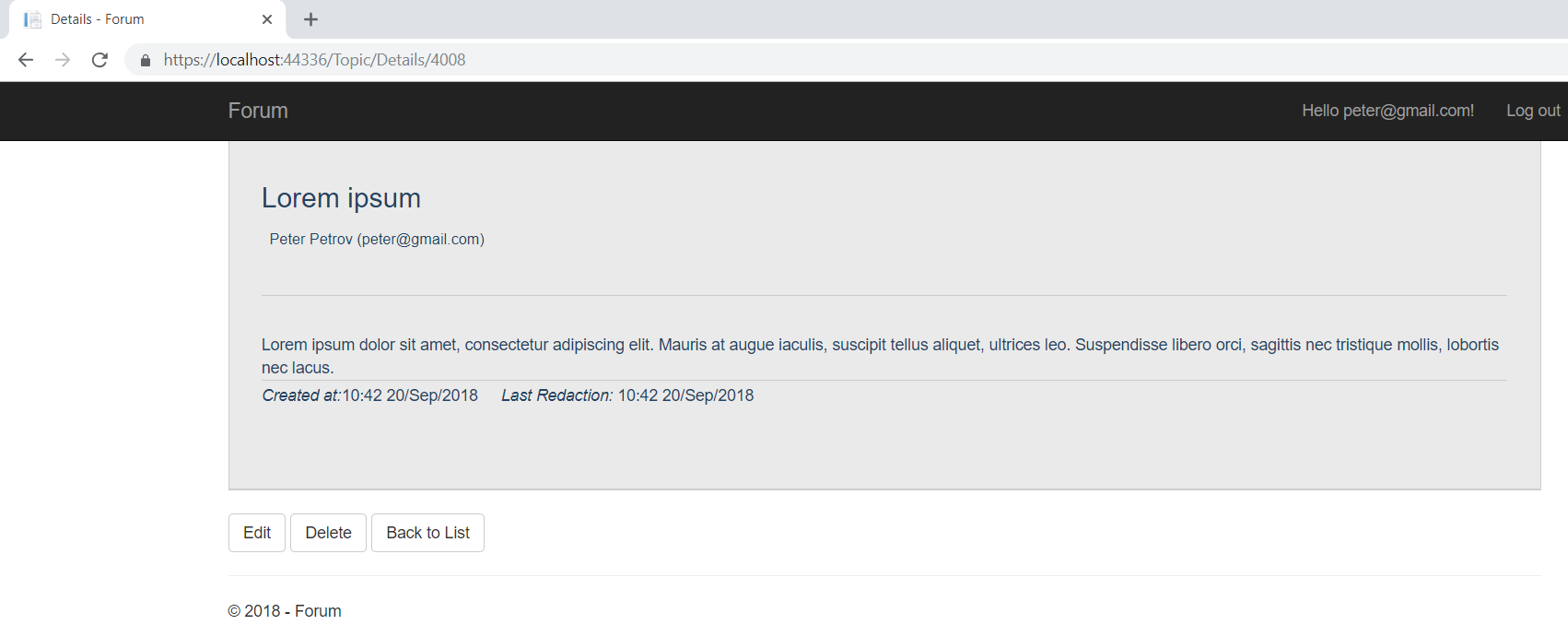
|  |
| --- |
| @model Topic  @{  ViewData["Title"] = "Details";  }  <div>  <article class="forum-post post-item highlighted-post">  <div class="head-post table-holder">  <div class="post-content">  <div class="post-title">  <header>@Model.Title</header>  <small class="user-name">@Model.Author.FullName (@Model.Author.Email)</small>  </div>  <div>  <p>@Model.Description</p>  </div>  <div class="meta-info table-holder meta-details">  <div class="table-holder">  <div class="cell">  <div class="cell">  <span class="item-wrap date-wrap">  <i class="fa fa-calendar">Created at: </i>  <span class="post-date" title="Create at">  @Model.CreatedDate.ToString("HH:mm dd/MMM/yyyy")  </span>  </span>  <span class="item-wrap category-wrap">  <i class="fa fa-exchange">Last Redaction: </i>  <span class="date-time cell" title="Last Redaction">@Model.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  </div>  </div>  </div>  </div>  </div>  </div>  </article>  </div>  <div>  <**a** **asp-action**="Edit" **asp-route-id**="@Model.Id" class="btn btn-default">Edit</**a**>  <**a** **asp-action**="Delete" **asp-route-id**="@Model.Id" class="btn btn-default">Delete</**a**>  <**a** **asp-controller**="Home" **asp-action**="Index" class="btn btn-default">Back to List</**a**>  </div> |

There are some new elements in the code.

Inspect the buttons in the last div. There are three **Action Links**.

They are created with the tag helper asp-action which creates a link, which takes as parameters the **display value** of the button (what it will show in the browser), **the action** and **the controller if needed** to which the link points.

You can now test the **Details** view:

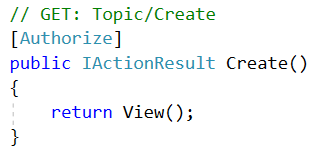


# Creating Topics

## Creating the Get Method

For topic creation, we need two actions (**get** and **post**) and a view.

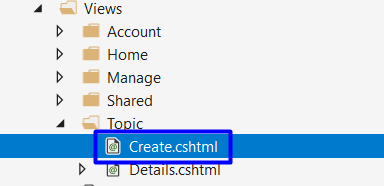
In **TopicController**, create the first action called **Create**:



We want only registered users to be able to create a topic. To do that the only thing we need is the **attribute** Authorize.

## Creating the View

Before we create the second method, lets create the view -> "Views/Topic/Create.cshtml":

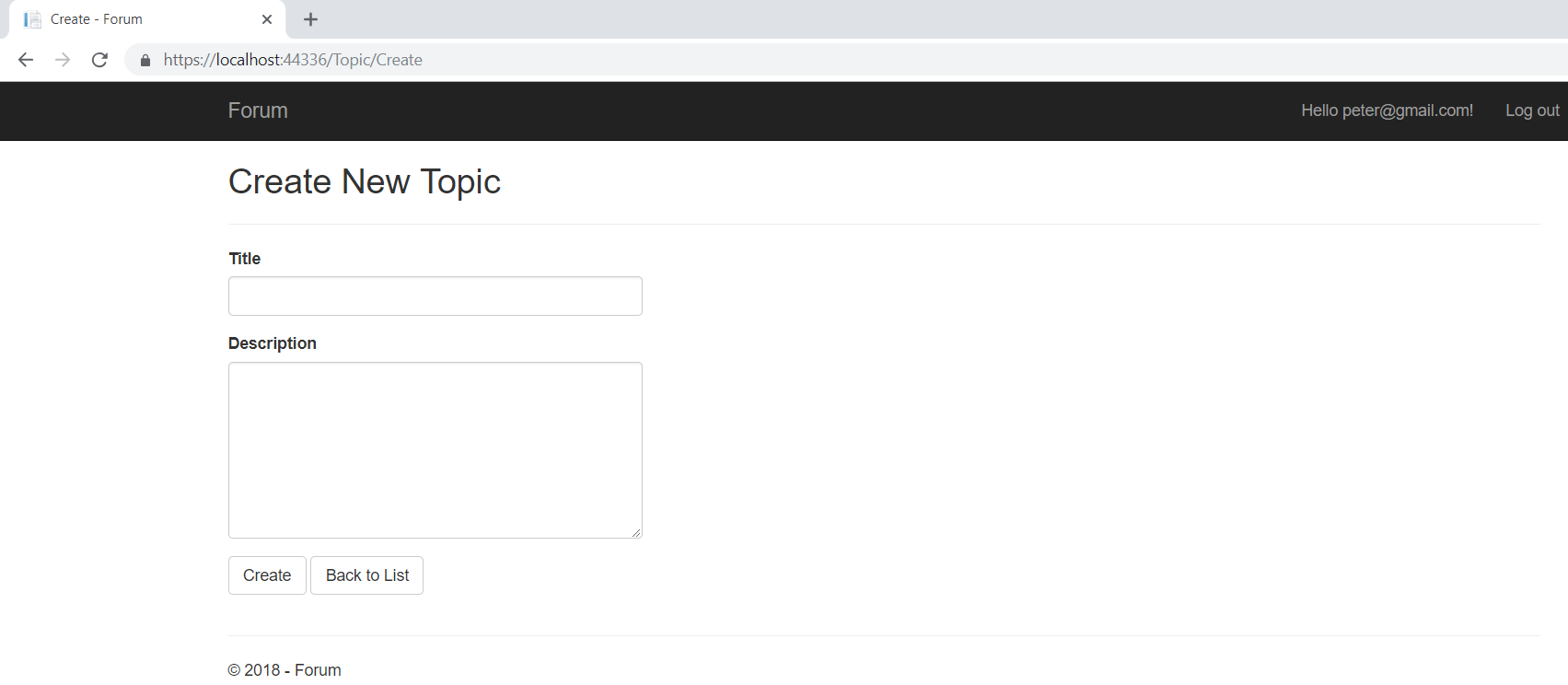


Paste the following code there:

|  |
| --- |
| @model Topic  @{  ViewData["Title"] = "Create";  }  <h2>Create New Topic</h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-action**="Create">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <div class="form-group">  <**label** **asp-for**="Title" class="control-label"></**label**>  <**input** **asp-for**="Title" class="form-control" />  <**span** **asp-validation-for**="Title" class="text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Description" class="control-label"></**label**>  <**textarea** **asp-for**="Description" class="form-control" rows="7"></**textarea**>  <**span** **asp-validation-for**="Description" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Create" class="btn btn-default" />  <**a** **asp-controller**="Home" **asp-action**="Index" class="btn btn-default">Back to List</**a**>  </div>  </**form**>  </div>  </div>  @section Scripts {  @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}  } |

This should be already familiar. It's a **view** with a single form in which there are **text box** for title, **text area** for content and **two buttons**. One of the buttons **submits the form** and the other redirects to the home of the forum.

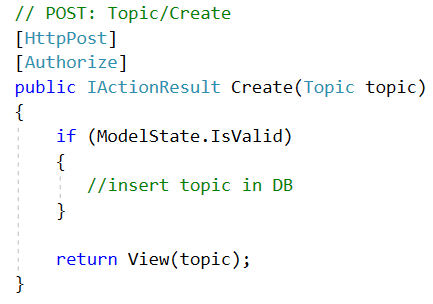
The final **view** should look something like this:



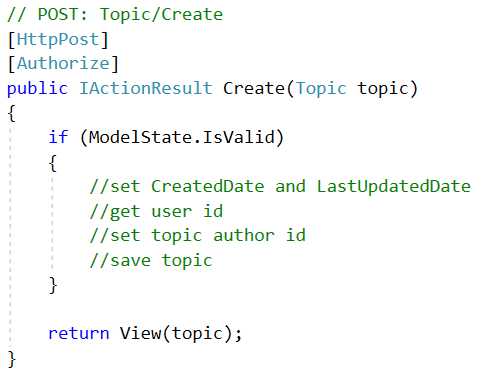
## Creating the Post Method

**Adding topics** to the database is done through the **post action method** that we mentioned earlier. Basically, once you hit the submit button you **send the information from the form to the action**.

