Building Events Management System

ASP.NET MVC CASE STUDY

VENKAT SHIVA REDDY

ASP.NET MVC Practical Hands-On Lab

A **practical MVC hand-on lab (tutorial)** that gives rich experience in **building data-driven Web applications with ASP.NET MVC**, ASP.NET MVC 5, SQL Server, C#, Visual Studio, Entity Framework (code first), AJAX, jQuery, Bootstrap and other modern technologies and tools. In this lab you will create a web-based event management system, from zero to fully working web application. Enjoy this **ASP.NET MVC tutorial for beginners**. The source code is intentionally given as images to avoid copy-pasting.

ASP.NET MVC Lab – Web Based Events Management System

The goal of this lab is to learn how to **develop ASP.NET MVC data-driven Web applications**. You will create MVC application to list / create / edit / delete events. The recommended development IDE to use for this Lab is **Visual Studio 2013 or later** with the latest available updates + SQL Server 2012. Let's start building the event management system in ASP.NET MVC step by step.

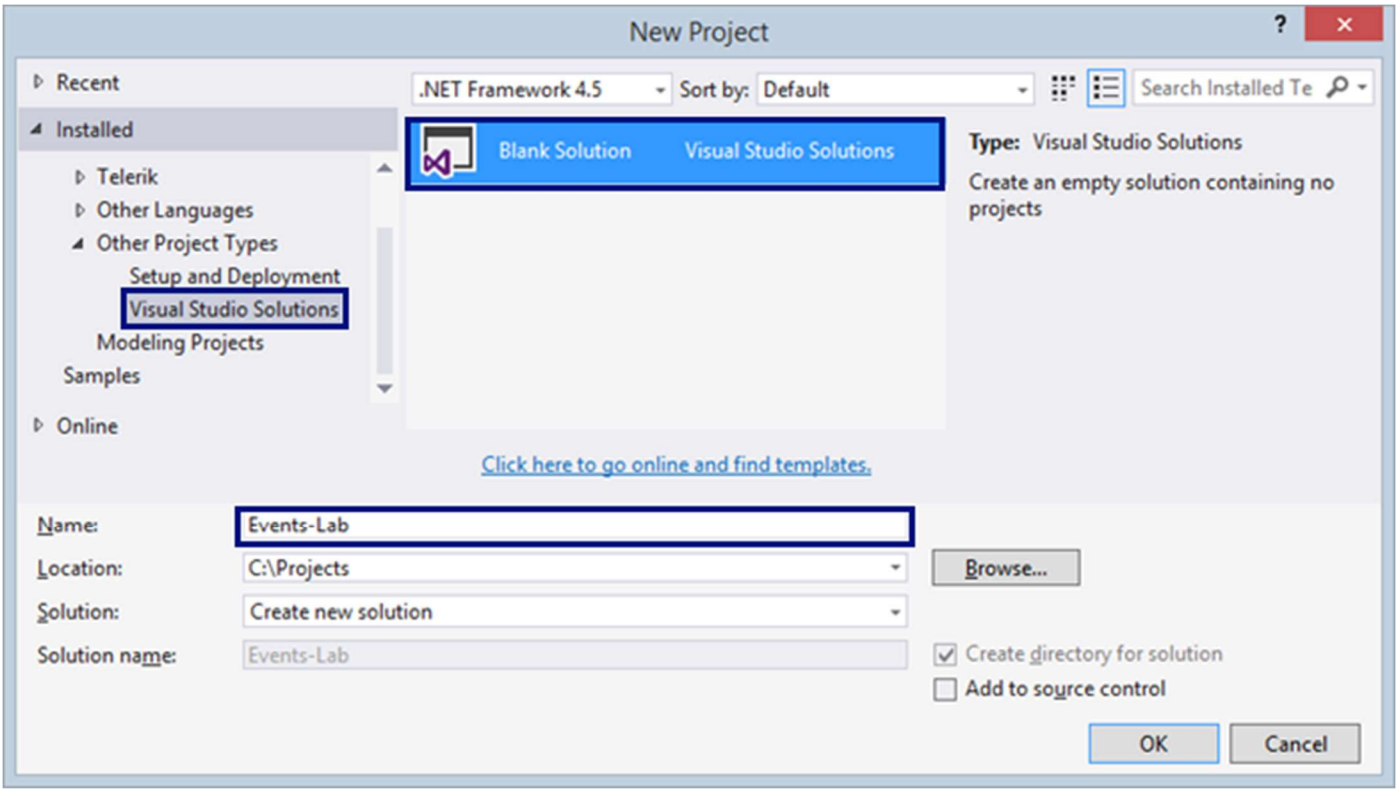
Project Assignment

Design and implement a **Web based event management system**.

