**Client and Server Security Assignment**

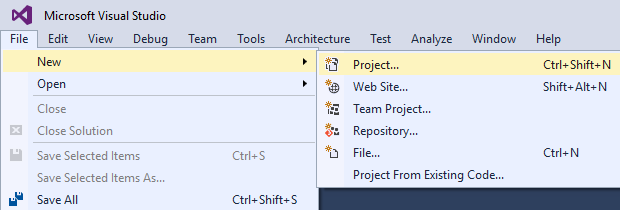
**4th year NPC**

**Lab Project**

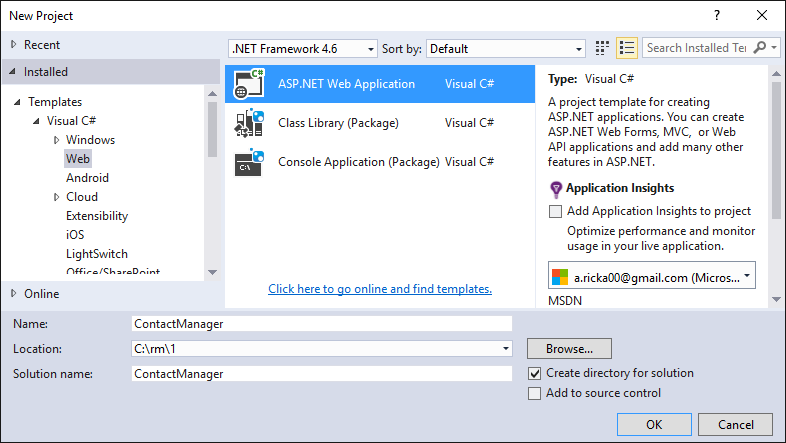
**Introduction to MVC (**[**www.tutorialspoint.com/csharp**](http://www.tutorialspoint.com/csharp) **)**

1. **Create an ASP.NET MVC 5 application**

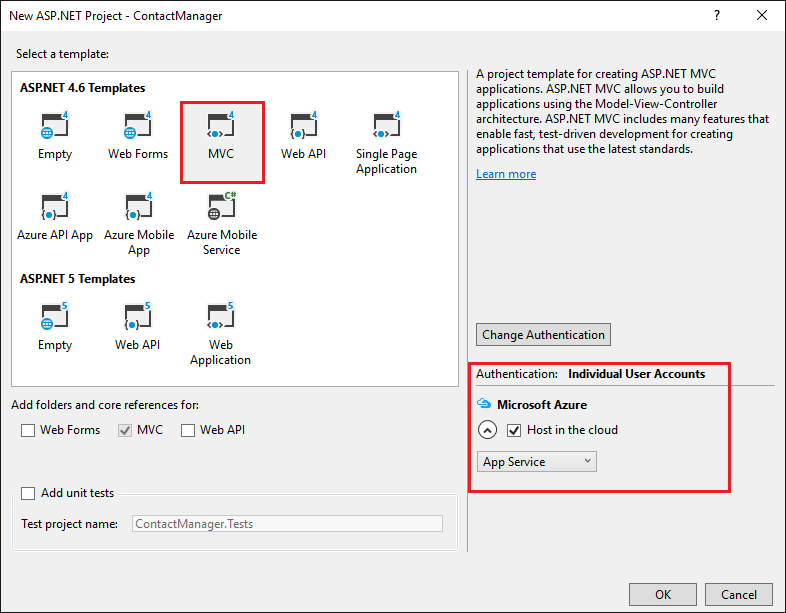
Install Visual Studio (latest version), create the project, From the File menu, click New Project.



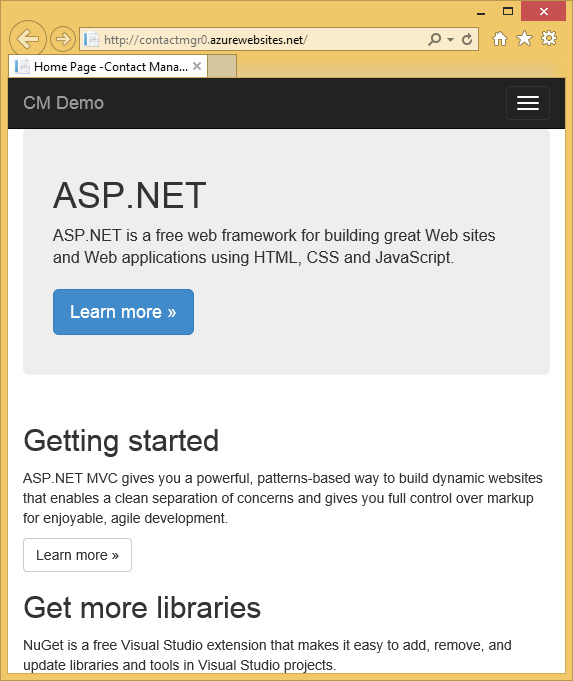
In the **New Project** dialog box, expand **C#** and select **Web** under **Installed Templates**, and then select **ASP.NET Web Application**. Name the application **ContactManager**, and then click **OK**.



In the **New ASP.NET Project** dialog box, select the **MVC** template. Verify **Authentication** is set to **Individual User Accounts**, **Host in the cloud** is checked, and **App Service** is selected.

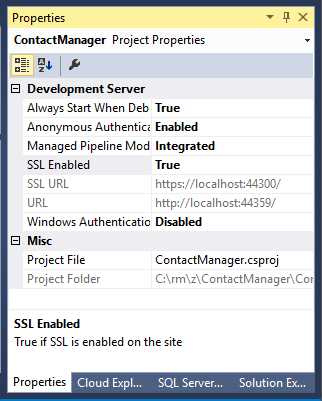


Click **OK**. Run the application locally Press CTRL+F5 to run the app. The application home page appears in the default browser.

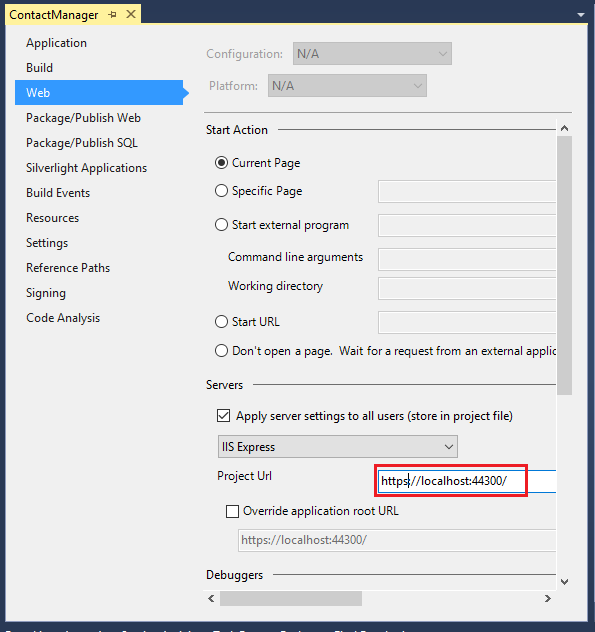


1. **Enable SSL for the Project**
2. In **Solution Explorer**, click the **ContactManager** project, then click F4 to open the **Properties** window.
3. Change **SSL Enabled** to **True**. Copy the **SSL URL**.

The SSL URL will be https://localhost:44300/ unless you've previously created SSL web apps.

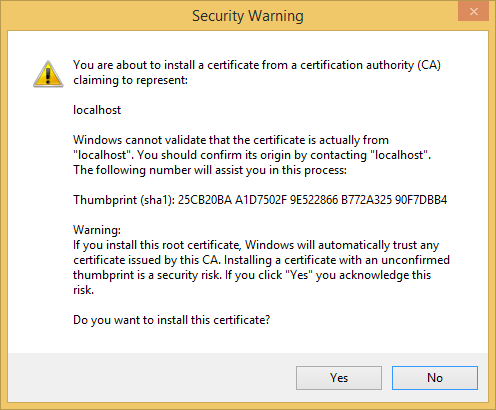


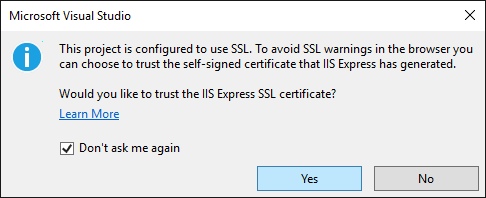
1. In **Solution Explorer**, right click the **Contact Manager** project and click **Properties**.
2. Click the **Web** tab. Change the **Project Url** to use the **SSL URL** and save the page (Control S).



Press CTRL+F5 to run the application. Click **Yes** to start the process of trusting the self-signed certificate that IIS Express has generated.