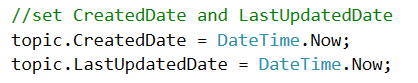
Start with method creation:



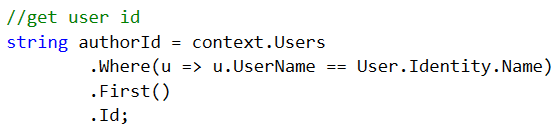
The **method receives** an **article model**. If the model is valid we can **insert it in the db**. If not, we just display the same page with the info from the model. We can chop the article insertion into smaller problems:



1. First set CreatedDate and LastUpdatedDate with the current date and time:



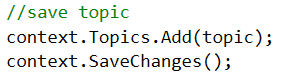
1. We can **get the author id from the db** this way:



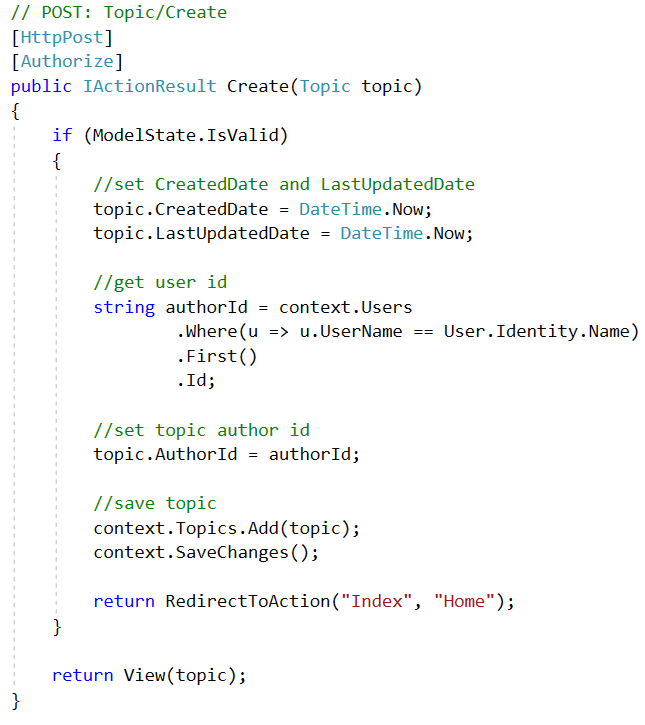
1. Now, **set the author id of the topic**:



1. And **add the topic** to the db:



In the end, the **method should look like this**:



If you now try to create a topic when you are not registered, you won't be able to.

# Deleting Topics

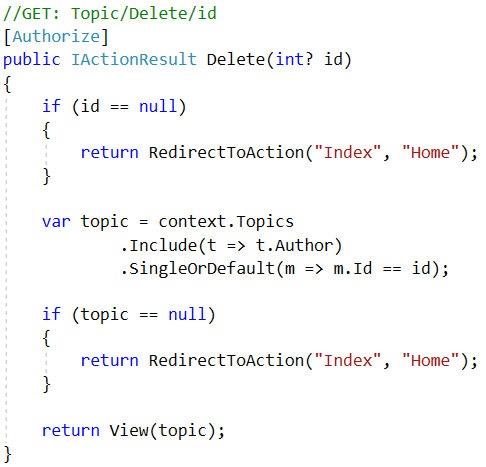
## Creating the Get Method

Deleting topics follow the same pattern as creating them. A **get** and a **post** method will be needed.

Create the **GET Method** in the **Topic Controller**:

It should **receive a topic id** so it would know **which topic to delete**. It also checks if the id is **valid**.

Then, **get the topic** from database and if it exists, pass it to the **Delete** **View**:



## Creating the View

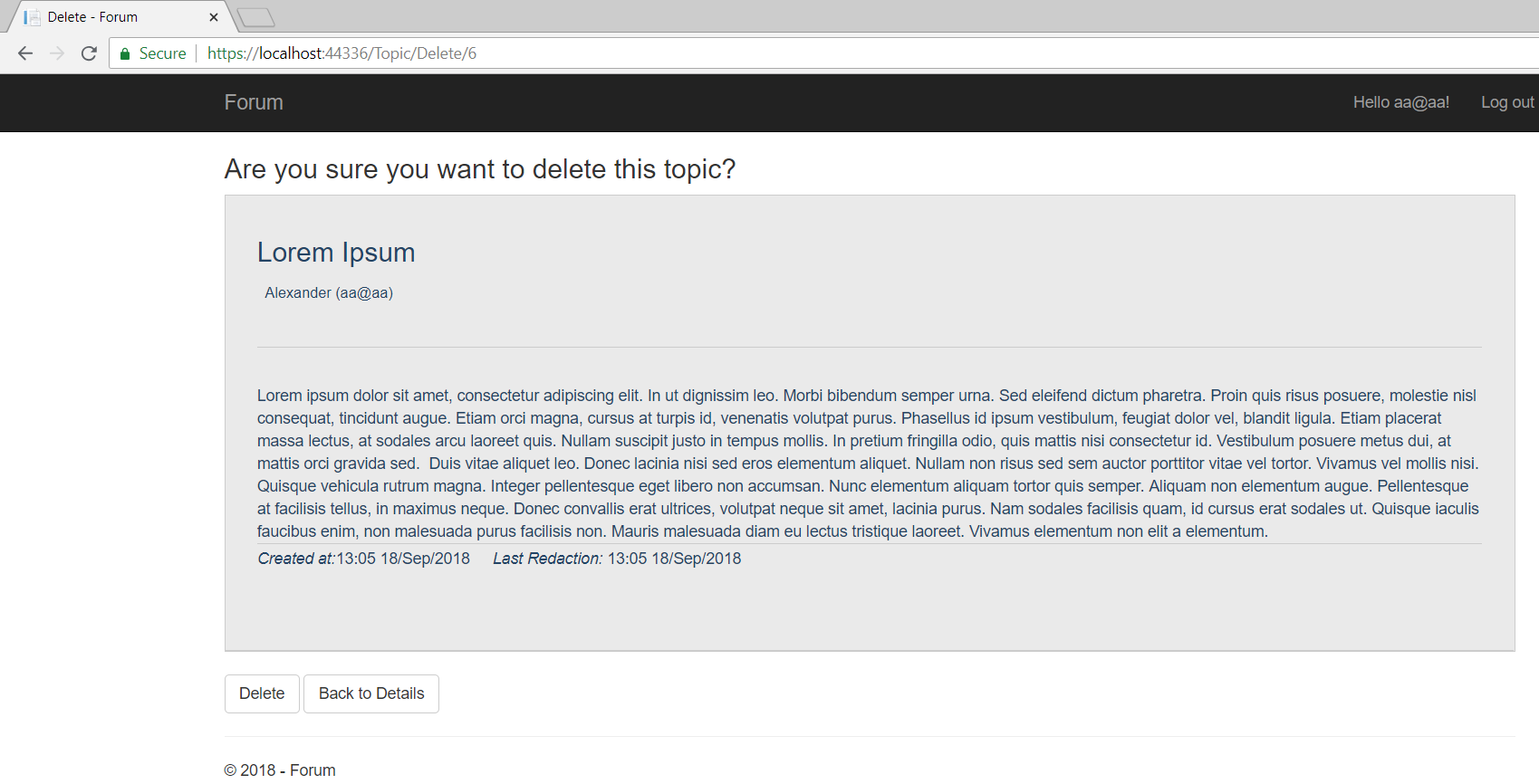
The above method directs to a delete view, but it doesn’t exist yet.

Create a "**Delete**" view:



Paste the following code:

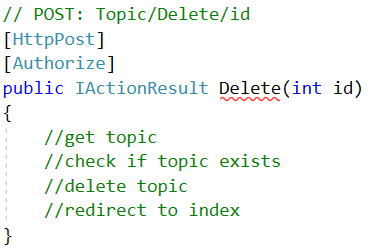
|  |
| --- |
| @model Topic  @{  ViewData["Title"] = "Delete";  }  <h3>Are you sure you want to delete this topic?</h3>  <div>  <article class="forum-post post-item highlighted-post">  <div class="head-post table-holder">  <div class="post-content">  <div class="post-title">  <header>@Model.Title</header>  <small class="user-name">@Model.Author.FullName (@Model.Author.Email)</small>  </div>  <div>  <p>@Model.Description</p>  </div>  <div class="meta-info table-holder meta-details">  <div class="table-holder">  <div class="cell">  <span class="item-wrap date-wrap">  <i class="fa fa-calendar">Created at: </i>  <span class="post-date" title="Create at">  @Model.CreatedDate.ToString("HH:mm dd/MMM/yyyy")  </span>  </span>  <span class="item-wrap category-wrap">  <i class="fa fa-exchange">Last Redaction: </i>  <span class="date-time cell" title="Last Redaction">@Model.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  </div>  </div>  </div>  </div>  </div>  </article>  </div>  <div>  <**form** **asp-action**="Delete">  <**input** **type**="hidden" **asp-for**="Id" />  <input type="submit" value="Delete" class="btn btn-default" />  <**a** **asp-action**="Details" **asp-route-id**="@Model.Id" class="btn btn-default">Back to Details</**a**>  </**form**>  </div> |



## Creating the Post Method

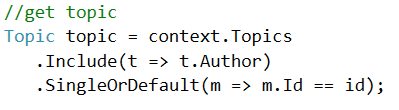
When you click on the "**Delete**" button (this is actually the submit of the form) you are sent to the **POST Method** in the controller.

Let's create it:

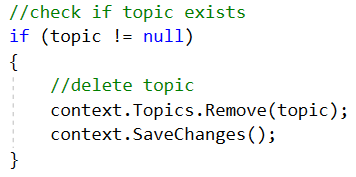


Make sure you add the attribute [**HttpPost**].

1. Now, let’s **get the topic** form the database:



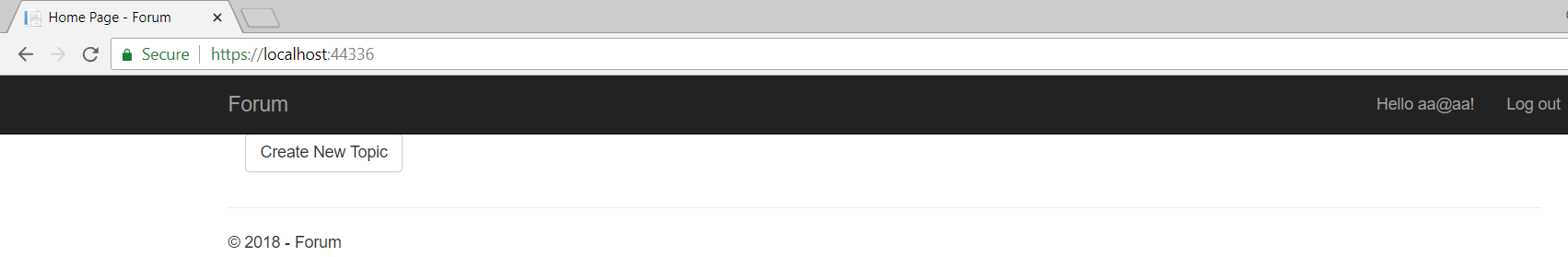
1. **Check** if the topic exists and **Delete** the topic:



1. And, **redirect** to the index page:



Test if you can delete a topic:

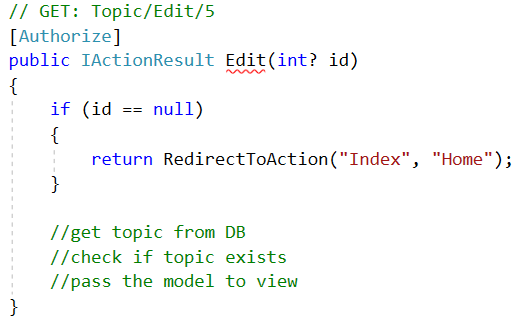


# Editing Topics

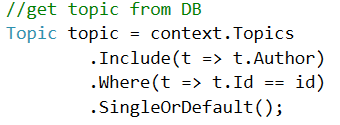
## Creating the Get Method

Again, editing topics follow the same pattern. A **get** and a **post** method.