- **Events** have **title**, **start date** and optionally **start time**. Events may have also (optionally) **duration**, **description**, **location** and **author**. Events can be **public** (visible by everyone) and **private** (visible to their owner author only). Events may have **comments**. **Comments** belong to certain event and have **content** (text), **date** and optional **author** (owner).
- **Anonymous** users (without login) can **view all public events**. The home page displays all public events, in two groups: upcoming and passed. Events are shown in **short form** (title, date, duration, author and location) and have a **[View Details]** button, which loads dynamically (by **AJAX**) their description, comments and [Edit] / [Delete] buttons.
- Anonymous users can **register** in the system and **login** / **logout**. Users should have mandatory **email**, **password** and **full name**. User's email should be unique. User's password should be non-empty but can be just one character.
- **Logged-in users** should be able to **view their own events**, to **create new events**, **edit their own events** and **delete their own events**. Deleting events requires a **confirmation**. Implement client-side and sever-side data validation.
- A special user "**`admin@admin.com`**" should have the role "**`Administrator`**" and should have full permissions to **edit** / **delete events** and **comments**.

Step 1. Empty Visual Studio Solution

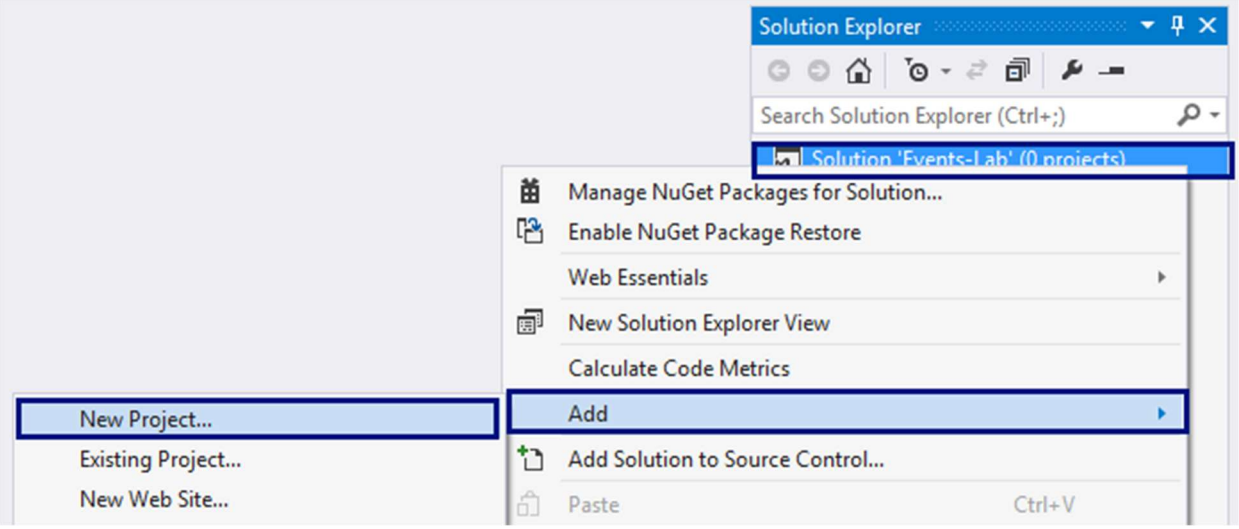
Create a new empty Visual Studio Solution named "**`Events-Lab`**":



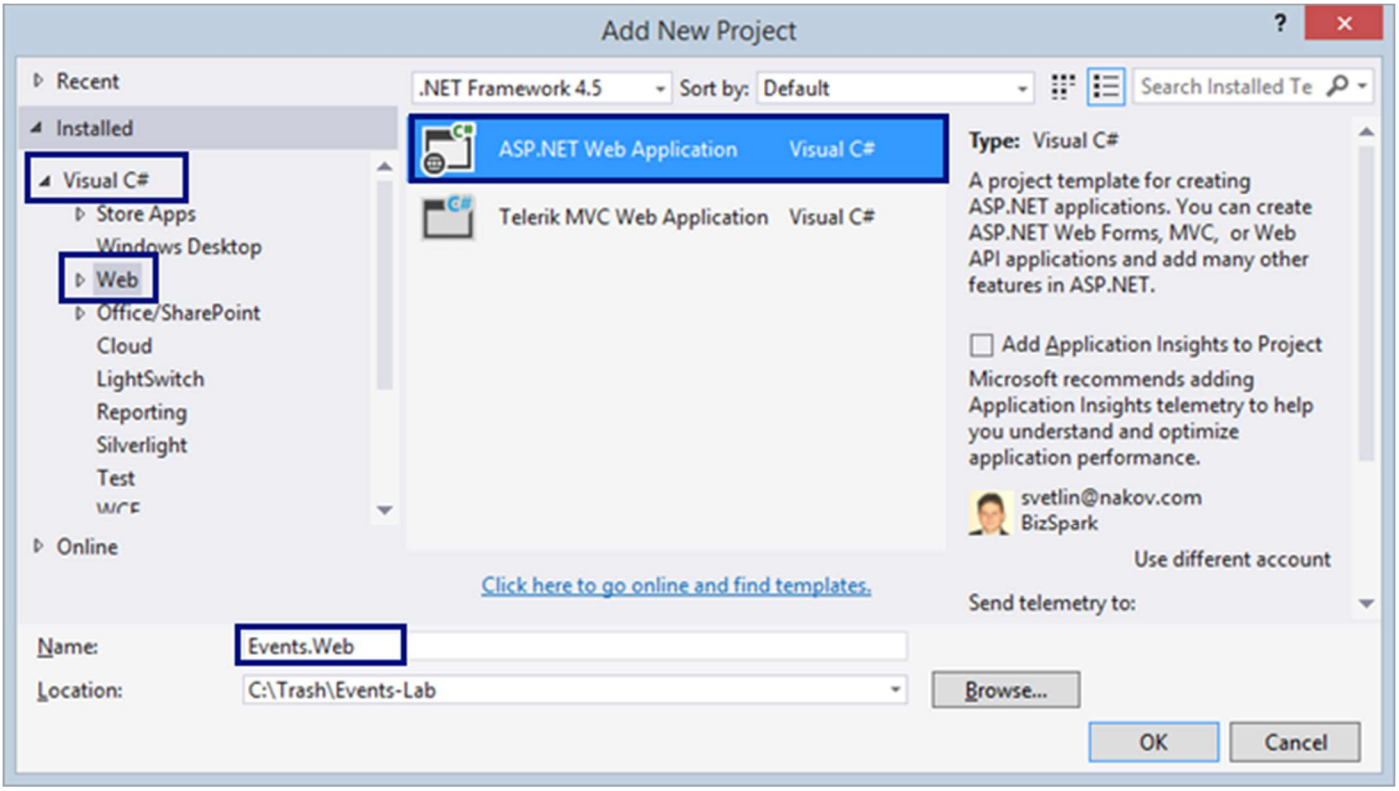
This VS solution will hold your projects: **data layer project** and **ASP.NET Web application project**.