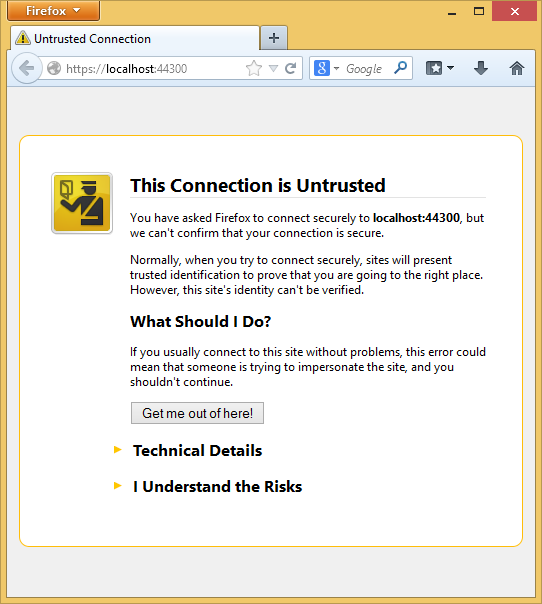
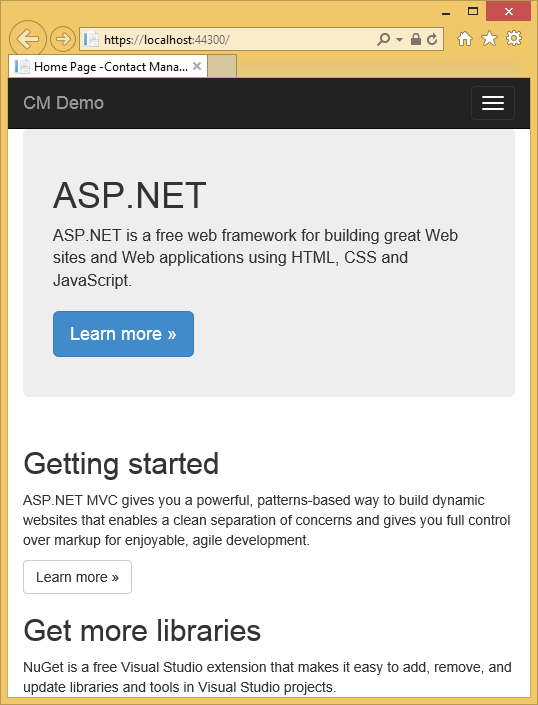
Read the **Security Warning** dialog and then click **Yes** if you want to install the certificate representing **localhost**.





IE shows the *Home* page and there are no SSL warnings.

Internet Explorer is a good choice when you're using SSL because it accepts the certificate and shows HTTPS content without a warning. Microsoft Edge and Google Chrome also accept the certificate. Firefox uses its own certificate store, so it displays a warning.



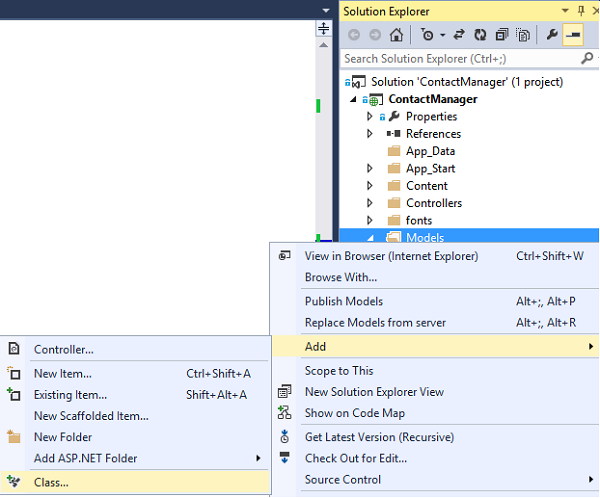
## Add a database to the application

Next, you'll update the app to add the ability to display and update contacts and store the data in a database. The app will use the Entity Framework (EF) to create the database and to read and update data.

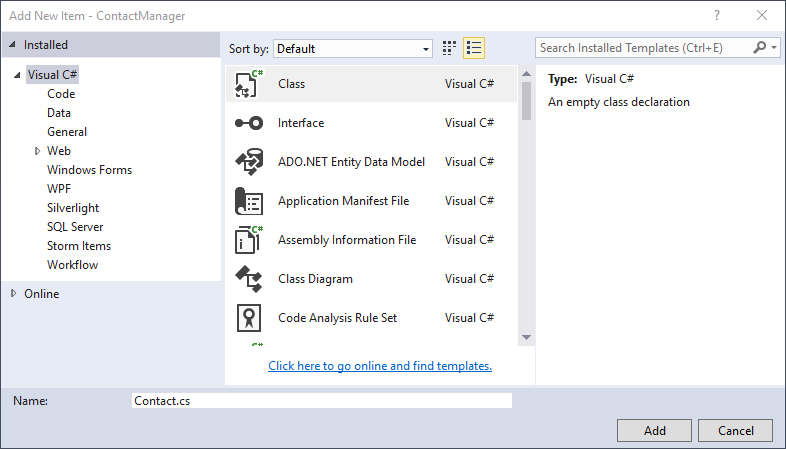
### Add data model classes for the student

You begin by creating a simple data model in code.

1. In **Solution Explorer**, right-click the Models folder, click **Add**, and then **Class**.



1. In the **Add New Item** dialog box, name the new class file *Student.cs*, and then click **Add**.



1. Replace the contents of the Contact.cs file with the following code. Copy

using System.ComponentModel.DataAnnotations;

using System.Globalization;

namespace StudentManager.Models

{

public class Student

{

public int StudentId { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public string City { get; set; }

public string Email { get; set; }

}

}

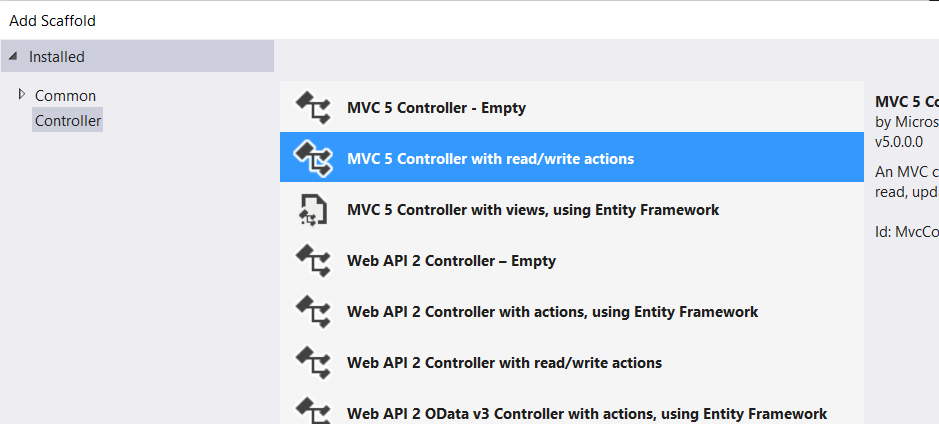
The **Contact** class defines the data that you will store for each contact, plus a primary key, StudentId, that is needed by the database.

### Create web pages that enable app users to work with the Student

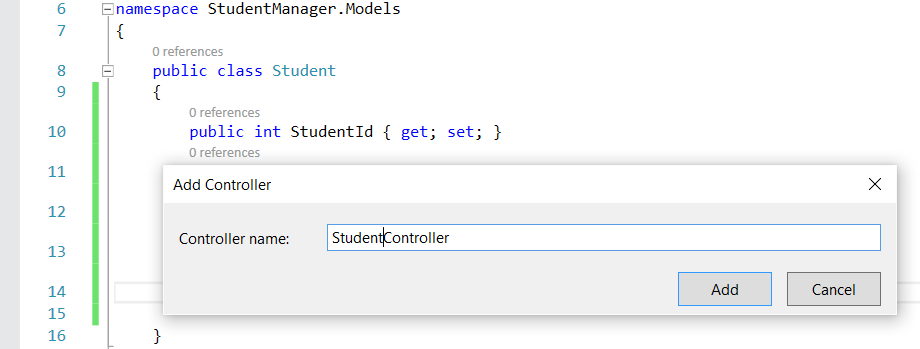
The ASP.NET MVC scaffolding feature can automatically generate code that performs create, read, update, and delete (CRUD) actions.

1. Build the project **(Ctrl+Shift+B)**. (You must build the project before using the scaffolding mechanism.)
2. In **Solution Explorer**, right-click the Controllers folder and click **Add**, and then click **Controller**.

In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using EF** and then click **Add**.