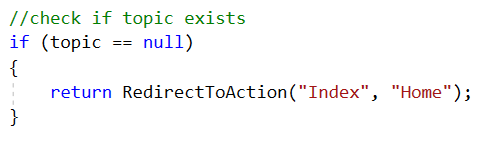
Create the **GET Method**:



1. **Get** the topic:



1. **Check** if it exists:



1. **Pass** it to the view:

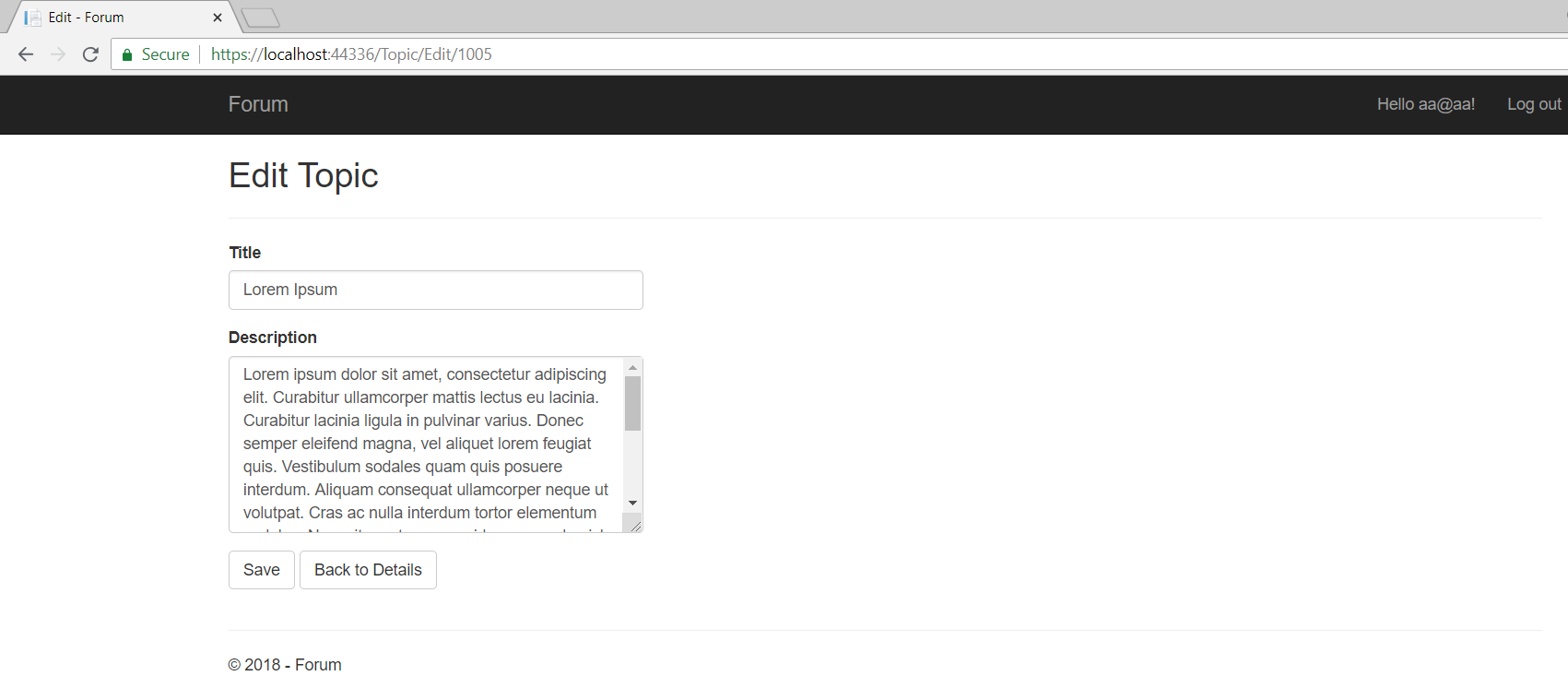


## Creating the View

Create another view "Edit":

|  |
| --- |
| @model Topic  @{  ViewData["Title"] = "Edit";  }  <h2>Edit Topic</h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-action**="Edit">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <**input** **type**="hidden" **asp-for**="Id" />  <div class="form-group">  <**label** **asp-for**="Title" class="control-label"></**label**>  <**input** **asp-for**="Title" class="form-control" />  <**span** **asp-validation-for**="Title" class="text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Description" class="control-label"></**label**>  <**textarea** **asp-for**="Description" class="form-control" rows="7"> </**textarea**>  <**span** **asp-validation-for**="Description" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Save" class="btn btn-default" />  <**a** **asp-action**="Details" **asp-route-id**="@Model.Id" class="btn btn-default">Back to Details</**a**>  </div>  </**form**>  </div>  </div>  @section Scripts {  @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}  } |

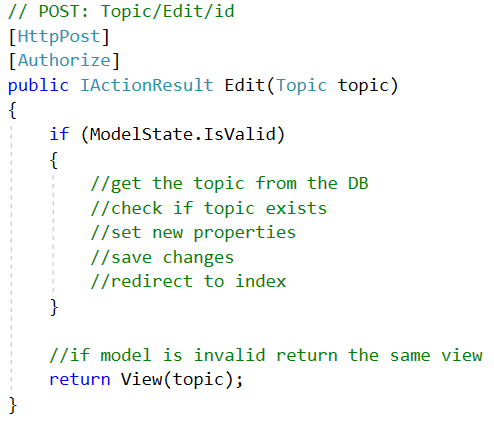
It should result in this:



## Creating the Post Method

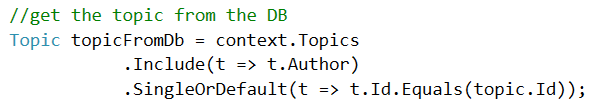
We can also use the Topic model In the **Post Method**. This is done **through method parameters**:

**Create the method**:

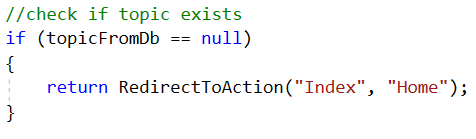


To implement the method:

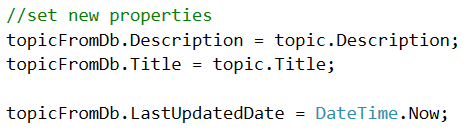
1. First, **get the topic** from the database



1. **Check** if it exists



1. Next, **set topic's** new **values**



1. **Save changes**



1. **Redirect** to the **index page**



To this moment we have done the basic functionality of the forum, which is:

1. Listing topics.
2. Creating topics.
3. Editing topics.
4. Deleting topics.

In the next couple sections below, we will add some additional functionality. For every topic we will be able to add comments, edit and delete them if we want to.

*"No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first." - Andy Hunt*

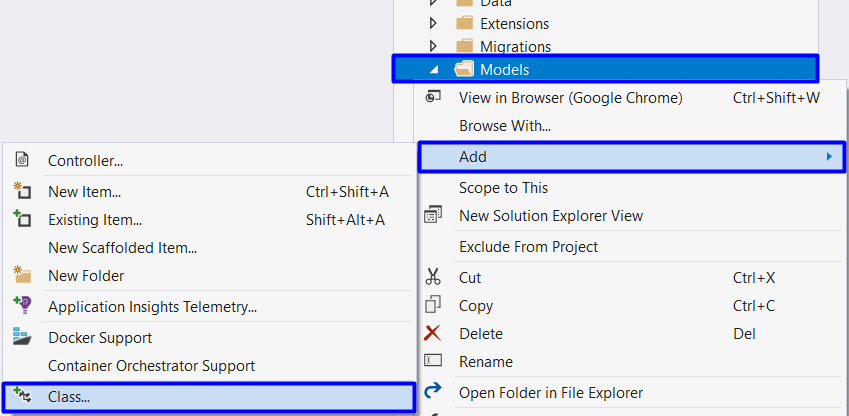
# Creating Comment Entities

Let's start with comments.

Just like topics, the comments will also need a **controller** and a **model**. In the controller, we will have action for **creating a comment**, and actions for **editing** and **deleting comment**. For each action, we need a **view**.

## Creating the Comment Model

This is a class that will **hold information about a single comment** and will be **saved in the database** (e.g. an **entity**). So, **create a new class** in the "**Models**" folder and name it "**Comment**":

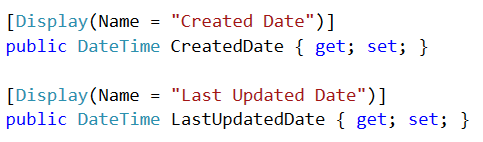


Now we need the properties of a comment: **CommentId,** **Description**, **CreatedDate**, **LastUpdatedDate**, **Topic** and **Author**.

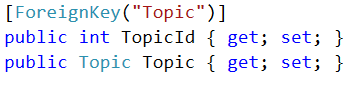
The **description** won't have any restriction to length and it can be an **empty** **string**, so:



The Display tag gets or sets a value that is used for display in Views:



With this we create relation between topic and comments



And we need an **author id**, which will be of type **string**:



An author for the topic, which will be of type ApplicationUser:



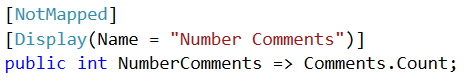
You should end up with something like this:

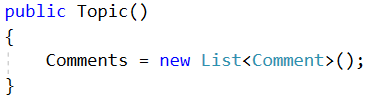


Before we continue we need to go to the Topic.cs model.