Step 2. Empty MVC Project with User Accounts

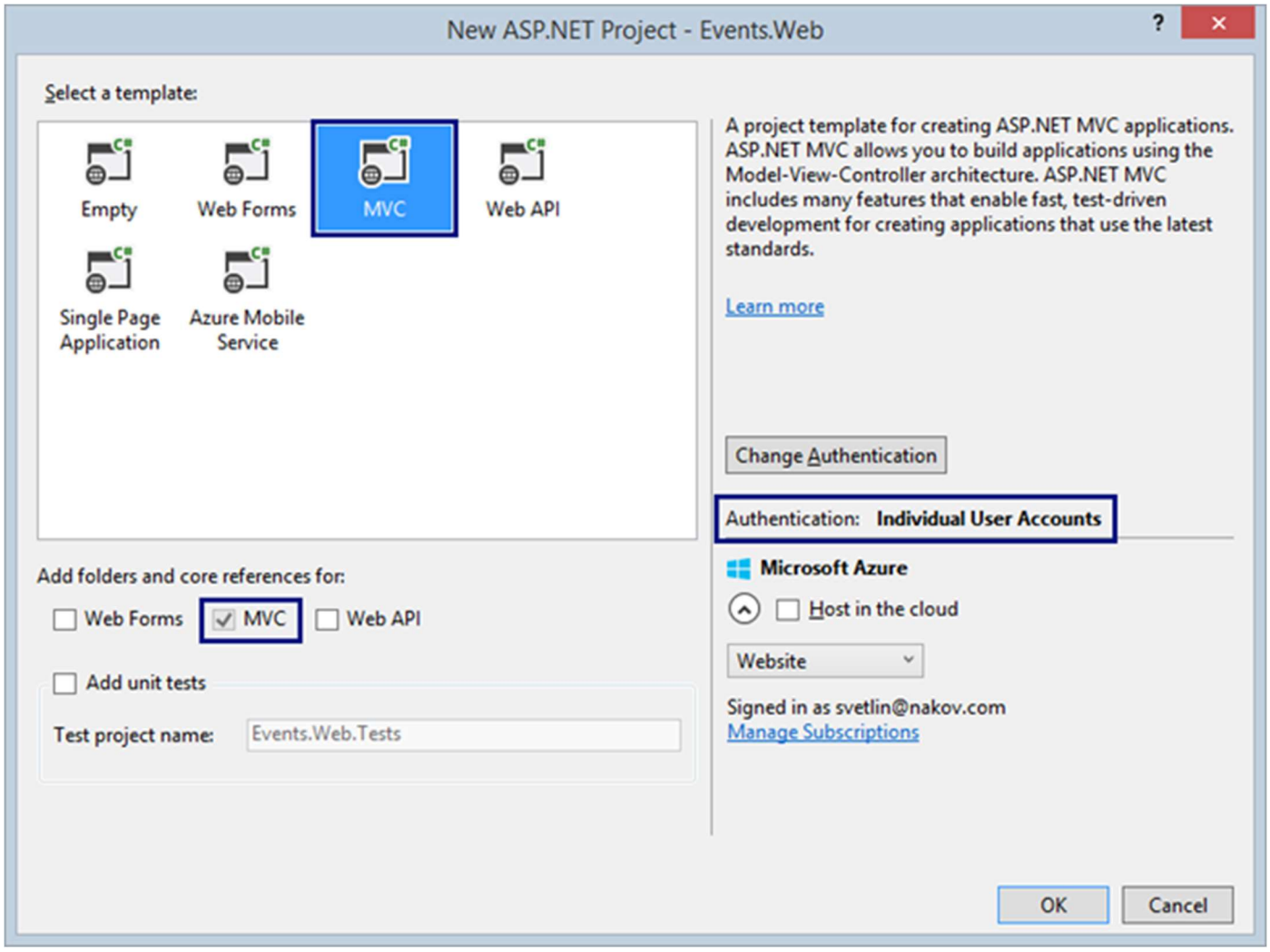
Create and **empty ASP.NET MVC project** by the default Visual Studio template with built-in user accounts and authentication. First, click on the solution, choose [Add] → [New Project…]:



Choose [Visual C#] → [Web] → [**ASP.NET Web Application**]. Name the application "**`Events.Web`**". It will hold your ASP.NET MVC Web application (controllers, views, view models, etc). The data access layer (entity classes + Entity Framework data context) will be separated in another application.



Next, choose the "**MVC**" project template + "**Individual User Accounts**" as authentication type.



Visual Studio will **generate an ASP.NET MVC application** by template, which will serve as **start** (skeleton) for your Events management system. The application uses a typical Web development stack for .NET developers: **ASP.NET MVC 5** as core Web development framework, **Bootstrap** as front-end UI framework, **Entity Framework** as data layer framework, **ASP.NET Identity system** as user management framework, and **SQL Server** as database.

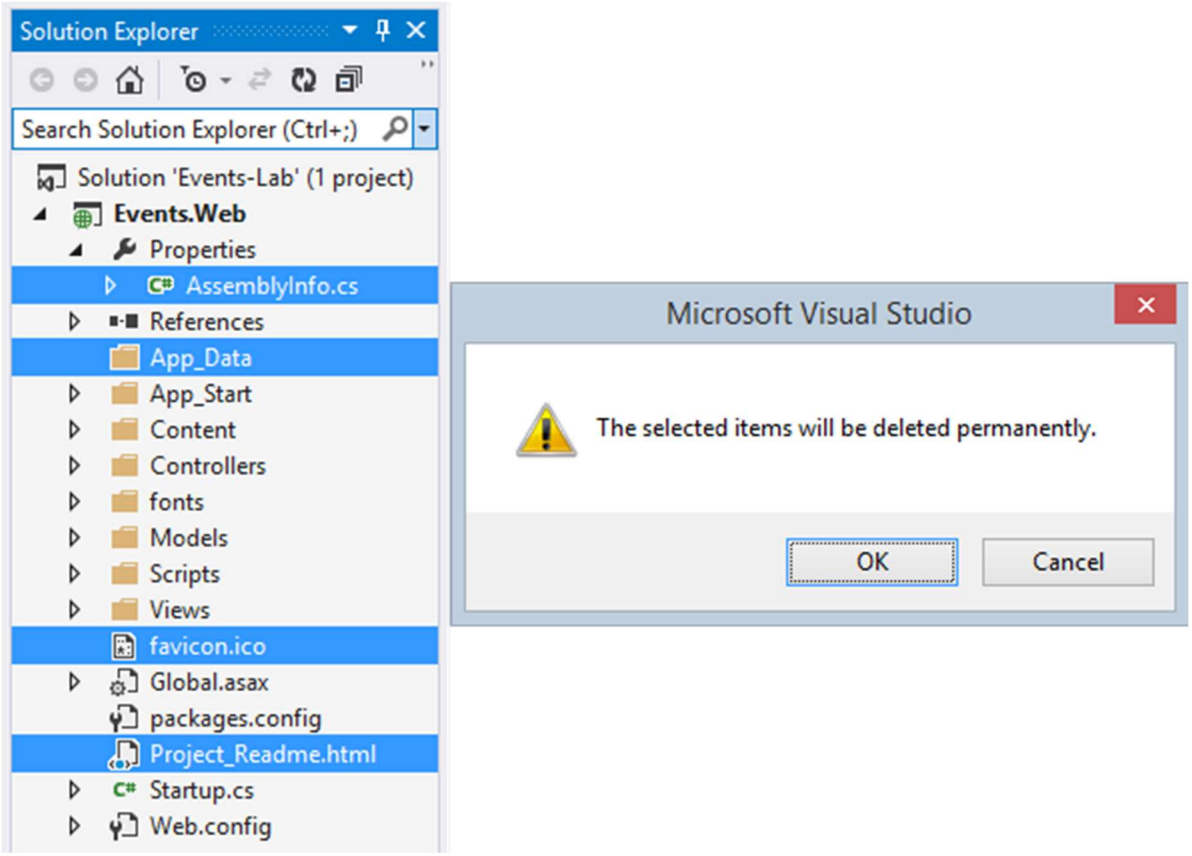
Step 3. Customize the MVC Application

The project generated by the Visual Studio template needs many adjustments. Let's customize the generated code for your Events management system.