Add a controller name **StudentController**



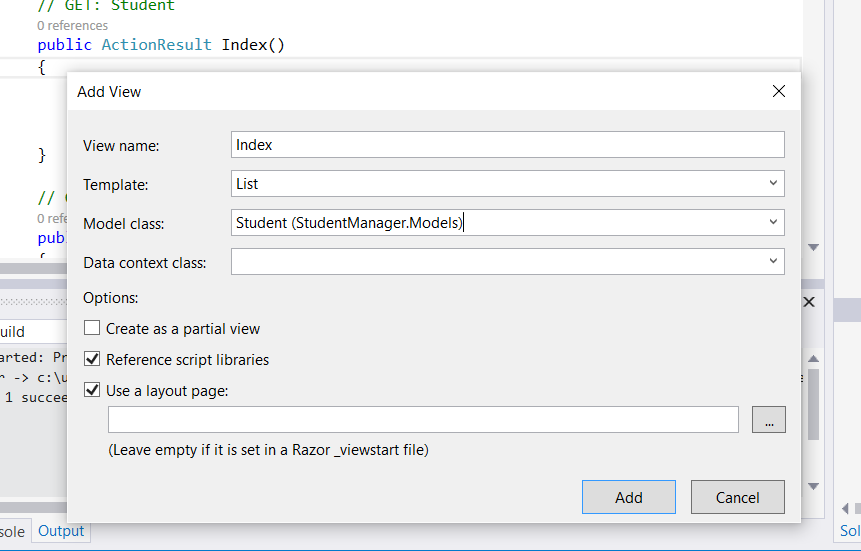
Click **Add**.

1. **Adding View to the Application**

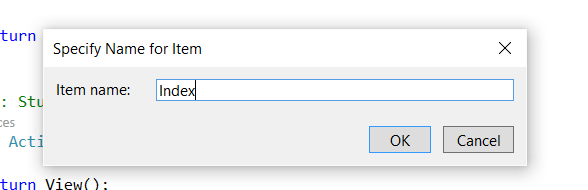
In the studentController class, right click on index ActionResult and select **Add View**. Name the view Index, Select List Template and choose **Student (StudentManager.Models)** from the list of Model Class. Leave any other field default

**And click add**

If you look in the folder Student a new file **Index.cshtml** was created wich is the View of our controller

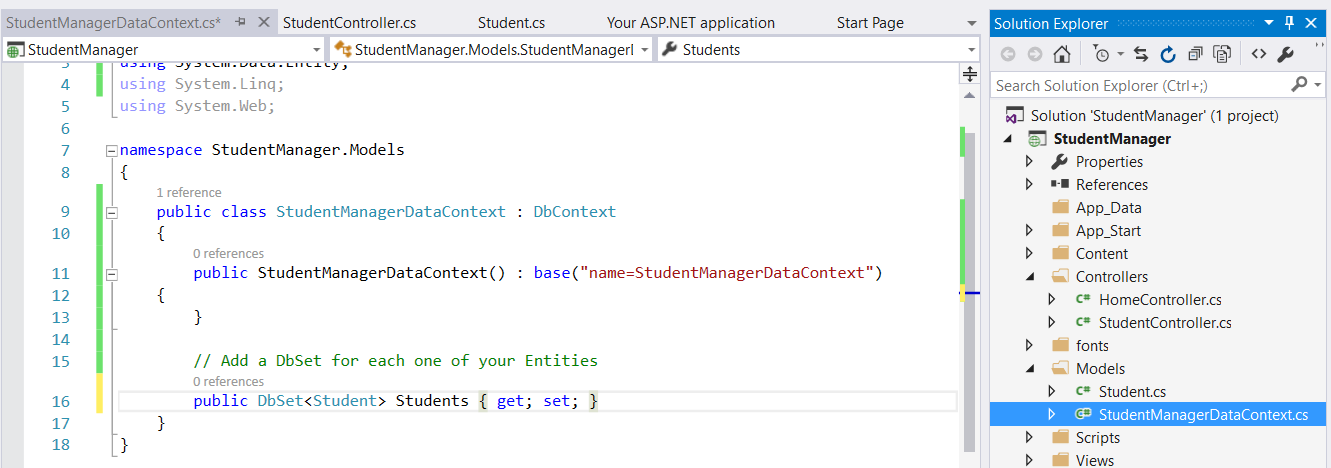


Call it index and click OK and then Ok again to accept all default



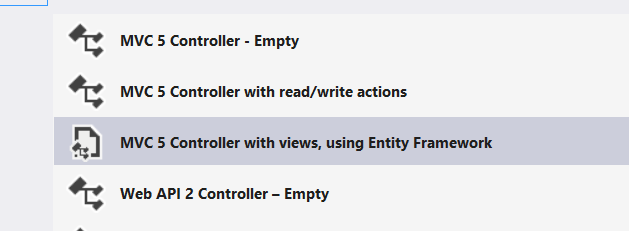
1. **Add DbContext to create a database**

To add another DbContext, just add a file C# class to models folder called **StudentManagerDataContext**.**cs**

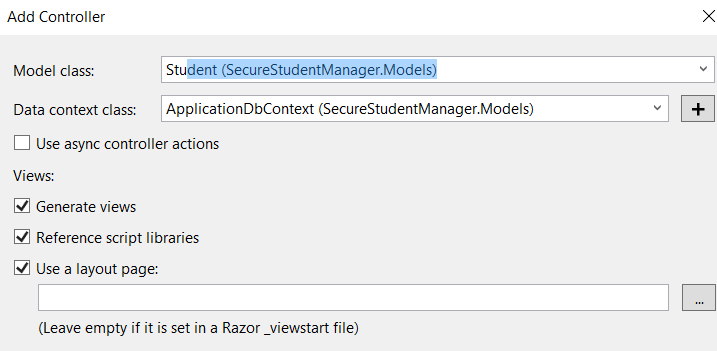


Using the **ApplicationDbContext** which provided for us by the entityframework

Create new controller by selecting MVC5 using entity framework (see next project)



Type the model Name and select **ApplicationDBContext** for data context class



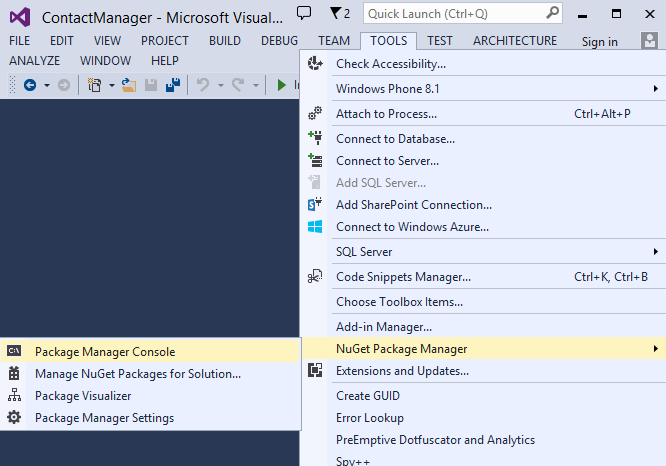
If you open IdentityModels.cs you will notice this line of code

public DbSet<Student> Students { get; set; }, which will be used to create a table called Students in the databases with columns corresponding to each property of this class like (StudentId, Name, Email, ..).

## Enable Migrations, create the database, add sample data and a data initializer

The next task is to enable the [**Code First Migrations**](http://msdn.microsoft.com/library/hh770484.aspx) feature in order to create database tables based on the data model that you created.

1. In the **Tools** menu, select **NuGet Package Manager** and then **Package Manager Console**.



* 1. **Adding References to the Application**
* In the **Package Manager Console** window, enter the following command:

Install-Package EntityFramework

Install-Package EntityFramework.Migrations -Pre

Install-Package EntityFramework.SqlServer -Pre

Then, type

Enable-Migrations

The **enable-migrations** command creates a *Migrations* folder, and it puts in that folder a *Configuration.cs* file that you can edit to seed the database and configure Migrations.