1. Add property "Comments" of type List<Comment>:



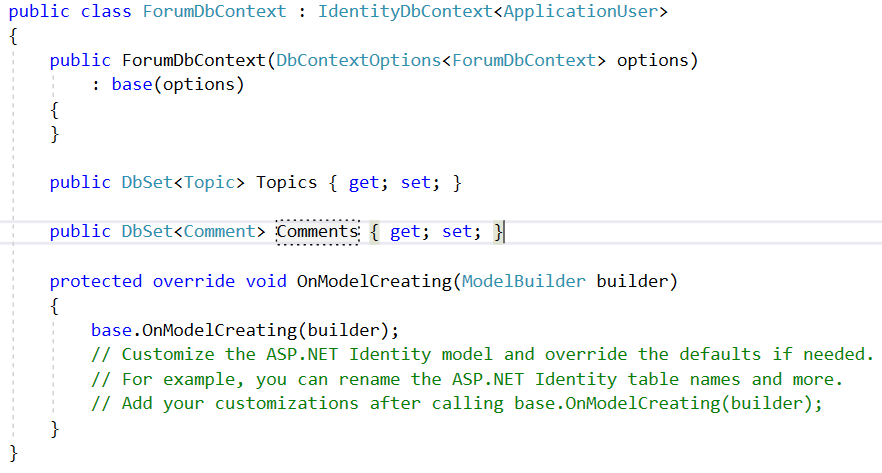
1. Create property NumberComments. Using the **attribute NotMapped** for don't map property in the database. We use that property to show in the view how comments have every topic. 
2. And finally add constructor who initialize the collection:



## Inserting Comments Manually

Like Topic Model we must add **DbSet** for **Comments**. We can do this in the file "ForumDbContext.cs".

Just create a new property that is of type **public DbSet<Comment>** and name it Comments:



Apply another migration adding Comment.(Look section V-5)

**Build the application** and then do something involving **modifying the database**, like creating a new topic. After that you can see that in there is a new table called "Comments".

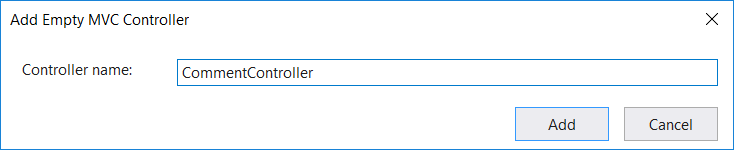
The only problem is that we don't have a place in the forum where we see the comments. For this we will need a **Comment Controller**.

## Creating the Comment Controller

Head to the "**Controllers**" folder and create a new **Controller**:







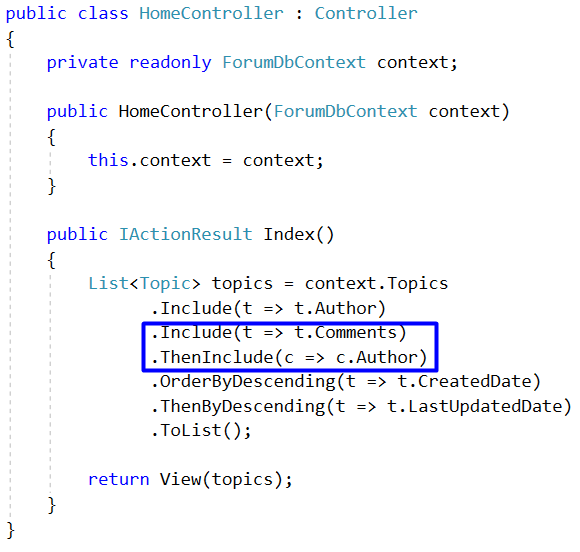
We will leave this as it is for now and come back later to write some logic:



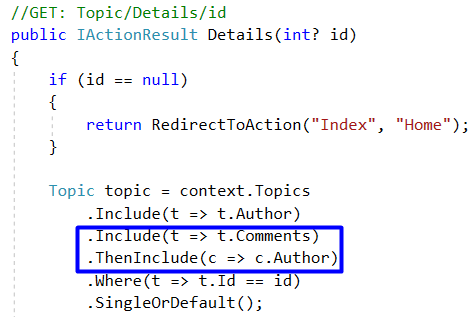
# List Comments

## Creating the Get Method

Now, go to "**Controllers/HomeController.cs"** head to the **Index()** action and we need to refactor the method. We need to **include** **Comments** and an **Author** for every comment. If we don’t do that, the comments and comment's author would be left **null** and if we try to get information about him inside the view, an **exception will be thrown**.



After that also go to the **"TopicController.cs"** **-> Details** method and do the same thing:



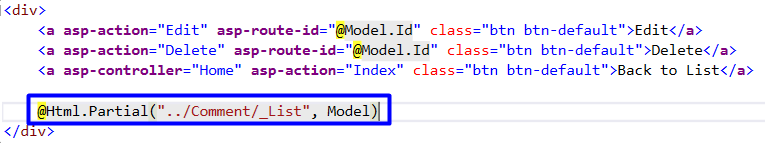
## Creating the View

The above action gives the view comments. Now we need to show it in the browser. In Views Folder create new folder "**Comment"** and inside create new view called "\_List".

Paste the following code:

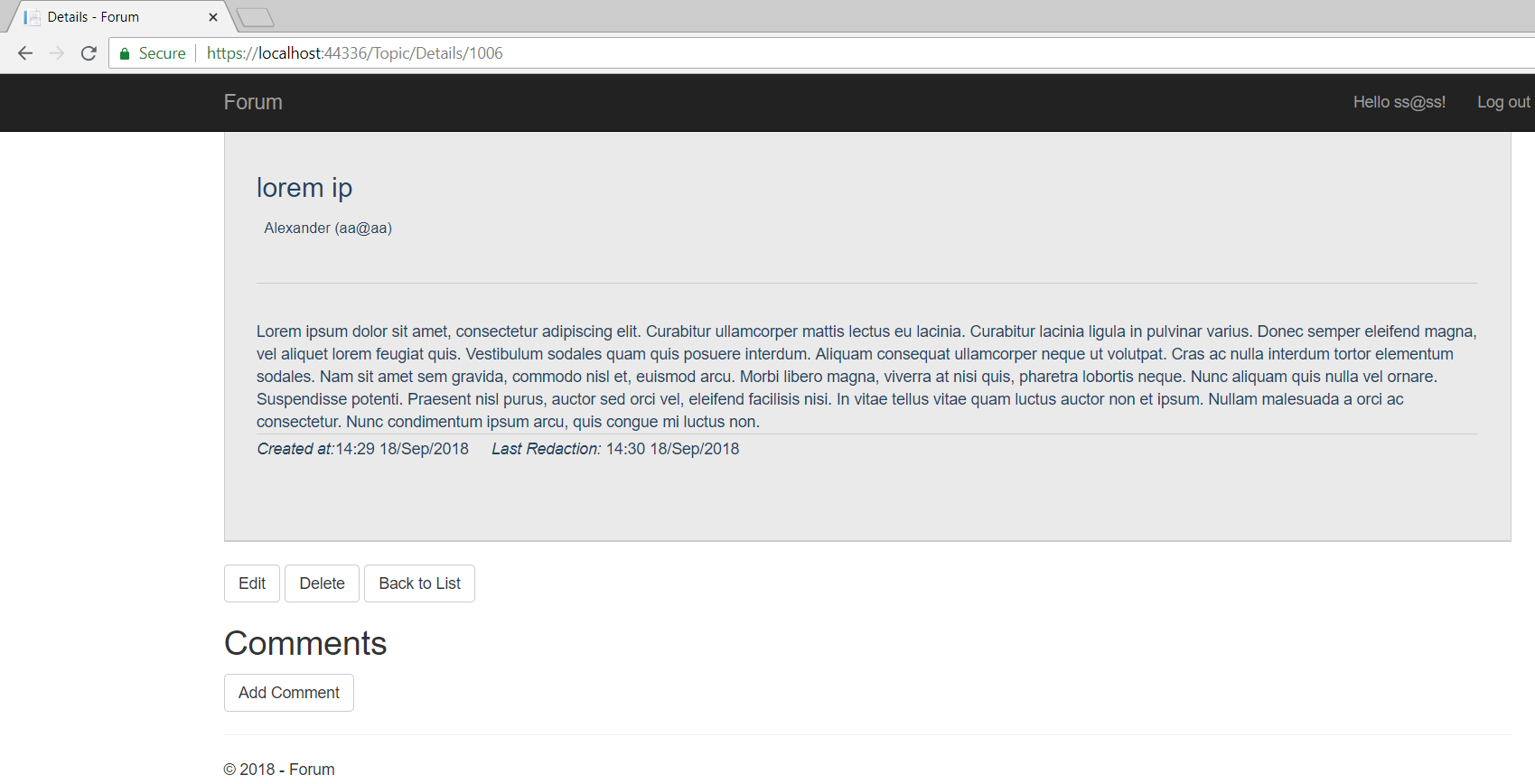
|  |
| --- |
| @model Topic  @{  ViewData["Title"] = "Details";  }  <h2>Comments</h2>  <p>  <**a** **asp-controller**="Comment" **asp-action**="Create" class="btn btn-default">Add Comment</**a**>  </p>  <div>  @foreach (var comment in Model.Comments)  {  <article class="forum-post post-item highlighted-post">  <div class="head-post table-holder">  <div class="post-content">  <div>  <p>@comment.Description</p>  </div>  <div class="meta-info table-holder meta-details">  <div class="table-holder">  <div class="row">  <div class="cell">  <span class="item-wrap date-wrap">  <i class="fa fa-calendar">Created at: </i>  <span class="post-date" title="Create at">  @comment.CreatedDate.ToString("HH:mm dd/MMM/yyyy")  </span>  </span>  <span class="item-wrap category-wrap">  <i class="fa fa-exchange">Last Redaction: </i>  <span class="date-time cell" title="Last Redaction">@comment.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  <span class="item-wrap">  <i class="fa fa-calendar">Author: </i>  <span class="post-date" title="Author">  @comment.Author.UserName  </span>  </span>  <span class="item-wrap">  <a href="/Topic/Details/@ViewContext.RouteData.Values["id"]/Comment/Edit/@comment.CommentId" class="btn btn-default">Edit</a>  <a href="/Topic/Details/@ViewContext.RouteData.Values["id"]/Comment/Delete/@comment.CommentId" class="btn btn-default">Delete</a>  </span>  </div>  </div>  </div>  </div>  </div>  </div>  </article>  }  </div> |

Now we go to the Topic/Details.cshtml view and add in last div tag:



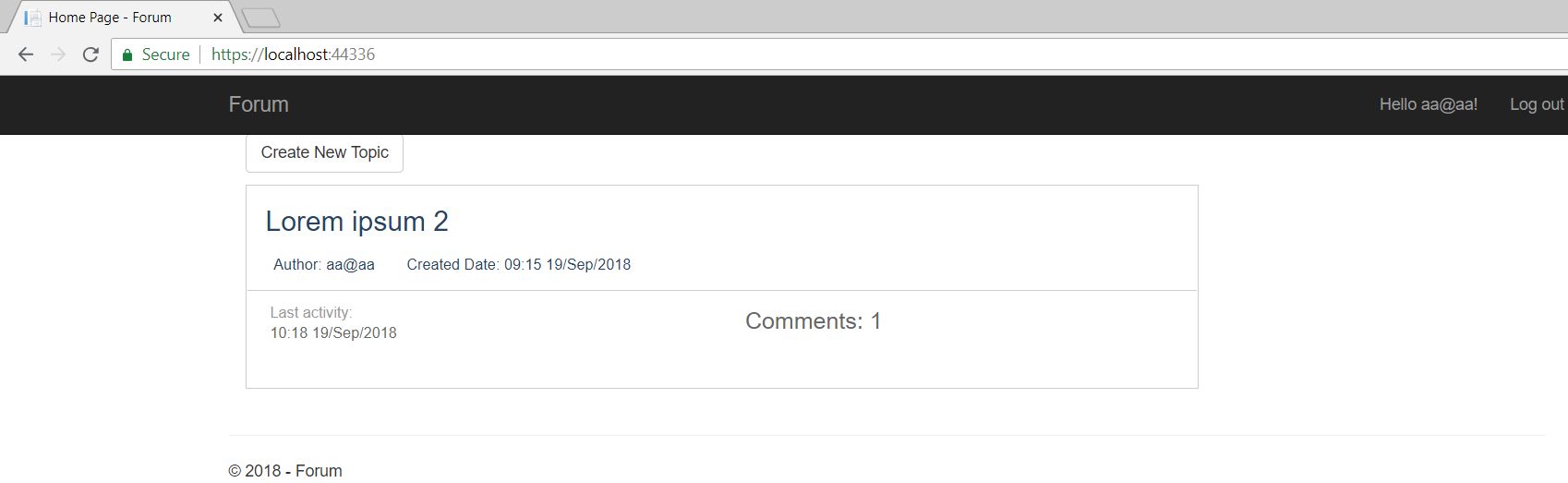
This renders the partial view (a view inside another view) and the second parameter passes the model into the partial view.