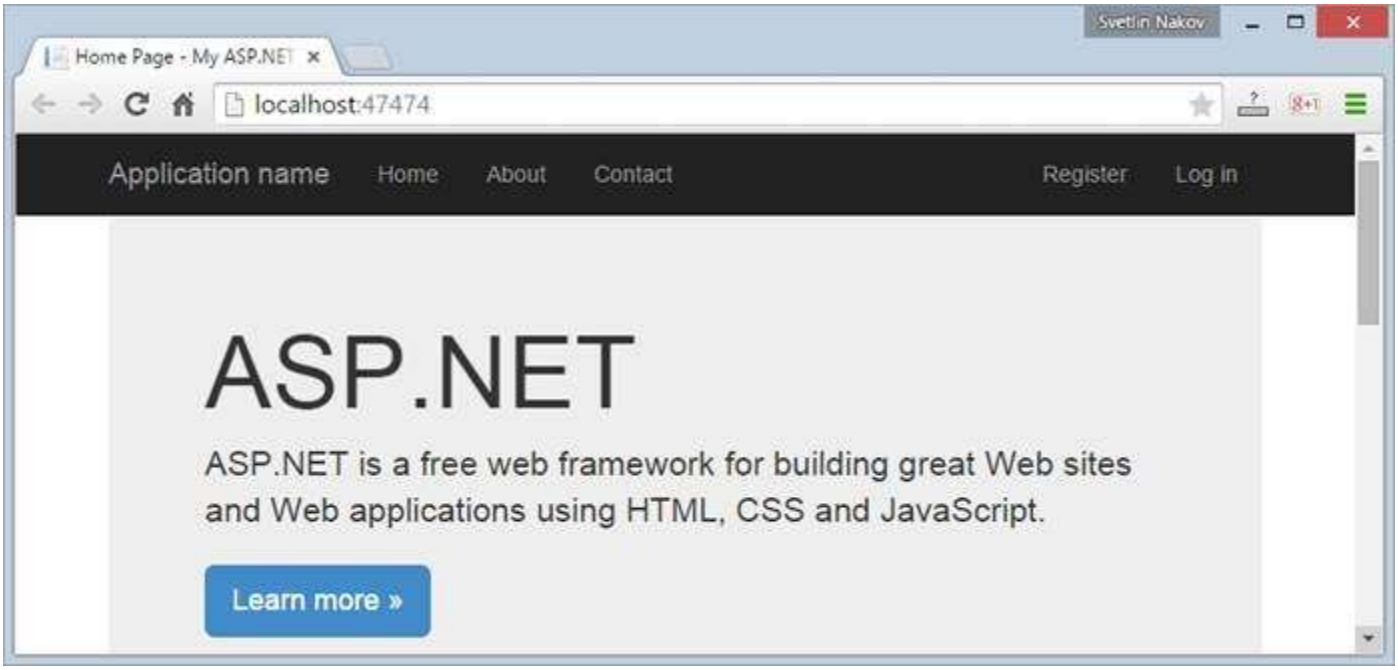
1. Edit the **connection string** in the **`Web.config`** file to match your database server settings. In our case, we will use a database "**`Events`**" in MS SQL Server 2014 Local DB: **`(LocalDb)\MSSQLLocalDB`**:

```
Web.config
<connectionStrings>
  <add name="DefaultConnection"
    connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=Events;Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

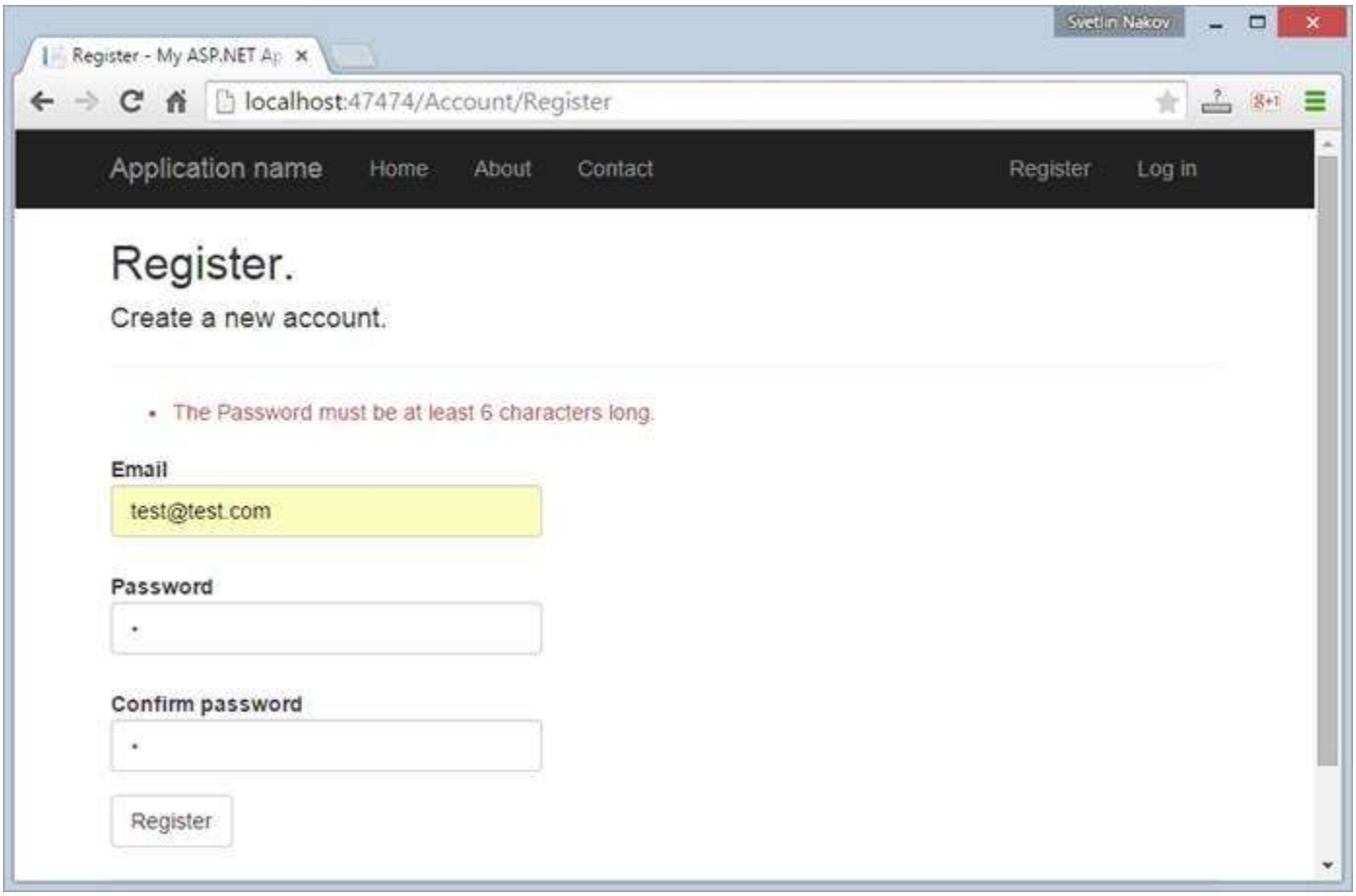
2. **Delete some unneeded files**:
   - **`AssemblyInfo.cs`** – holds application metadata (not needed at this time)