* In the **Package Manager Console** window, enter the following command:

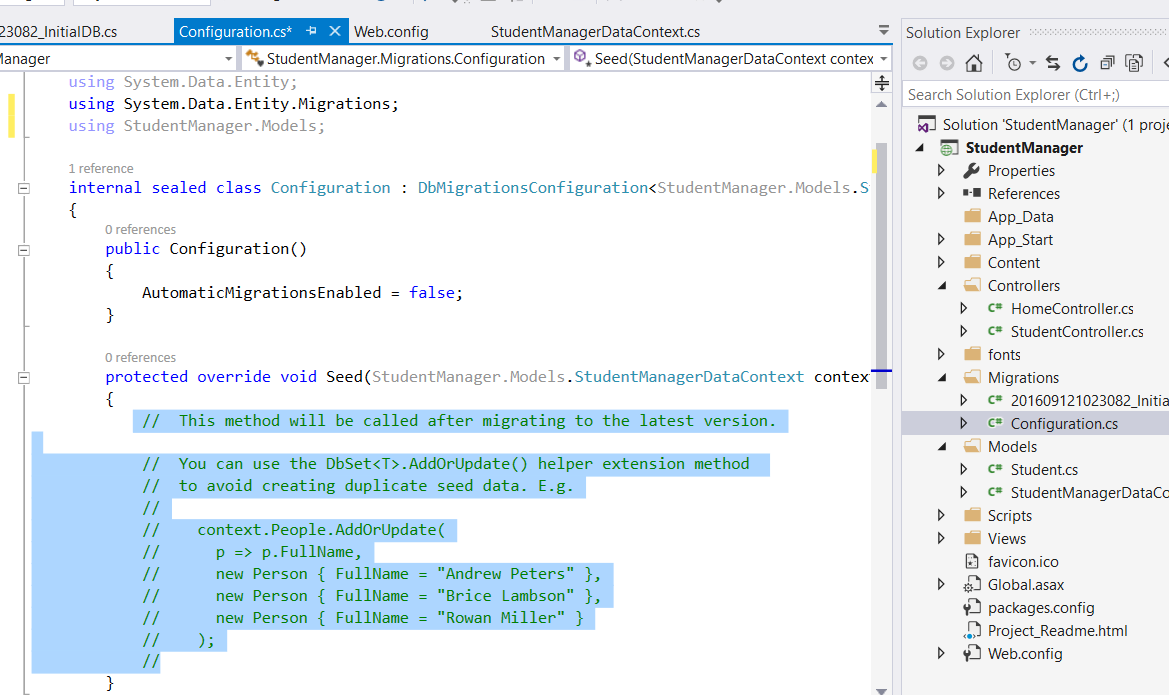
Add-Migration InitialDB

The **add-migration Initial** command generates a file named **<date\_stamp>Initial** in the*Migrations* folder. The code in this file creates the database tables. The first parameter (**Initial** ) is used to create the name of the file. You can see the new class files in **Solution Explorer**.

In the **Initial** class, the **Up** method creates the Student table, and the **Down** method (used when you want to return to the previous state) drops it.

* Open the *Migrations\Configuration.cs* file.
* Add the following using statement.

using StudentManager.Models;



* Replace the *Seed* method with the following code:
* Replace the content in the *Seed* method with the following code:

protected override void Seed(StudentManager.Models context)

{

context.Students.AddOrUpdate(p => p.Name,

new Student

{

Name = "Debra Garcia",

Address = "1234 Main St",

City = "Redmond",

Email = "debra@example.com",

},

new Student

{

Name = "Thorsten Weinrich",

Address = "5678 1st Ave W",

City = "Redmond",

Email = "thorsten@example.com",

},

new Student

{

Name = "Yuhong Li",

Address = "9012 State st",

City = "Redmond",

Email = "yuhong@example.com",

},

new Student

{

Name = "Jon Orton",

Address = "3456 Maple St",

City = "Redmond",

Email = "jon@example.com",

},

new Student

{

Name = "Diliana Alexieva-Bosseva",

Address = "7890 2nd Ave E",

City = "Redmond",

Email = "diliana@example.com",

}

);

}

This code initializes (seeds) the database with Studentinformation.

In the **Package Manager Console** enter the command:

Update-Database

The **update-database** runs the first migration which creates the database. By default, the database is created as a SQL Server Express LocalDB database.

* Press CTRL+F5 to run the application, and then click the **CM Demo** link; or navigate to **https://localhost:(port#)/Student/index.**
  1. **Show the data from the Database**

In the ActionResult called Index replace the content with the following code in **yellow**

using StudentManager.Models;

namespace StudentManager.Controllers

{

public class StudentController : Controller

{

// GET: Student

public ActionResult Index()

{

//create a reference (an instance) of our context class

StudentManagerDataContext ctx = new StudentManagerDataContext();

var studentList = ctx.Students;

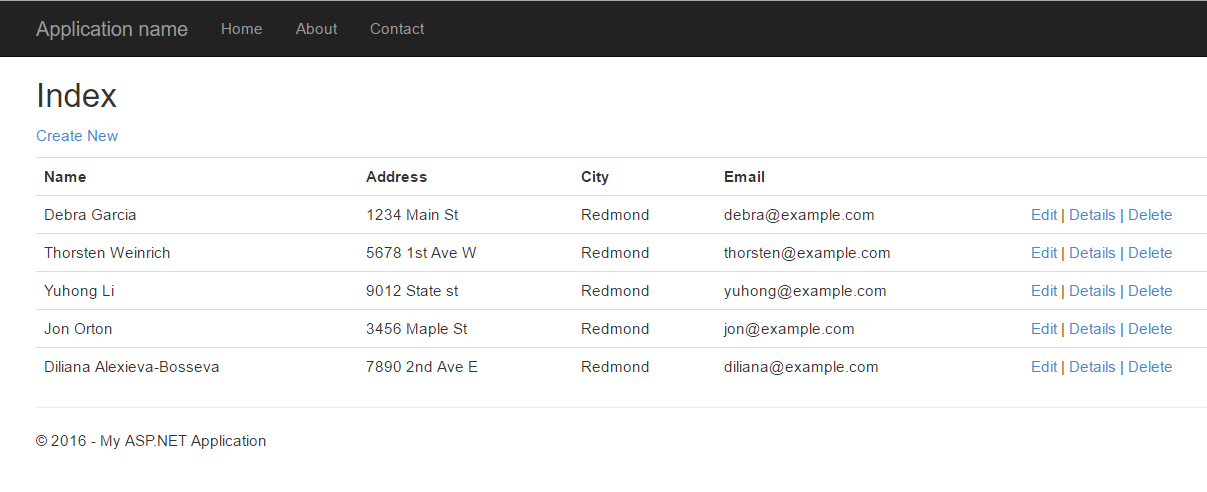
return View(studentList);

}

Don’t forget to add a reference like this using StudentManager.Models; on top of the class.

Run the application again or type in[**https://localhost:(port#)/Student/index**](about:blank)**.**

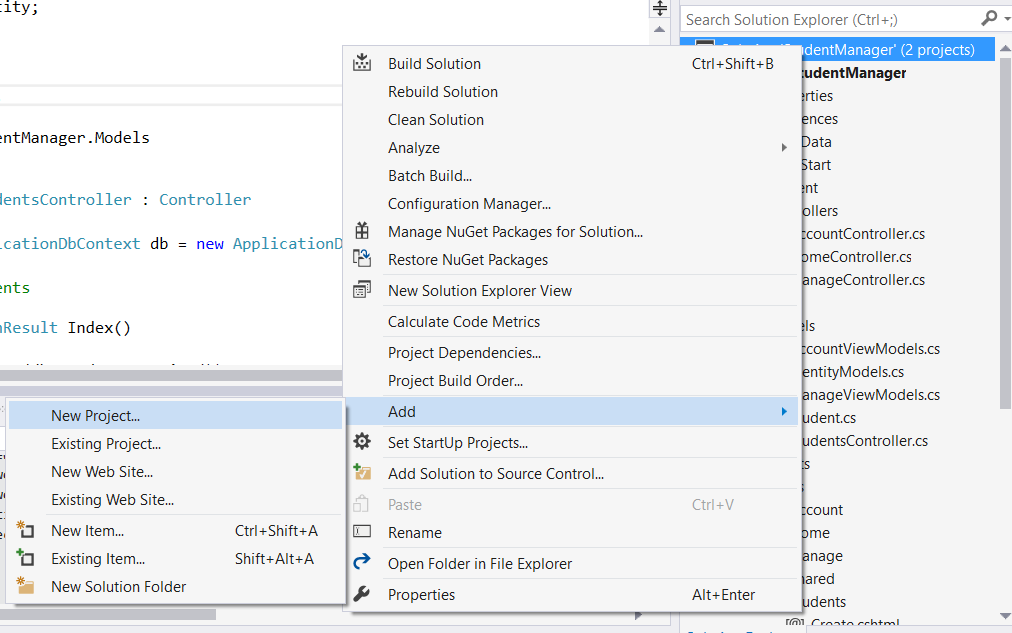
The application shows the seed data and provides edit, details and delete links. You can create, edit, delete and view data.