You can now test the **Details** view, it should look like this:



We need to show how many comments we have for every topic in home page. To do that go to "Home/Index.cshtml" and add a div tag to visualize comments and their number. Your code should look like this:



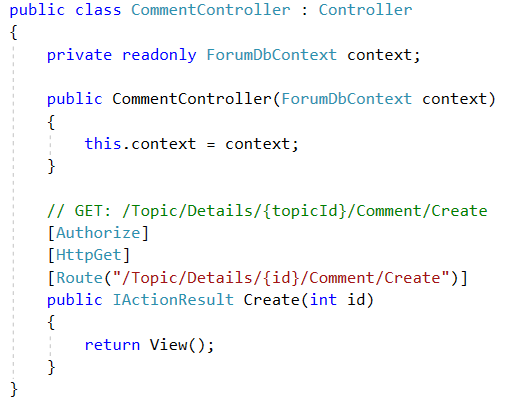


# Creating Comments

## Creating the Get Method

Just like creating a topic, in order to create comments, we need an instance of the class **ForumDbContext**, two actions (**get** and **post**) and a view.

In **CommentController**, create context property and the first action called **Create**:



We use Route attribute, because we want to explicitly use the given route. It automatically sets the "{id}" with the current **topic id**.

## Creating the View

Now lets create the view -> "Views/Comment/Create.cshtml":

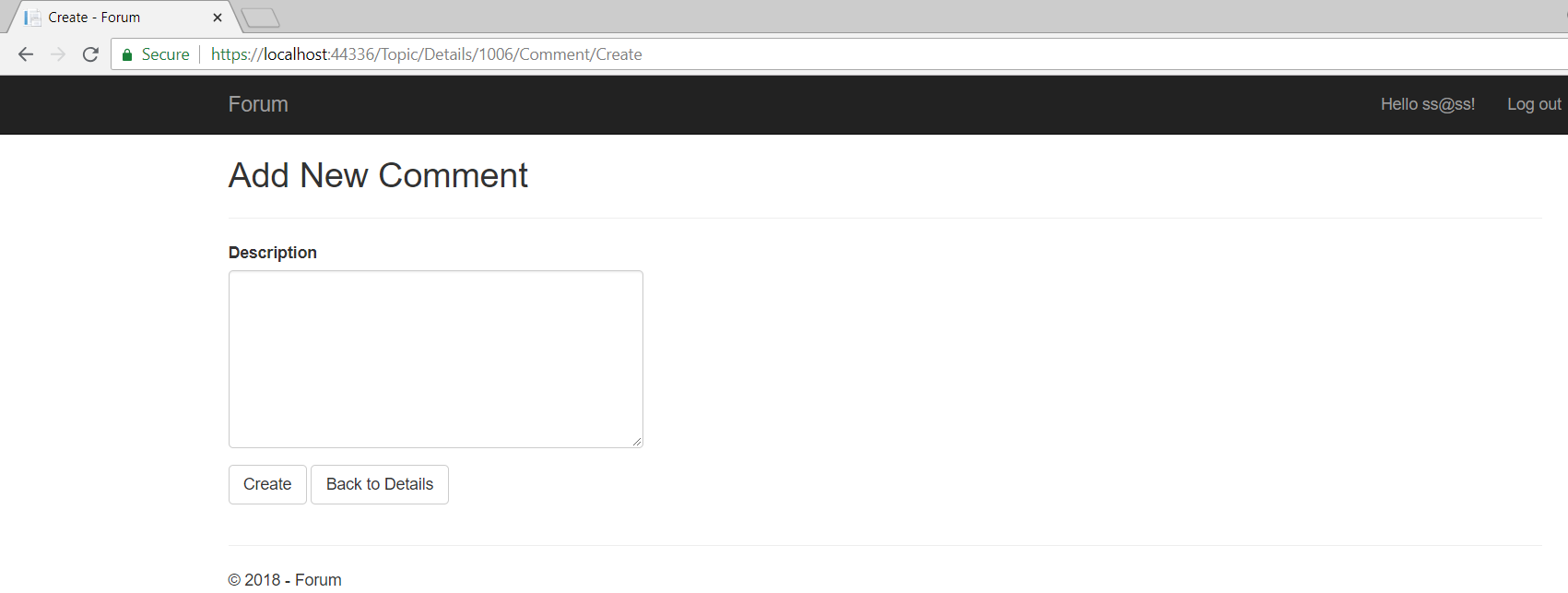


Paste the following code there:

|  |
| --- |
| @model Comment  @{  ViewData["Title"] = "Create";  }  <h2>Add New Comment</h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-controller**="Comment" **asp-action**="Create">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <div class="form-group">  <**label** **asp-for**="Description" class="control-label"></**label**>  <**textarea** **asp-for**="Description" class="form-control" rows="7"> </**textarea**>  <**span** **asp-validation-for**="Description" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Create" class="btn btn-default" />  <a href="/Topic/Details/@ViewContext.RouteData.Values["id"]" class="btn btn-default">Back to Details</a>  </div>  </**form**>  </div>  </div> |

We should already be familiar with this. It's a **view** with a single form in which there are **text area** for description and **two buttons**. One of the buttons **submits the form** and the other redirects to the details of the topic.

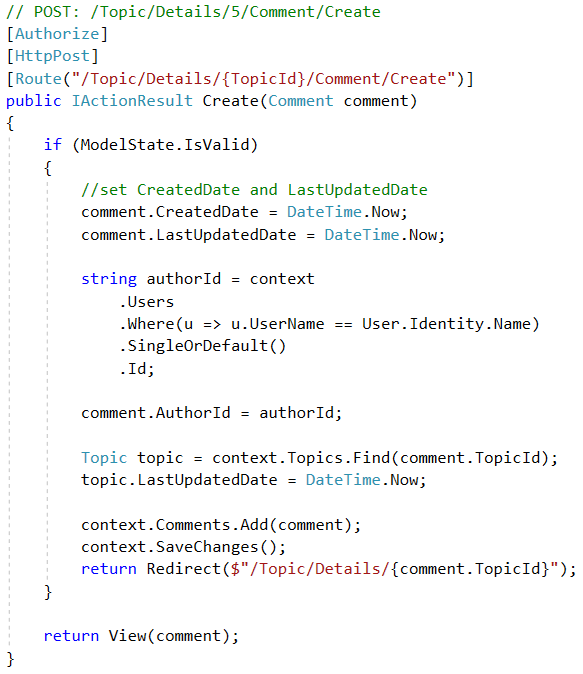
The final **view** should look something like this:



## Creating the Post Method

First thing we need to do is to set CreatedDate and LastUpdatedDate with the current date and time. Second get the user id from the db whose **name is equal** to the name of the current **logged in user** and set it on **comment's author**. We are almost ready… Now we need to get the topic from the db and set LastUpdatedDate with current date. Last thing is to **add** the comment in the database, save the changes and redirect to the **topic**.

In the end, the **method should look like this**:

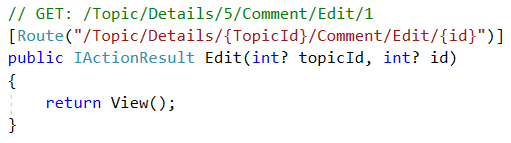


# Editing Comments

## Creating the Get Method

Again, editing a comment follows the same pattern. A **get** and a **post** method. In **"CommentController.cs"**

Create the **GET Method**:



Check if comment id is null and if it is: redirect to topic details. Then get the comment from db by id. Include Author, Topic and Topic's Author. If Comment is null redirect to topic details, else show edit view. The last state of the method should look like this:



## Creating the View

Create another view "Edit" in **Comment** folder and paste this code:

|  |
| --- |
| @model Comment  @{  ViewData["Title"] = "Edit";  }  <h2>Edit Comment</h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-controller**="Comment" **asp-action**="Edit">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <**input** **type**="hidden" **asp-for**="CommentId" />  <**input** **type**="hidden" **asp-for**="TopicId" />  <div class="form-group">  <**label** **asp-for**="Description" class="control-label"></**label**>  <**textarea** **asp-for**="Description" class="form-control" rows="7"> </**textarea**>  <**span** **asp-validation-for**="Description" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Save" class="btn btn-default" />  <a href="/Topic/Details/@ViewContext.RouteData.Values["TopicId"]" class="btn btn-default">Back to Details</a>  </div>  </**form**>  </div>  </div>  @section Scripts {  @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}  } |

It should result in this:



## Creating the Post Method

We can also use the view model In the **Post Method**. This is done **through method parameters**:

**Create the method**:



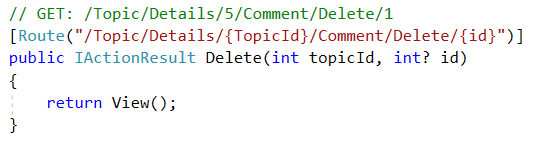
To implement the method first we have to get the **comment** from the database and **include** the **Author**. Check if comment is **null**, **redirect** to **topic** **details**. If comment is not null, **update** **properties** and set topic's LastUpdatedDate with **current date and save changes**. Then redirect to topic details. At the end it should look like this:



# Deleting Comments

## Create the Get Method

The GET Method for Delete is the same as the GET Method for Edit, except for the route as we will see below.



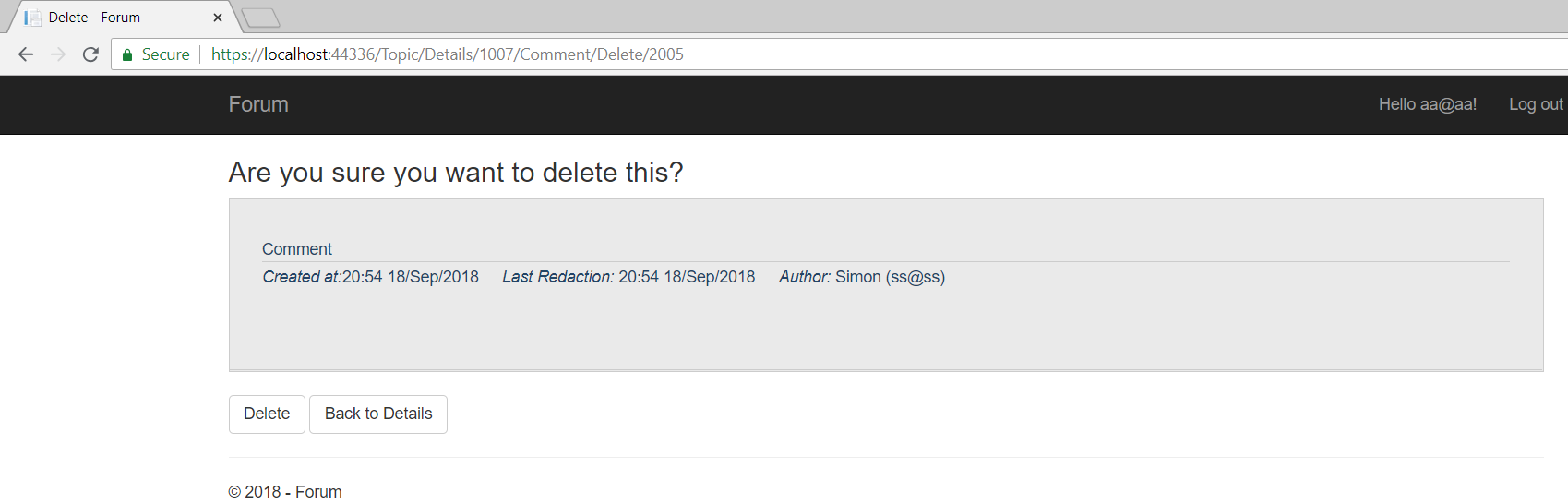
Check if comment **id** is **null** and if it is: redirect to topic details. Then get the comment from db by **id**. Include **Author**, **Topic** and **Topic's Author**. If Comment is **null** redirect to topic details, else show edit view. The final state of the method should look like this:



## Create the View

Create a "**Delete**" view. Paste the following code:

|  |
| --- |
| @model Comment  @{  ViewData["Title"] = "Delete";  }  <h3>Are you sure you want to delete this?</h3>  <div>  <article class="forum-post post-item highlighted-post">  <div class="head-post table-holder">  <div class="post-content">  <div>  <p>@Model.Description</p>  </div>  <div class="meta-info table-holder meta-details">  <div class="table-holder">  <div class="cell">  <div class="cell">  <span class="item-wrap date-wrap">  <i class="fa fa-calendar">Created at: </i>  <span class="post-date" title="Create at">  @Model.CreatedDate.ToString("HH:mm dd/MMM/yyyy")  </span>  </span>  <span class="item-wrap category-wrap">  <i class="fa fa-exchange">Last Redaction: </i>  <span class="date-time cell" title="Last Redaction">@Model.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  <span class="item-wrap category-wrap">  <i class="fa fa-folder-open">Author: </i>  <span class="cell" title="Author">  @Model.Author.FullName (@Model.Author.Email)  </span>  </span>  </div>  </div>  </div>  </div>  </div>  </div>  </article>  </div>  <div>  <**form** **asp-action**="Delete">  <**input** **type**="hidden" **asp-for**="CommentId" />  <input type="submit" value="Delete" class="btn btn-default" />  <a href="/Topic/Details/@ViewContext.RouteData.Values["TopicId"]" class="btn btn-default">Back to Details</a>  </**form**>  </div> |



## Create the Post Method

When you click on the "**Delete**" button (this is actually the submit of the form) you are sent to the **POST Method** in the controller.

Let's create it. Get the comment from the database, check if the comment **is not** **null**, then get the topic from db and update LastUpdatedDate, remove the comment and save changes. Lastly, redirect to topic details:



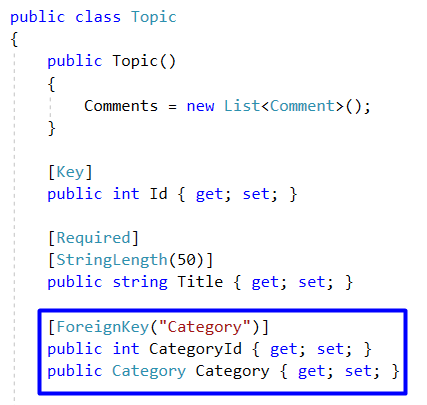
That’s all for the topics and related to them comments. Now we have to proceed with the implementation  
of the categories and the functionality required to handle clients interactions.

# Creating Category Entity

## Creating the Category Model

The class will **hold information about a single category** and will be **saved in the database** (e.g. an **entity**). So, **create a new class** in the "**Models**" folder and name it "**Category**". Now we need the properties of a category: **Id** with **Key** attribute**,** **Name** which is required and will be displayed in the view as **"Category Name"**, **Author** property of type ApplicationUser and **AuthorId** which have Foreign Key for the **Author**, **Topics** collection which collects all topics for current category, and **NumberTopics** which shows how many topics the category has (we use **NotMapped** attribute for don't excluded from database). And lastly we need constructor to initialize the collection. The entity should look like this:

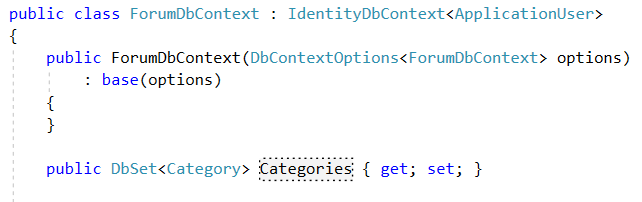


To create **relation** between the **categories** and the **topics** we have to add **CategoryId** in the **topic** entity.  


## Inserting Category Manually

Like Topic Model we must add **DbSet** for **Categories**. We can do this in the file "Data/ForumDbContext.cs".

Just create a new property that is of type **public DbSet<Category>** and name it Categories:



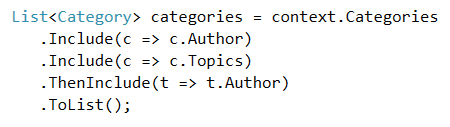
Apply another migration adding Categories.(Look section V-5)  
Find the newly created table in the database called Categories and add a Category.

You might face a **problem** **updating** **database** if you have topic records, so try to **delete** them first.

# List Categories

## Modifying Home Controller Index Action

Now, go to "**Controllers/HomeController.cs"** head to the **Index()** action, we need to refactor the method. Here we need to create new variable which gets from the database all categories, **include** Author and all **Topics** with their **Authors**. If we don’t do that, the topics and topic's author would be left **null** and if we try to get information about them inside the view, an **exception will be thrown**.



So we have both collections **Categories** and **Topics**, you might be wondering how we send them to the view.  
We want to display all the categories aside like a side bar. Using the side bar we can easily navigate through the categories and manage them.

So we'll pass the categories collection using **ViewData** and **Topics** will be send as a **model** to the view.



## Modify Home/Index View

Now we must modify the view to render both collections.

It should look something like this.

|  |
| --- |
| @model IEnumerable<Topic>  @{  ViewData["Title"] = "Index";  List<Category> categories = (List<Category>)ViewData["Categories"];  }  <div class="container-fluid">  <div class="row forum-container">  <aside class="col-md-3 col-sm-12 forum-sidebar">  <div class="category-wrapper">  <p>  <**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-default">Create New Category</**a**>  </p>  <h3>Categories</h3>  <ul>  <li>  <**a** **asp-controller**="Home" **asp-action**="Index">All Topics</**a**>  </li>  @foreach (var category in categories)  {  <li>  <**a** **asp-controller**="Category" **asp-action**="Details" **asp-route-id**="@category.Id" class="">@category.Name</**a**>  <span>(@category.NumberTopics)</span>  </li>  }  </ul>  </div>  </aside>  <section class="col-md-9 col-sm-12 article-wrapper">  <p>  <**a** **asp-controller**="Topic" **asp-action**="Create" class="btn btn-default">Create New Topic</**a**>  </p>  @foreach (var topic in Model)  {  <article class="forum-post">  <div class="head-post">  <h2 class="post-title">  <**a** **asp-controller**="Topic" **asp-action**="Details" **asp-route-id**="@topic.Id">@topic.Title</**a**>  </h2>  <p>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Author: @topic.Author.UserName</span>  </span>  </span>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Created Date: @topic.CreatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  </span>  <span class="user-wrap">  <span class="cell">  Category: <**a** **asp-controller**="Category" **asp-action**="Details" **asp-route-id**="@topic.CategoryId">@topic.Category.Name</**a**>  </span>  </span>  </p>  </div>  <div class="meta-info">  <div class="col-md-6 last-replay">  <div class="table-holder">  <div class="last-replay-inner cell">  <span class="cell">Last activity: </span>  <span class="cell replay-date">@topic.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </div>  </div>  <div class="col-md-6 stats-holder">  <span class="item-wrap">  <span class="cell">Comments: </span>  <span class="cell">@topic.NumberComments</span>  </span>  </div>  </div>  </div>  </article>  }  </section>  </div>  </div> |

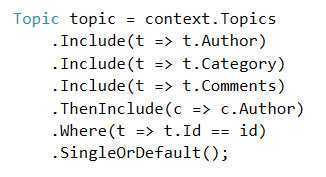
# Updating Topic Views and Controller

Now when we already have at least one manually inserted category, we want to make use of it. We are going to update the views and the functionality of the topics to make them related with the categories even in the UI.

## Update Topic Controller

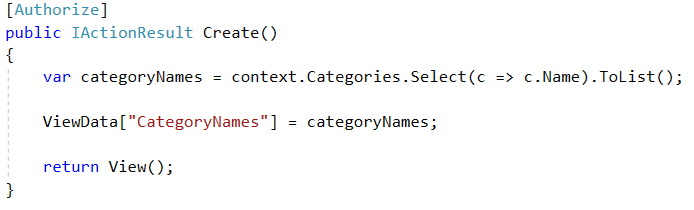
### Details Action

The task here is simple just edit the **LINQ** query to include the categories to the topic object



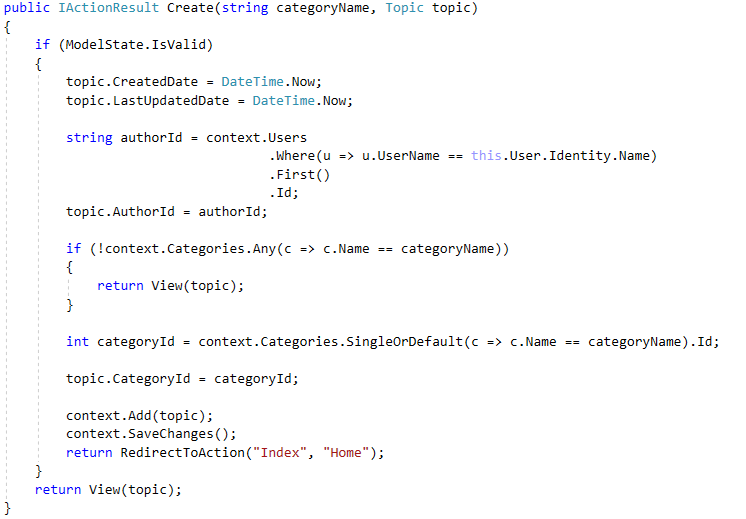
### Topic Create Get Action

In this method we have to get all the categories names and pass them to the view.