1. **Create a secure ASP.NET MVC web app with log in, email confirmation and password reset**

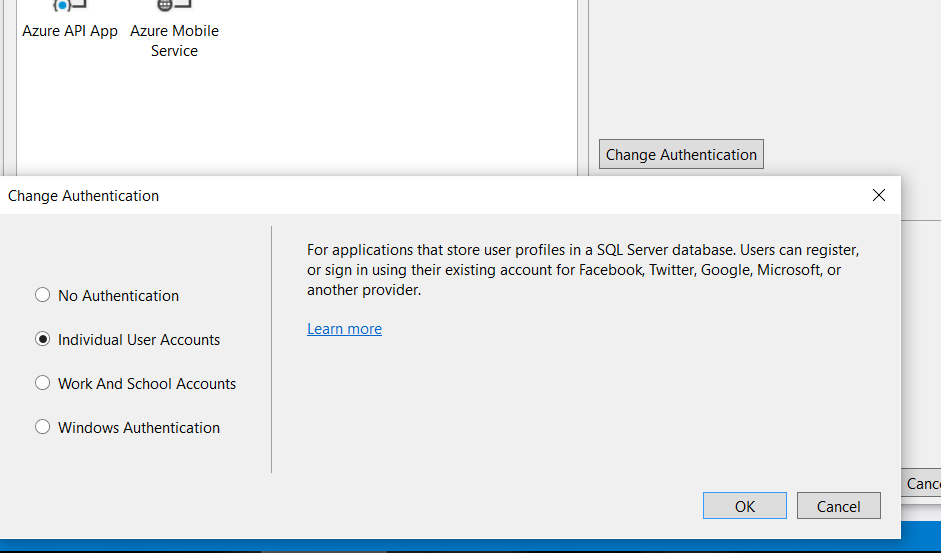
Add a new project select ASP.NET Web Application(.NET Framework) Give a name to the project;

Name it SecureStudentManager, by right clicking the solution and add a new project



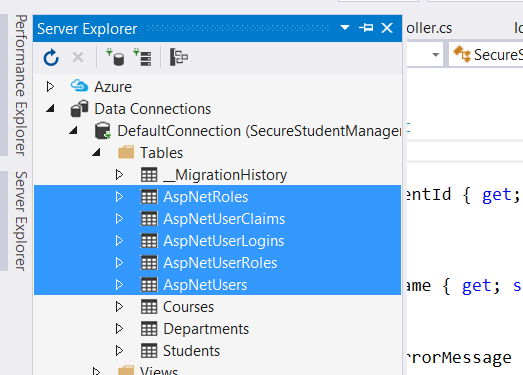
While creating the project select MVC and then Click on

**Change Authentication** and select **Individual User Account**



This creates an application for as with built in functionalities to log in and register new User.

Some tables which store users logins related data in the database were created for us.



**8.1. Models and Data Access**

Copy the models as in the previous projects and create database ApplicationDbContext which provided for us by the **entityframework**

So far, we’ve just been passing “dummy data” from our Controllers to our View templates. Now we’re ready to hook up a real database. which makes it really convenient for local development.

Database access with Entity Framework Code-First

We’ll use the **Entity Framework (EF)** to query and update the database. EF is a flexible object relational mapping (ORM) data API that enables developers to query and update data stored in a database in an object-oriented way.

Entity Framework supports a development paradigm called code-first. Code-first allows you to create model object by writing simple classes (also known as POCO from "plain-old" CLR objects), and can even create the database on the fly from your classes.

**8.2. Changes to our Model Classes**

We will be leveraging the database creation feature in Entity Framework in this tutorial. Before we do that, though, let’s make a few minor changes to our model classes to add in some things we’ll be using later on.

**8.3. Using Data Annotations for Model Validation**

We have a major issue with our Create and Edit forms: they’re not doing any validation. We can do things like leave required fields blank or type letters in the Price field, and the first error we’ll see is from the database.

We can easily add validation to our application by adding Data Annotations to our model classes. Data Annotations allow us to describe the rules we want applied to our model properties, and ASP.NET MVC will take care of enforcing them and displaying appropriate messages to our users.

**Adding Validation** to our Student form (create View). We’ll use the following Data Annotation attributes:

**Required** – Indicates that the property is a required field

**DisplayName** – Defines the text we want used on form fields and validation messages

**StringLength** – Defines a maximum length for a string field

**EmailAddress** -valididate Email Address

**Phone -** valididate Phone Number

**ErrorMessage** – to display error messages

**Range** – Gives a maximum and minimum value for a numeric field

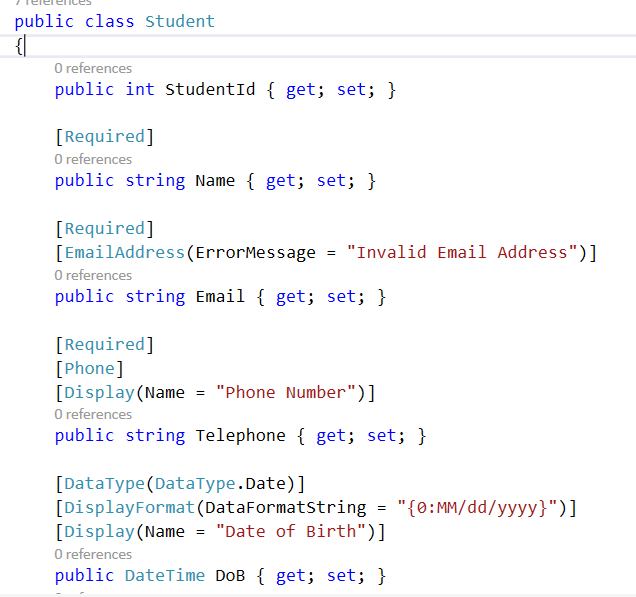
Note: For more information on Model Validation using Data Annotation attributes, see the MSDN documentation at <http://go.microsoft.com/fwlink/?LinkId=159063>

**8.5. Model Validation and Client-side validation**

The following illustration shows error messages that are automatically displayed in a browser when client-side validation fails. Messages displayed in the browser in response to validation errors

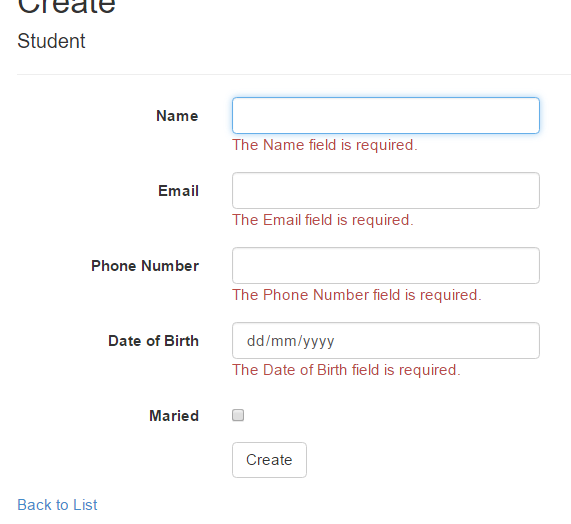
### To add validation using DataAnnotations attributes

1. Add a class to your project to contain the partial class definitions.
2. Add a namespace declaration to the partial class that matches the namespace of the data model that you are using.
3. Add attribute to each property in the class student as shown



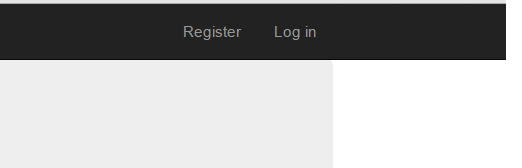
What this means is that we will always run all validators on an object, if that object had at least one value bound into it during model binding. We run the property-level validators first, and if all of those succeed, we'll run the model-level validators. In the case of our Student object with [**Required**] on the editable properties such as Name, if you accidentally left off the Name property (or a bad guy "under-posted" the form without it), the validation system would see the null Name value and trigger the error message.

Or instead of putting phone number you just typed strings or a wrong form of email, an error will be displayed

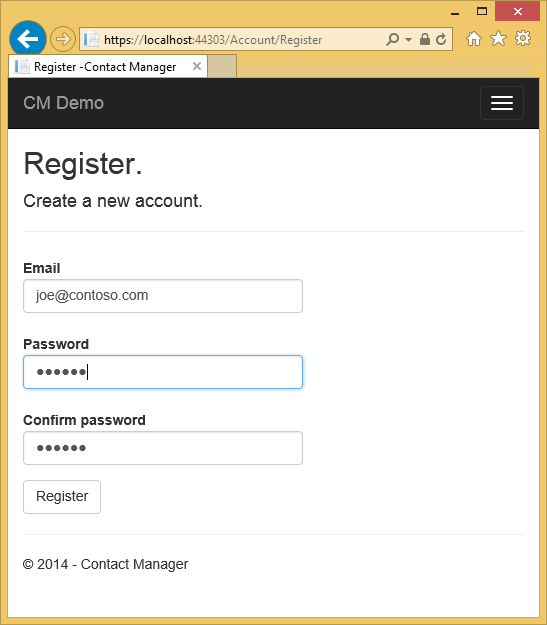


**Register New User**

Click on register and add a new name and password(number, letters and characters)

****

* Press CTRL+F5 to run the application.
* If you are still logged in from a previous session, hit the **Log out** link.
* Click on the **About** or **Contact** links. You are redirected to the login page because anonymous users cannot view those pages.
* Click the **Register as a new user** link and add a local user with email *joe@contoso.com*. Verify *Joe* can view the Home, About and Contact pages.