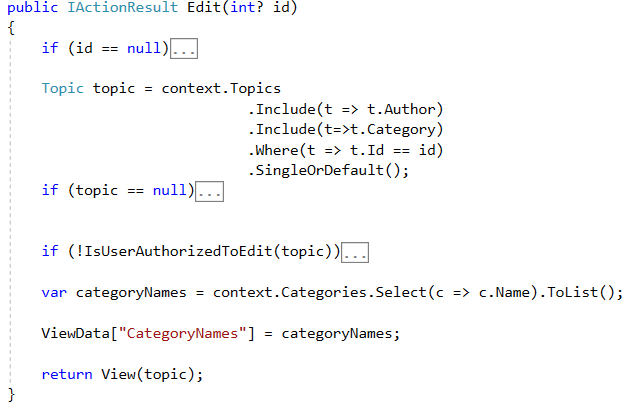
### Topic Create Post Action

In this method the changes are much more, first we are receiving a string parameter which is called **categoryName**.  
Then we have to find the given category by name and extract it from the db. Assign the category's Id to the topic and add the new Topic to the database.



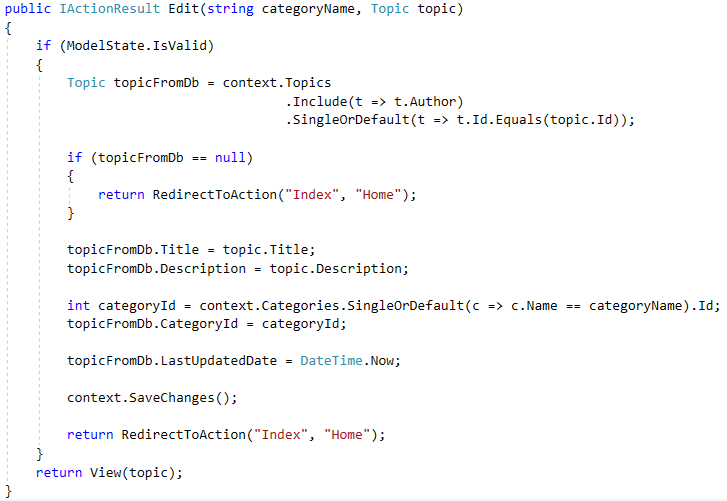
### Topic Edit Get Action

The changes to this action are the same as the changes in Create Get Action. Get all the names of the categories and pass them as collection to the View using ViewData. Check if your LINQ query needs to be edited too.



### Topic Edit Post Action

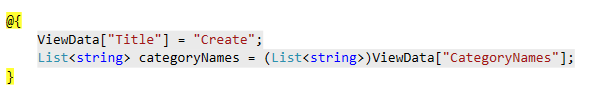
Similar to the changes in Topic Create Post Action we need to add string **categoryName** parameter.  
Assign the **categoryId** to the **topic** and save the changes.



## Update Topic Views

### Topic Create View Update

First change the top of your view to look like this.



Now we need a dropdown menu where we can chose which category our topic will be related to.  
So go to the "Views/Topic/Create.cshtml" and paste the following code after the description **div**.

|  |
| --- |
| <div class="form-group">  <**label** **asp-for**="Category" class="control-label"></**label**>  <select class="form-control" name="categoryName" required>  @foreach (var categoryName in categoryNames)  {  <**option** **value**="@categoryName">@categoryName</**option**>  }  </select>  </div> |

### Topic Details View Update

All we need to add here is element which will display the topic's category. Add the element right after the **last redaction**.

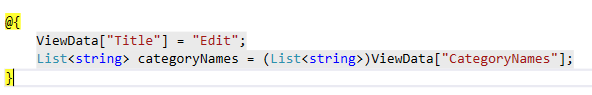
Views/Topic/Details.cshtml

|  |
| --- |
| <span class="item-wrap category-wrap">  <i class="fa fa-folder-open">Category: </i>  <span class="cell" title="Category">  @Model.Category.Name  </span>  </span> |

### Topic Edit View Update

The changes here are just like in the Topic Create View. Get the category names from the ViewData.

Views/Topic/Edit.cshtml

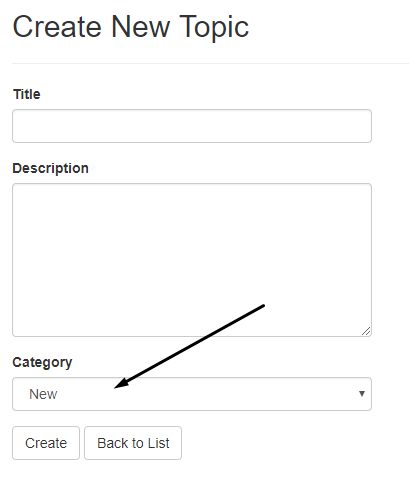
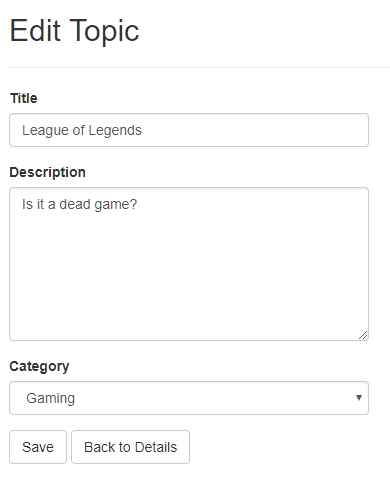


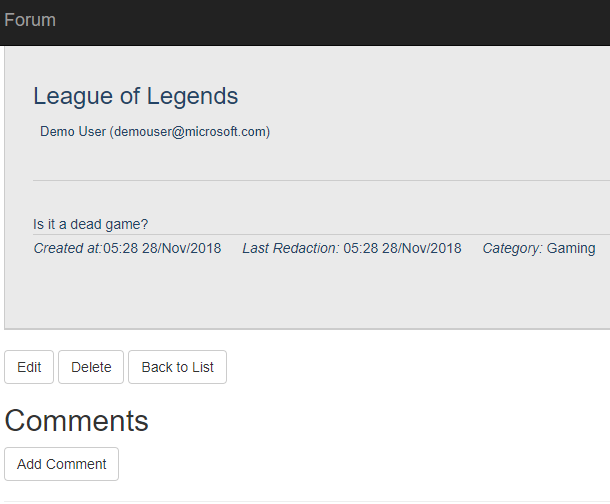
And render them at the bottom of the form, right below the Description.

|  |
| --- |
| <div class="form-group">  <**label** **asp-for**="Category" class="control-label"></**label**>  <select class="form-control" name="categoryName" required>  @foreach (var categoryName in categoryNames)  {  if (categoryName == Model.Category.Name) {  <**option** **value**="@categoryName" selected>@categoryName</**option**>  }  <**option** **value**="@categoryName">@categoryName</**option**>  }  </select>  </div> |

Now we should have categories implemented in all of the topic views.







# Creating Categories

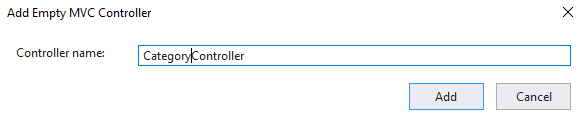
Let's make UI to create categories. The only problem is that we don't have a place in the forum where we see the categories. For this we will need a **Category Controller**.

## Creating the Category Controller

Head to the "**Controllers**" folder and create a new **Controller**:



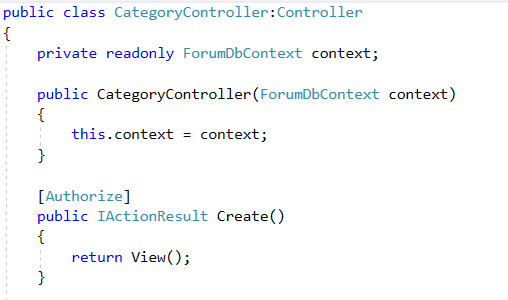




## Creating the Get Method

Just like creating a topic and comment, in order to create category, we need an instance of the class **ForumDbContext**, two actions (**get** and **post**) and a view.

In **CategoryController.cs**, create context property and the first action called **Create**:



## Creating the View

Now lets create the view -> "Views/Category/Create.cshtml":

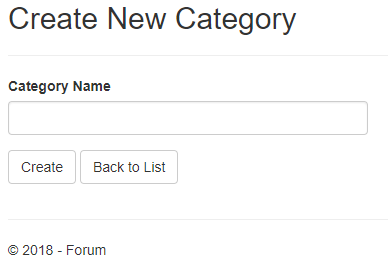


Paste the following code in the view:

|  |
| --- |
| @model Category  @{  ViewData["Title"] = "Create";  }  <h2>Create New Category</h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-action**="Create">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <div class="form-group">  <**label** **asp-for**="Name" class="control-label"></**label**>  <**input** **asp-for**="Name" class="form-control" />  <**span** **asp-validation-for**="Name" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Create" class="btn btn-default" />  <**a** **asp-controller**="Home" **asp-action**="Index" class="btn btn-default">Back to List</**a**>  </div>  </**form**>  </div>  </div>  @section Scripts {  @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}  } |

We should already be familiar with this. It's a **view** with a single form in which there are **text area** for category name and **two buttons**. One of the buttons **submits the form** and the other redirects to the details of the topic.

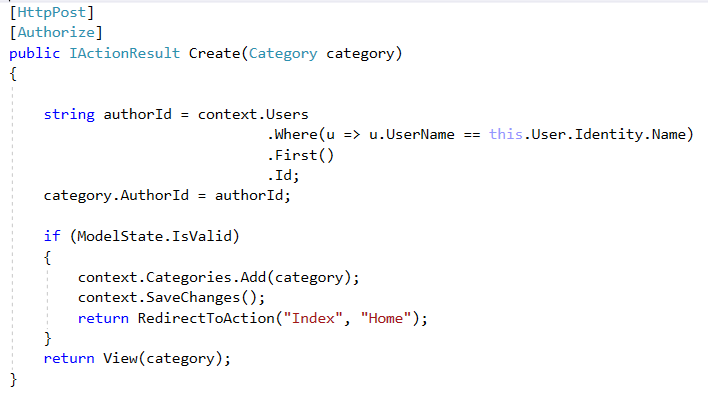
The final **view** should look something like this:



## Creating the Post Method

After the create form is submitted. ASP.Net core will map the data to a Category object so half of the work is already done. We should have some kind of validation of course using **ModelState.IsValid**. Our job here is to find the author using **User.Identity.Id** and to set the given **Category's** **AuthorId**. After all just save the category to the database.

Method should look like **this**:



# Category Details

After editing the **Index View** you've already noticed that every single one of our categories are links referring to a **details** view.

Let's create **action** which will **handle** this **request** and **view** which will **render** the **information**.   
Once we press on one of the **categories** the view must provide us with all **Topics** related to the category and buttons for managing the category aka **Edit** and **Delete**.

## Category Details Actions

We need a **GET Action** which will receive **int Id** as parameter. We must find the given **category**, to include all the data from the **Database** related to the category and pass it to the view. Also the list of categories must be passed to the view to render the side bar.



## Create Details View

Just create new file in **Views/Category** folder named **Details.cshtml** and paste there the following code.

|  |
| --- |
| @model Category  @{  List<Category> categories = (List<Category>)ViewData["Categories"];  }  <div class="container-fluid">  <div class="row forum-container">  <aside class="col-md-3 col-sm-12 forum-sidebar">  <div class="category-wrapper">  <p>  <**a** **asp-controller**="Category" **asp-action**="Create" class="btn btn-default">Create New Category</**a**>  </p>    <h3>Categories</h3>  <ul>  <li>  <**a** **asp-controller**="Home" **asp-action**="Index">All Topics</**a**>  </li>  @foreach (var category in categories)  {  <li>  <**a** **asp-controller**="Category" **asp-action**="Details" **asp-route-id**="@category.Id" class="">@category.Name</**a**>  <span>(@category.NumberTopics)</span>  </li>  }  </ul>  </div>  </aside>  <section class="col-md-9 col-sm-12 article-wrapper">  <p>  <**a** **asp-controller**="Topic" **asp-action**="Create" class="btn btn-default">Create New Topic</**a**>  <**a** **asp-controller**="Category" **asp-action**="Edit" **asp-route-id**="@Model.Id" class="btn btn-default">Edit</**a**>  <**a** **asp-controller**="Category" **asp-action**="Delete" **asp-route-id**="@Model.Id" class="btn btn-default">Delete</**a**>  </p>  @foreach (var topic in Model.Topics  .OrderByDescending(t => t.CreatedDate)  .ThenByDescending(t => t.LastUpdatedDate))  {  <article class="forum-post">  <div class="head-post">  <h2 class="post-title">  <**a** **asp-controller**="Topic" **asp-action**="Details" **asp-route-id**="@topic.Id">@topic.Title</**a**>  </h2>  <p>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Author: @topic.Author.UserName</span>  </span>  </span>  <span class="user-wrap">  <span class="cell">  <span class="user-name">Created Date: @topic.CreatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </span>  </span>  <span class="user-wrap">  <span class="cell">  Category: <**a** **asp-controller**="Category" **asp-action**="Details" **asp-route-id**="@topic.CategoryId">@topic.Category.Name</**a**>  </span>  </span>  </p>  </div>  <div class="meta-info">  <div class="col-md-6 last-replay">  <div class="table-holder">  <div class="last-replay-inner cell">  <span class="cell">Last activity: </span>  <span class="cell replay-date">@topic.LastUpdatedDate.ToString("HH:mm dd/MMM/yyyy")</span>  </div>  </div>  </div>  <div class="col-md-6 stats-holder">  <span class="item-wrap">  <span class="cell">Comments: </span>  <span class="cell">@topic.NumberComments</span>  </span>  </div>  </div>  </article>  }  </section>  </div>  </div> |

# Editing Categories

After all of the implementations you're probably tired of someone telling you what to do step by step and showing off all of the logic so now it's your time to shine. ☺ We'll give you some kind of description what you need to do and you must try to implement the logic by yourself.

## Create Post and Get Edit Action

Well we are doing this through the whole process so implementing those two methods must be easy task for you.

### Create GET Edit Action

First our action will receive **int Id** as parameter. Validate if the id is null and extract the Category from the Database using **LINQ query**. Check if the given category **exists**. And pass the category to the View.

### Create POST Edit Action

Post Edit Action is a bit more complicated. The action receives Category as a parameter. You must validate the model. Get the Category from the Database with the same Id. Validate if such a category exists and assign the new changes, it's actually only the name. Save the changes and redirect to the Home.

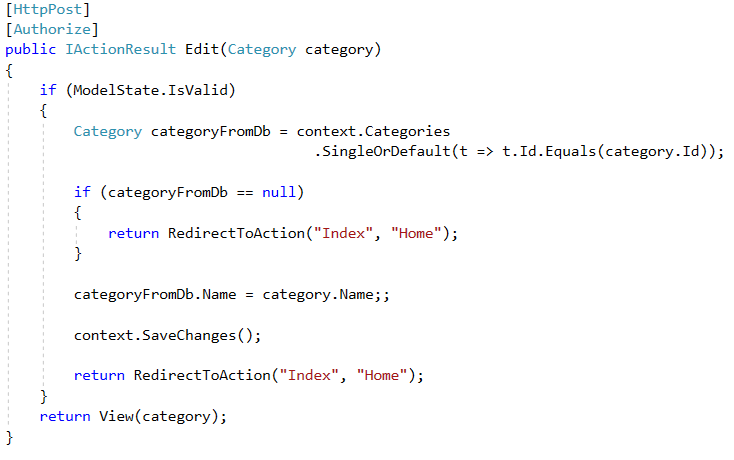
## Create the View

The view will be provided, but if you feel comfortable enough feel free to implement it by yourself, the view is actually almost the same as the Create view.

|  |
| --- |
| @model Category  <h2>Edit the @Model.Name Category </h2>  <hr />  <div class="row">  <div class="col-md-4">  <**form** **asp-action**="Edit">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <div class="form-group">  <**label** **asp-for**="Name" class="control-label"></**label**>  <**input** **asp-for**="Name" class="form-control" />  </div>  <div class="form-group">  <input type="submit" value="Save" class="btn btn-default" />  <**a** **asp-controller**="Home" **asp-action**="Index" class="btn btn-default">Back to List</**a**>  </div>  </**form**>  </div>  </div>  @section Scripts {  @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}  } |

## Hints

Of course we won't let you without any hints, here you can see the both methods fully implement



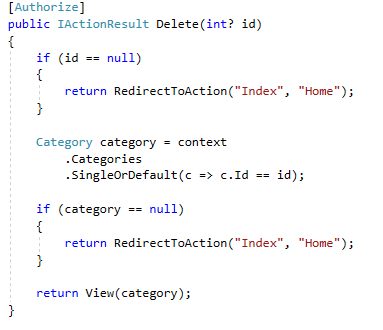
# Deleting Categories

## Create Post and Get Edit Action

### Create GET Delete Action

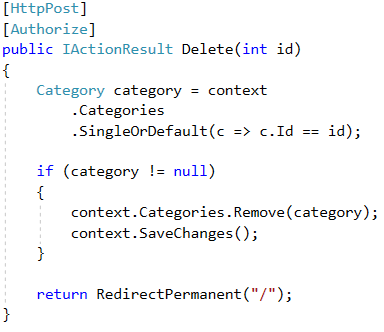
First we must create the GET action, which will be responsible for finding the category by Id and pass the data to the View. The view will just render category name and ask the client if he really wants to delete the category, absolutely nothing new.

Starting with the method. First we must validate the Id, get the category from the db as object, check if the category is null, and pass the object to the view.



### Create POST Delete Action

**Post Delete Action** is the same as all delete actions. We have the Id and all we have to do is validations and the remove operation.



## Create the Delete View

Add new **cshtml** file in the **Views/Category** folder named **Delete.cshtml** and paste the following code.

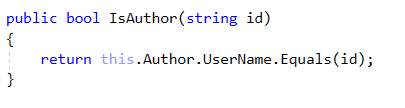
|  |
| --- |
| @model Category  @{  ViewData["Title"] = "Delete";  }  <h3>Are you sure you want to delete the category?</h3>  <div>  <article class="forum-post post-item highlighted-post">  <div class="head-post table-holder">  <div class="post-content">  <div>  <h1>@Model.Name</h1>  </div>  </div>  </div>  </article>  <div>  <**form** **asp-action**="Delete">  <**input** **type**="hidden" **asp-for**="@Model.Id" />  <input type="submit" value="Delete" class="btn btn-default" />  <**a** class= "btn btn-default" **asp-controller**="Home", **asp-action**="Index">Cancel</**a**>  </**form**>  </div>  </div> |

# Security

If our app have functionality to be used by many users probably we should implement functionality to prevent a user to edit and delete posts to other user.

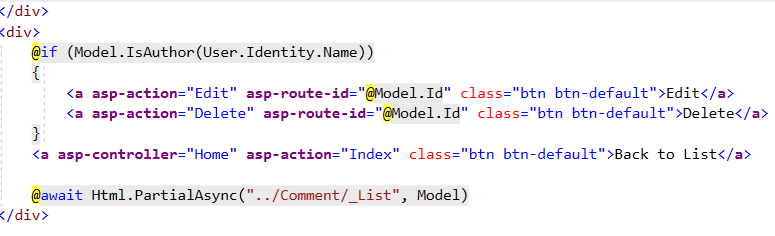
To implement some kind of restriction we will start from the models.

In the topic, comment and category models add the following method.

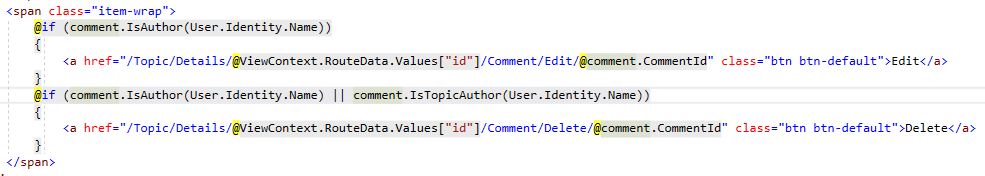
   
This method will help us to Identify if a user is author or not.

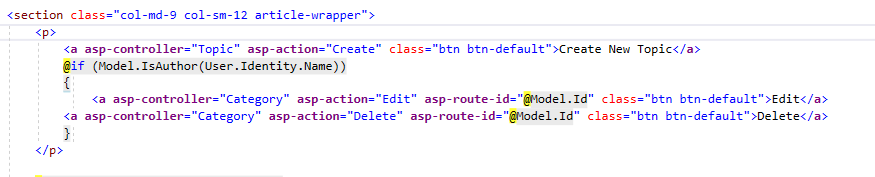
Now let's head to the views and restrict some of the functionality.

For example in **Views/Topic/Details.cshtml** we can use **if statement** to prevent **Edit** and **Delete** buttons of rendering.



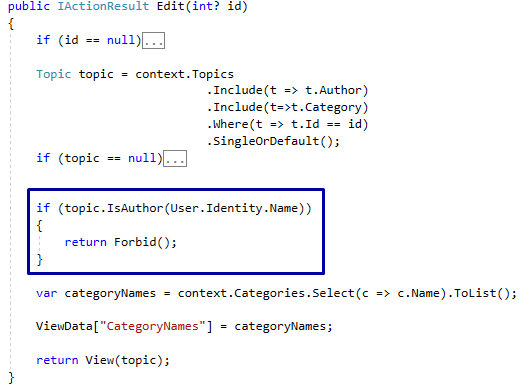
Editing the **Views/Comment/\_List.cshtml** will be exactly the same



Of course we won't forget about the categories. Open the **Views/Category/Details.cshtml** and edit the code.

Even though we restricted the UI still we can access **Delete** or **Edit** get actions using the **URL**.   
What we can do here is to use exactly the same method in the **controllers** to **forbid** this kind of actions.

For Example in **TopicController.cs** we can use the validation to prevent users accessing **Edit View**.  
In the **Edit Get Action** implement the validation just like this.



We will leave to you to implement the validations to other Controllers and Actions.