* Click the *CM Demo* link and verify that you see the data.
* Click an edit link on the page, you will be redirected to the login page (because a new local user is not added to the *canEdit* role).
* Log in as *user1@musanze.com* with password of "Password1" (the "0" in "word" is a zero). You are redirected to the edit page you previously selected.
* If you can't log in with that account and password, try copying the password from the source code and pasting it. If you still can't log in, check the **UserName** column of the **AspNetUsers** table to verify *user1@musanze.com* was added.
* Verify you can make data changes.

1. **Add a local user and the *canEdit* role to the database.**

Only those users in the *canEdit* role will be able to edit data. A best practice is to name roles by the actions they can perform, so *canEdit* is preferred over a role called *admin*. When your application evolves, you can add new roles as you wish such as *canDeleteMembers* rather than the less descriptive *superAdmin*.

1. Open the *migrations\configuration.cs* file and add the following using statements:

using Microsoft.AspNet.Identity;

using Microsoft.AspNet.Identity.EntityFramework;

1. Add **RolesController** (see code handout) with AdduserToRoles actions and add Role and User modes as well as corresponding views

[HttpPost]

public ActionResult AddUserToRoles(string UserId, string RoleId)

{

if(UserId != null && RoleId != null)

{

var um = new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(context));

var rm = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));

var correspondingUserId = um.Users.Where(x => x.Id == UserId).Select(x => x.Id).SingleOrDefault();

var correspondingRole = rm.Roles.Where(x => x.Id == RoleId).Select(x => x.Name).SingleOrDefault();

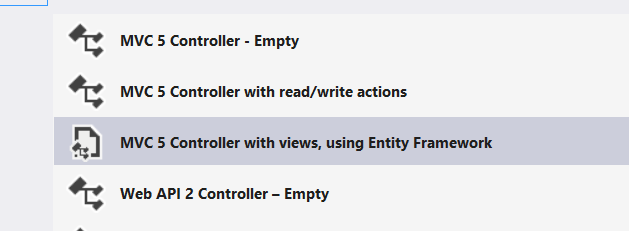
um.AddToRole(correspondingUserId, correspondingRole);

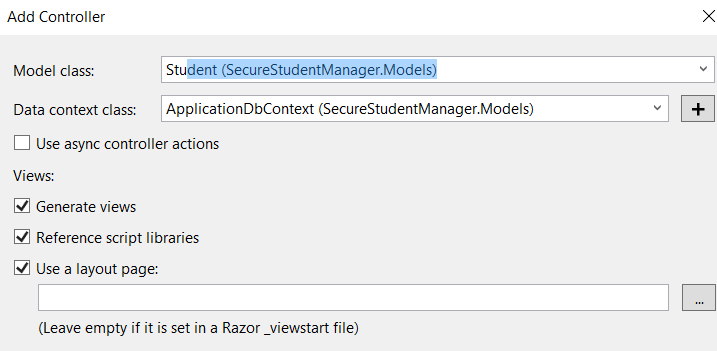
}

return RedirectToAction("Index");

}

Create new controller by selecting MVC5 using entity framework and selecting ApplicationDBContext as it is the context that create the relent user, roles database tables.





Add as many models you want for your application.

Run the enable migration, add migration name and update database to create the data base.

## Securing Application with Authorize Attribute

In this section you apply the [Authorize](http://msdn.microsoft.com/library/system.web.mvc.authorizeattribute.aspx) attribute to restrict access to the action methods. Anonymous users will be able to view only the **Index** action method of the home controller. Registered users will be able to create data (The **Index** and **Details** pages of the Cm controller). Only users in the *canEdit* role will be able to access action methods that change data. **There 2 approaches (secure the whole application or secure individual Controller)**

1. Open the *App\_Start\FilterConfig.cs* file and replace the *RegisterGlobalFilters* method with the following (which adds the two filters):

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new HandleErrorAttribute());

filters.Add(new System.Web.Mvc.AuthorizeAttribute());

filters.Add(new RequireHttpsAttribute());

}

This code adds the [Authorize](http://msdn.microsoft.com/library/system.web.mvc.authorizeattribute.aspx) filter and the [RequireHttps](http://msdn.microsoft.com/library/system.web.mvc.requirehttpsattribute.aspx) filter to the application. The[Authorize](http://msdn.microsoft.com/library/system.web.mvc.authorizeattribute.aspx) filter prevents anonymous users from accessing any methods in the application. You will use the [AllowAnonymous](http://blogs.msdn.com/b/rickandy/archive/2012/03/23/securing-your-asp-net-mvc-4-app-and-the-new-allowanonymous-attribute.aspx) attribute to opt out of the authorization requirement in a couple methods, so anonymous users can log in and can view the home page. The[RequireHttps](http://msdn.microsoft.com/library/system.web.mvc.requirehttpsattribute.aspx) requires that all access to the web app be through HTTPS.

1. An alternative approach is to add the [Authorize](http://msdn.microsoft.com/library/system.web.mvc.authorizeattribute.aspx) attribute and the [RequireHttps](http://msdn.microsoft.com/library/system.web.mvc.requirehttpsattribute.aspx) attribute to each controller, but it's considered a security best practice to apply them to the entire application. By adding them globally, every new controller and action method you add is automatically protected -- you don't need to remember to apply them.

Add the [AllowAnonymous](http://blogs.msdn.com/b/rickandy/archive/2012/03/23/securing-your-asp-net-mvc-4-app-and-the-new-allowanonymous-attribute.aspx) attribute to the **Index** method of the Home controller. The[AllowAnonymous](http://blogs.msdn.com/b/rickandy/archive/2012/03/23/securing-your-asp-net-mvc-4-app-and-the-new-allowanonymous-attribute.aspx) attribute enables you to white-list the methods you want to opt out of authorization.

// GET: Students

[AllowAnonymous]

public ActionResult Index()

{

return View(db.Students.ToList());

}*AllowAnonymous*, is also used in the login and registration methods of the Account controller.

In *CmController.cs*, add [Authorize(Roles = "canEdit")] to the HttpGet and HttpPost methods that change data (Create, Edit, Delete, every action method except Index and Details) in the *Cm* controller. A portion of the completed code is shown below:

[Authorize(Roles = "canEdit")]

public ActionResult Edit(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Student student = db.Students.Find(id);

if (student == null)

{

return HttpNotFound();

}

return View(student);

}

[HttpPost]

[ValidateAntiForgeryToken]

[Authorize(Roles = "canEdit")]

public ActionResult Edit([Bind(Include = "StudentId,Name,Email,Telephone,DoB,Maried")] Student student)

{

if (ModelState.IsValid)

{

db.Entry(student).State = EntityState.Modified;

db.SaveChanges();

return RedirectToAction("Index");

}

return View(student);

}}

return View(student);

}}

**10.1. ModelState (if (ModelState.IsValid))**

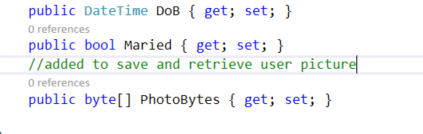
ModelState is a property of a Controller, and can be accessed from those classes that inherit from System.Web.Mvc.Controller. The ModelState represents a collection of name and value pairs that were submitted to the server during a POST.

It also contains a collection of error messages for each value submitted to the server, and to store the validation errors associated with those values.

With the above changes in place, modify the Add view to display the error messages if they occur: by adding @Html.ValidationMessageFor(x => x.Name)

1. **To add a picture in the project**

Add a new property in you student model class that you named **Student.cs** in the model folder bellow maried



**Remember to run**

* 1. **add-Migration second**
  2. **update-database**

In the view folder locate Create in the student folder and copy past this before the submit button add following code

<div class="fallback">

@Html.LabelFor(model => model.PhotoBytes, htmlAttributes: new {@class = "control-label col- md-2" })

<div class="col-md-10">

<input type="file" name="fileUpload" />

<br />

@Html.ValidationMessageFor(model => model.EnrollmentDate, "", new { @class = "text-danger" })

</div>

</div>

This is to help display the upload file button, so that you can upload a picture for each student.

In **StudentController add following:**

Copy past the content of the Create (in HttpPost mode)

public ActionResult Student(Student student)

{

//create a HttpPostedFileBase variable to hold uploaded files

HttpPostedFileBase file;

//check if the submitted data from the browser are valid

if (ModelState.IsValid)

{

//check if there is at least one submitted file and if none then give an error

if (Request.Files.Count < 0)

{

ModelState.AddModelError("File", "Please Upload Your file");

}

//if there is file uploaded the check fo rhte size and the format(extention)

else if (Request.Files[0].ContentLength > 0)

{

file = Request.Files[0];

int MaxContentLength = 1024 \* 1024 \* 5; //Size = 5 MB

//check the format to see if the file is an allowed picture format

string[] AllowedFileExtensions = new string[] { ".jpg", ".gif", ".png", ".pdf" };

if (!AllowedFileExtensions.Contains(file.FileName.Substring(file.FileName.LastIndexOf('.'))))

{

ModelState.AddModelError("File", "Please file of type: " + string.Join(", ", AllowedFileExtensions));

}

// if the size is bigger than 5Mdescribed above the give an error

else if (file.ContentLength > MaxContentLength)

{

ModelState.AddModelError("File", "Your file is too large, maximum allowed size is: " + MaxContentLength + " MB");

}

else

{

//save the picture to a folder named Upload. You have to create this folder in you project the same

//place the folder Model or controller or others are.. just right click the project name and add folder

var fileName = Path.GetFileName(file.FileName);

var path = Path.Combine(Server.MapPath("~/Upload"), fileName);

file.SaveAs(path);

ModelState.Clear();

ViewBag.Message = "File uploaded successfully. File path : ~/Upload/" + fileName;

// after keeping the picture on our webserve root folder now

//Read a file and add to the database.

byte[] imageData = null;

using (var binaryReader = new BinaryReader(file.InputStream))

{

imageData = binaryReader.ReadBytes(file.ContentLength);

}

student.PhotoBytes = imageData;

}

}

ctx.Students.Add(student);

ctx.SaveChanges();

return RedirectToAction("Index");

}

return View();

}

**In the Details actionresults**

public ActionResult Details(int id)

{

var student = ctx.Students.Find(id);

// Read a file from the database and display it and convert it.

//notice that we kept the result in the ViewBag for simplicity

ViewBag.ImageToShow = Convert.ToBase64String(student.PhotoBytes);

return View(student);

}

Then change the Detail view in the view folder like following by adding this

<dt>

@Html.DisplayNameFor(model => model.PhotoBytes)

dt>

dd>

@Html.Raw("<img src=\"data:image/jpeg;base64," + ViewBag.ImageToShow + "\" />");

</dd>

**ASP.NET Cookies Overview**

A cookie is a small bit of text that accompanies requests and pages as they go between the Web server and browser. The cookie contains information the Web application can read whenever the user visits the site.

A cookie is a small bit of text that accompanies requests and pages as they go between the Web server and browser. The cookie contains information the Web application can read whenever the user visits the site.For example, if a user requests a page from your site and your application sends not just a page, but also a cookie containing the date and time, when the user's browser gets the page, the browser also gets the cookie, which it stores in a folder on the user's hard disk.

Later, if user requests a page from your site again, when the user enters the URL the browser looks on the local hard disk for a cookie associated with the URL. If the cookie exists, the browser sends the cookie to your site along with the page request. Your application can then determine the date and time that the user last visited the site. You might use the information to display a message to the user or check an expiration date.

Cookies are associated with a Web site, not with a specific page, so the browser and server will exchange cookie information no matter what page the user requests from your site. As the user visits different sites, each site might send a cookie to the user's browser as well; the browser stores all the cookies separately.

Cookies help Web sites store information about visitors. More generally, cookies are one way of maintaining continuity in a Web application—that is, of performing state management. Except for the brief time when they are actually exchanging information, the browser and Web server are disconnected. Each request a user makes to a Web server is treated independently of any other request. Many times, however, it's useful for the Web server to recognize users when they request a page. For example, the Web server on a shopping site keeps track of individual shoppers so the site can manage shopping carts and other user-specific information. A cookie therefore acts as a kind of calling card, presenting pertinent identification that helps an application know how to proceed.

**Writing Cookies**

The browser is responsible for managing cookies on a user system. Cookies are sent to the browser via the HttpResponse object that exposes a collection called Cookies. You can access the HttpResponse object as the Response property of your Page class. Any cookies that you want to send to the browser must be added to this collection. When creating a cookie, you specify a Name and Value. Each cookie must have a unique name so that it can be identified later when reading it from the browser. Because cookies are stored by name, naming two cookies the same will cause one to be overwritten.

You can add cookies to the Cookies collection in a number of ways. The following example shows two methods to write cookies:

//// You can add cookies to the Cookies collection in a number of ways.

//Response.Cookies["userInfo"]["userName"] = "patrick";

//Response.Cookies["userInfo"]["lastVisit"] = DateTime.Now.ToString();

//Response.Cookies["userInfo"].Expires = DateTime.Now.AddDays(1);

HttpCookie aCookie = new HttpCookie("userInfo");

aCookie.Values["userName"] = "patrick";

aCookie.Values["mySession"] = Session["StudentDetails"].ToString();

aCookie.Values["lastVisit"] = DateTime.Now.ToString();

aCookie.Expires = DateTime.Now.AddDays(1);

Response.Cookies.Add(aCookie);

The example adds two cookies to the Cookies collection, one named userName and the other named lastVisit. For the first cookie, the values of the Cookies collection are set directly.

For the second cookie, the code creates an instance of an object of type HttpCookie, sets its properties, and then adds it to the Cookies collection via the Add method. When you instantiate an HttpCookie object, you must pass the cookie name as part of the constructor.

Both examples accomplish the same task, writing a cookie to the browser. In both methods, the expiration value must be of type DateTime. However, the lastVisited value is also a date-time value. Because all cookie values are stored as strings, the date-time value has to be converted to a String

**Reading Cookies**

When a browser makes a request to the server, it sends the cookies for that server along with the request. In your ASP.NET applications, you can read the cookies using the HttpRequest object, which is available as the Request property of your Page class. The structure of the HttpRequest object is essentially the same as that of the HttpResponse object, so you can read cookies out of the HttpRequest object much the same way you wrote cookies into the HttpResponse object. The following code example shows two ways to get the value of a cookie named username and display its value in a Label control:

if (Request.Cookies["userName"] != null)

Label1.Text = Server.HtmlEncode(Request.Cookies["userName"].Value);

if (Request.Cookies["userName"] != null)

{

HttpCookie aCookie = Request.Cookies["userName"];

Label1.Text = Server.HtmlEncode(aCookie.Value);

}

Before trying to get the value of a cookie, you should make sure that the cookie exists; if the cookie does not exist, you will get a NullReferenceException exception. Notice also that the HtmlEncode method was called to encode the contents of a cookie before displaying it in the page. This makes certain that a malicious user has not added executable script into the cookie. For more about cookie security, see the "Cookies and Security" section.

Because different browsers store cookies differently, different browsers on the same computer won't necessarily be able to read each other's cookies. For example, if you use Internet Explorer to test a page one time, but then later use a different browser to test again, the second browser won't find the cookies saved by Internet Explorer.

**Cookies and Security**

The security issues with cookies are similar to those of getting data from the client. In your application, cookies are another form of user input and are therefore subject to examining and spoofing. A user can as a minimum see the data that you store in a cookie, since the cookie is available on the user's own computer. The user can also change the cookie before the browser sends it to you.

You should never store sensitive data in a cookie, such as user names, passwords, credit card numbers, and so on. Do not put anything in a cookie that should not be in the hands of a user or of someone who might somehow steal the cookie.

Similarly, be suspicious of information you get out of a cookie. Do not assume that the data is the same as when you wrote it out; use the same safeguards in working with cookie values that you would with data that a user has typed into a Web page. The examples earlier in this topic showed HTML-encoding the contents of a cookie before displaying the value in a page, as you would before displaying any information you get from users.

Cookies are sent between browser and server as plain text, and anyone who can intercept your Web traffic can read the cookie. You can set a cookie property that causes the cookie to be transmitted only if the connection uses the Secure Sockets Layer (SSL). SSL does not protect the cookie from being read or manipulated while it is on the user's computer, but it does prevent the cookie from being read while in transit because the cookie is encrypted.

In the web.config file add following line

<httpCookies httpOnlyCookies="true" requireSSL="true" />

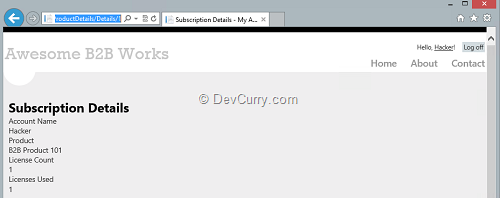
### What is Cross Site Request Forgery?

Cross Site Request forgery is a type of a hack where the hacker exploits the trust of a website on the user. In other words, the site trusts the user (because they have authenticated themselves) and accepts data that turns out to be malicious.

**How does it work?**

A CSRF attack depends on the fact that the site trusts the user’s input. From here on the hacker attempts to get authenticated users to click on links that submit data without the user actually realizing. For example, say you are logged on to your bank that has the ability to transfer money from one account to another. The hacker somehow reverse engineers this form and sets up a duplicate form that submits transfer requests to their own account.   
  
Today, I’ll demonstrate it in a simplified licensing scenario. Let’s suppose we have a cloud hosted B2B solution that allows your users to purchase a license count ‘n’ for  *10000$.*  Now obviously this is a costly license and say one of your customers is a bad guy. They initially bought 1 license but now their needs have increased and they need 5 more licenses. This is going to set them back by 5x 100000$ which is a lot of money in these hard economic times. Instead, they employ a BlackHat (hacker) for $2000, give them access to the current account, and ask them to devise a hack to do a surreptitious modification of license count. Let’s see how this could be possible.

### Hacking your Web App

Let’s say you have a standard ASP.NET MVC site that uses Forms Authentication. Your users have a user name/password they log in, do their stuff and log out. While logged in, they can see their subscription via an Account Details page that looks as follows  
  
  
  
**Guessing Game Begins**  
  
Hacking involves a lot of things amongst which guessing the functionality plays a big role. Now the first thing the hacker will note is that the ProductDetails/Details/1 in the URL. They will guess that it’s probably an ASP.NET MVC site that has a ProductDetails/Edit/1 and ProductDetails/Create pages for their corresponding actions in the ProductDetails controller.   
  
So the next thing they will try is alter the URL to go to the Edit Page. This will not work and they will get re-directed to the login page. Why? That’s because we were careful enough to Authorize only users in the **Admins** group access to the Edit action method. We did this using the Authorize Attribute as follows.  
  
[Authorize(Roles = "Admins")]   
public ActionResult Edit(int id = 0)   
{   
    ProductDetails productdetails = db.ProductDetails.Find(id);   
    if (productdetails == null)   
    {   
        return HttpNotFound();   
    }   
    return View(productdetails);   
}   
  
The Authorize attribute makes sure that only users in the **Admins** group can access the Edit action method. At the time of development this seemed like an adequate measure.   
  
To allow non admin users access to the Details method, we have allowed two groups ‘Admins’ and ‘Users’, by decorating the action method as follows.

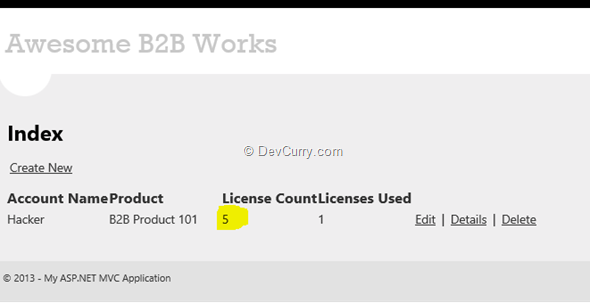
[Authorize(Roles="Admins,Users")]   
public ActionResult Details(int id = 0)   
{   
    ProductDetails productdetails = db.ProductDetails.Find(id);   
    if (productdetails == null)   
    {   
        return HttpNotFound();   
    }   
    return View(productdetails);   
}   
  
So because of the Authorize attributes, the hacker was upset in their first attempt.  
  
**Setting up the CSRF Hack**  
  
When the hacker tries to navigate to /ProductDetails/Edit, we redirect them to the login page. This gives them the hint that the page/action may exist. Based on that assumption, they setup a fake form with an increased number of licenses as follows  
  
<form name="csrfhackForm" method="post"   
    action="http://localhost:63577/ProductDetails/Edit/1">   
    <input type="hidden" name="Id" value="1" />   
    <input type="hidden" name="LicenseCount" value="5" />   
    <input type="hidden" name="LicenseUsedCount" value="1" />   
    <input type="hidden" name="AccountUserName" value="Hacker" />   
    <input type="hidden" name="ProductName" value="B2B Product 101" />   
< /form>   
< script type="text/javascript">   
    document.csrfhackForm.submit();   
< /script>   
  
The above form is malicious in the following ways

1. It submits to a URL that’s normally only available to Administrators.
2. Next it submits form data as soon as you load it
3. Redirects to Index page (because that’s the action on the Edit controller’s successful post method).
4. Posts data without leaving any trace of the hacker.

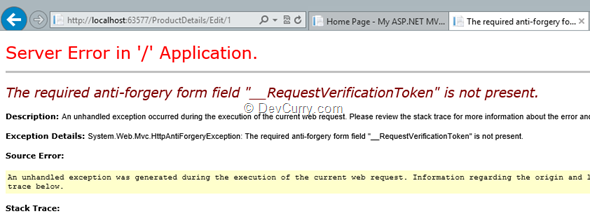
The Hacker now loads this form in a public website and a url is ready that will auto increment the License count.

### Authorized User – The missing Ingredient

As we mentioned earlier, a CSRF hack depends on the website’s trust on the user. Thus the hack is all set up, but it has one thing missing. The hacker can’t do anything unless an actual Admin clicks on the link. Using various social engineering techniques, hacker somehow gets the link to the Admin, say via an email.  
  
**Last line of defense and the breach**  
Let’s say the admin receives the malicious link in an email and clicks on the link, the final line of defense is the login page. The admin will be redirected to the login page and after logging in, because the form submit got thwarted, it will just present the Admin the Edit page. This will confuse the Admin but the hacker’s intent is still not accomplished.  
  
However things become easier if the Admin clicked on the ‘Remember me’ checkbox to keep an **authentication cookie alive** or was logged in to the site in another browser. In that case, clicking on the link will silently increase the license without the admin realizing it. And if the system logs changes, it will always log the administrator as the person who made the change. ATTACK SUCCESSFUL!!!

!!!  
  


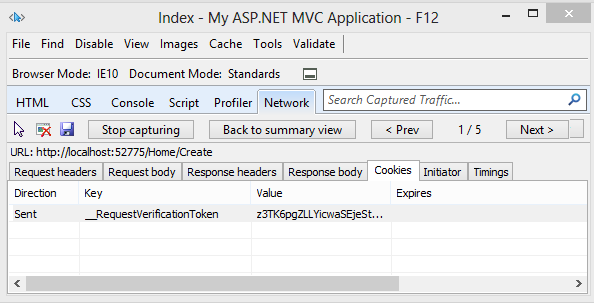
**Thwarting the Hacker**

Thankfully the solution to this hack is not far away or difficult to implement. This is based on the fact that the hacker created a third party form not hosted on the original site. Utilizing this information we can thwart the hacker. The easy way to do this is to use the ValidateAnitForgery token attribute in the ProductDetails post action method as follows  
  
[HttpPost]   
[Authorize(Roles = "Admins")]   
[ValidateAntiForgeryToken()]   
public ActionResult Edit(ProductDetails productdetails)   
{   
    if (ModelState.IsValid)   
    {   
        db.Entry(productdetails).State = EntityState.Modified;   
        db.SaveChanges();   
        return RedirectToAction("Index");   
    }   
    return View(productdetails);   
}   
  
To generate the AntiForgeryToken and the Cookie on the client side, we declare it as follows in the HTML form in the Edit.cshtml  
  
@using (Html.BeginForm()) {   
    @Html.ValidationSummary(true)   
    @Html.AntiForgeryToken()  
    <fieldset>   
        <legend>ProductDetails</legend>   
…   
  
This ensures that a form being posted to the server was actually generated by the same server. Thus fake forms that do not have the AntiForgeryToken from the correct server, gets rejected.  
  
  
  
As you can see, after adding the AntiForgeryToken, if you click on the malicious link, you get the above error instead.

## Web Stack Runtime XSRF mitigations

The ASP.NET Web Stack Runtime uses a variant of the [synchronizer token pattern](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#General_Recommendation:_Synchronizer_Token_Pattern) to defend against XSRF attacks. The general form of the synchronizer token pattern is that two anti-XSRF tokens are submitted to the server with each HTTP POST (In addition to the authentication token): one token as a cookie, and the other as a form value. The token values generated by the ASP.NET runtime are not deterministic or predictable by an attacker. When the tokens are submitted, the server will allow the request to proceed only if both tokens pass a comparison check.

The XSRF request verification *session token* is stored as an HTTP cookie and currently contains the following information in its payload:

* A security token, consisting of a random 128-bit identifier.  
  The following image shows the XSRF request verification session token displayed with the Internet Explorer F12 developer tools, Chrome code inspector +

## Generating the tokens

To generate the anti-XSRF tokens, call the [@Html.AntiForgeryToken](https://msdn.microsoft.com/en-us/library/dd470175.aspx) method from an MVC view or @AntiForgery.GetHtml() from a Razor page.

## Validating the tokens

To validate the incoming anti-XSRF tokens, the developer as shown above includes a [ValidateAntiForgeryToken](https://msdn.microsoft.com/en-us/library/system.web.mvc.validateantiforgerytokenattribute(VS.108).aspx) attribute on her MVC action or controller, or she calls @AntiForgery.Validate() from her Razor page. The runtime will perform the following steps:

1. The incoming session token and field token are read and the anti-XSRF token extracted from each. The anti-XSRF tokens must be identical per step (2) in the generation routine.
2. If the current user is authenticated, her username is compared with the username stored in the field token. The usernames must match.