   - **`App_Data`** folder – holds the application database (not needed, we will use SQL Server Local DB)
   - **`favico.ico`** – holds the Web application icon used in the browser location bar. We will not use it. We can add it later.
   - **`Project_Readme.html`** – holds instructions how to start out MVC project



3. Build and **run the application** to check whether it works, e.g. by clicking **[Ctrl + F5]**:

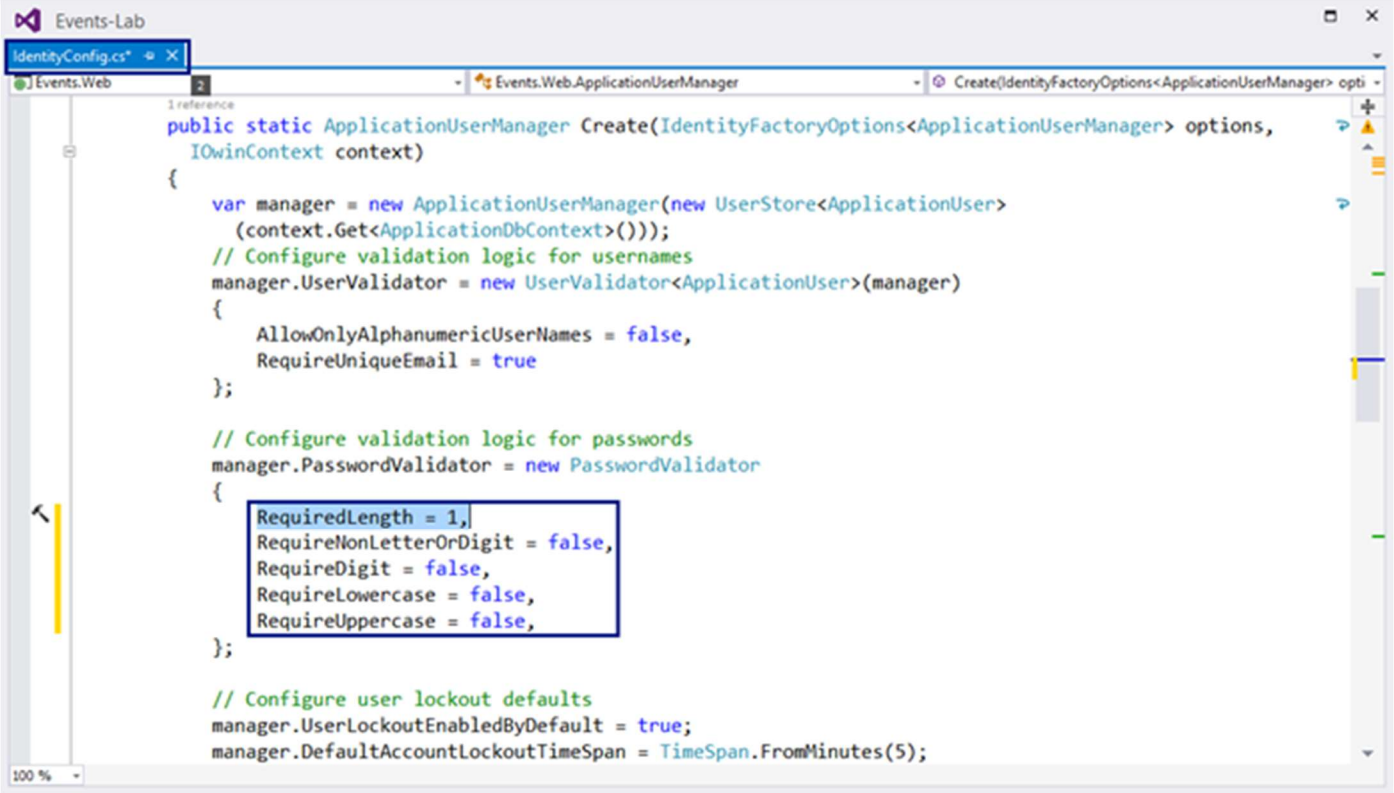


4. Test the **user registration** functionality. You should have out-of-the-box **user registration**, **login** and **logout**:

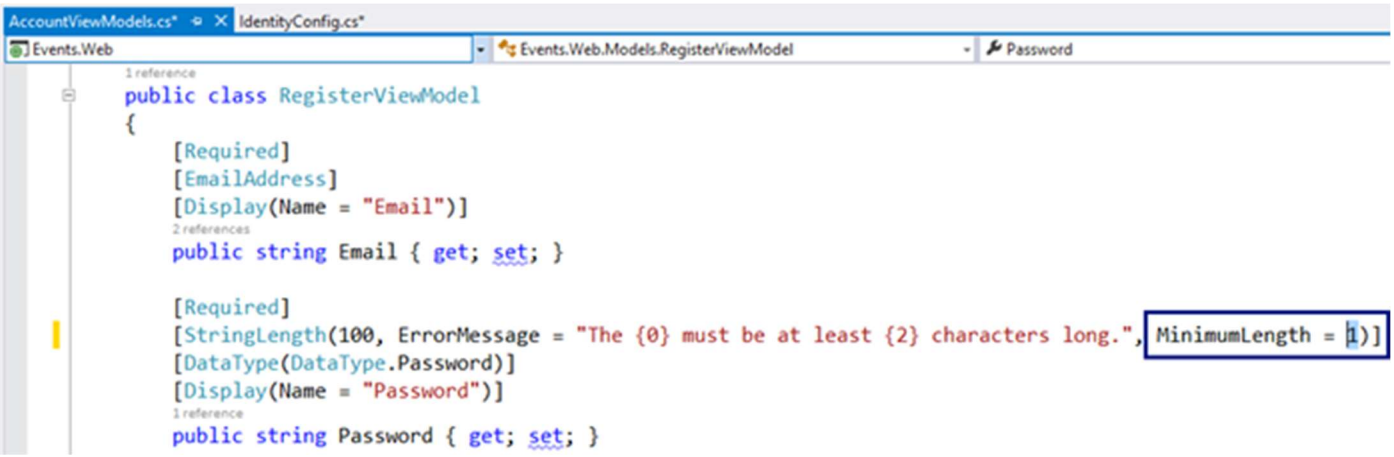


You have a problem. By default, the MVC application is configured to use **too complex passwords**. This will make hard the testing during the development.

5. Let's **change the password validation policy**:
   - Modify the **`PasswordValidator`** in the file **`App_Start\IdentityConfig.cs`**:



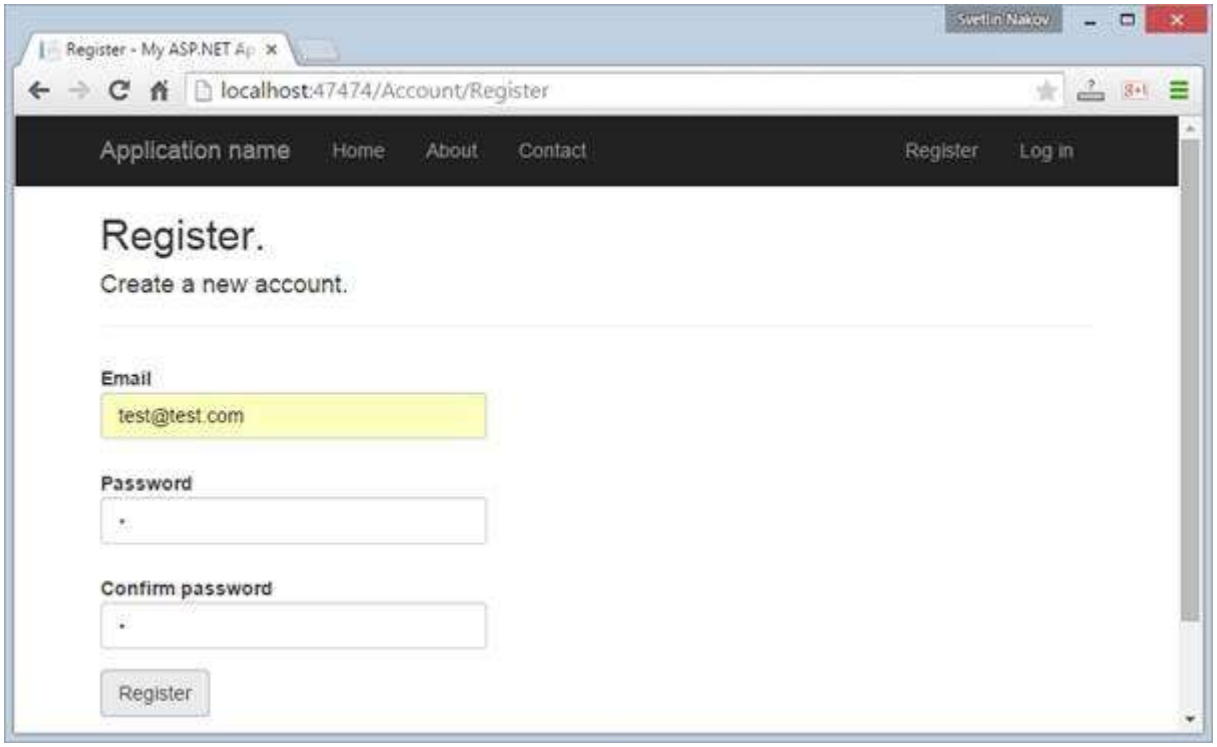
   - Modify the **password minimum length** in the class **`RegisterViewModel`**. It is located in the file **`Models\AccountViewModels.cs`**:



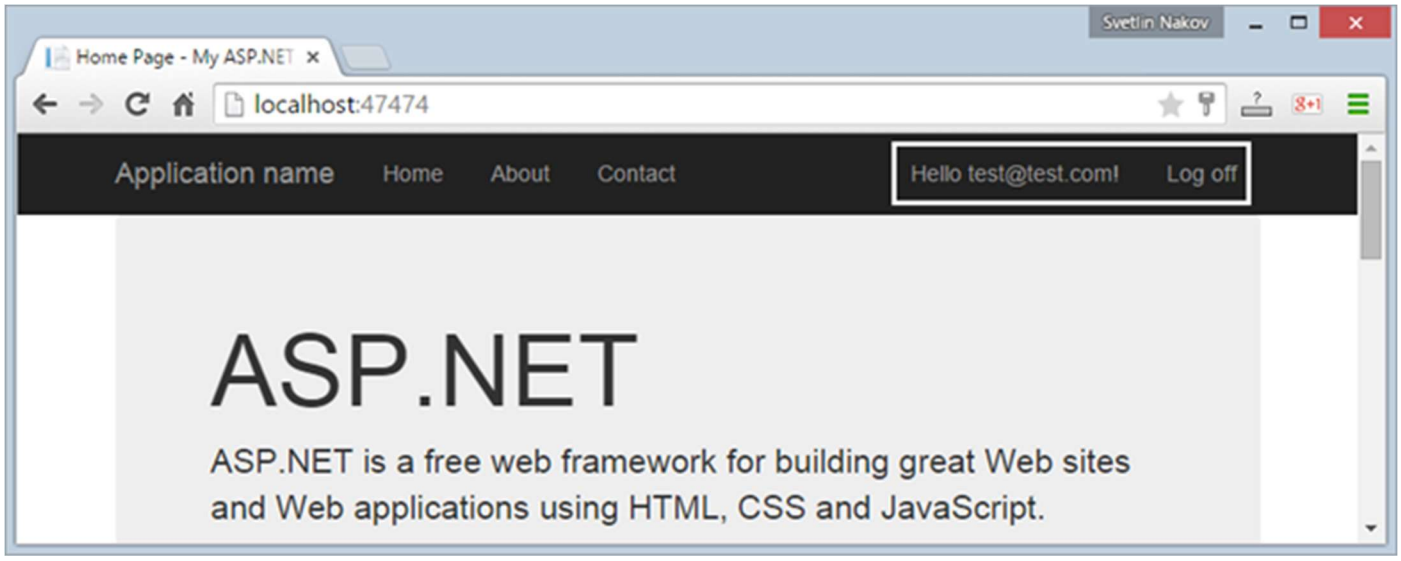
   - Modify the **password minimum length** in the **`SetPasswordViewModel`** and **`ChangePasswordViewModel`** classes, located in the file **`Models\ManageViewModels.cs`**:

```
ManageViewModels.cs*  AccountViewModels.cs*  IdentityConfig.cs*
Events.Web  Events.Web.Models.ChangePasswordViewModel  NewPassword

1 reference
public class ChangePasswordViewModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Current password")]
    1 reference
    public string OldPassword { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 1)]
    [DataType(DataType.Password)]
    [Display(Name = "New password")]
    1 reference
    public string NewPassword { get; set; }
```

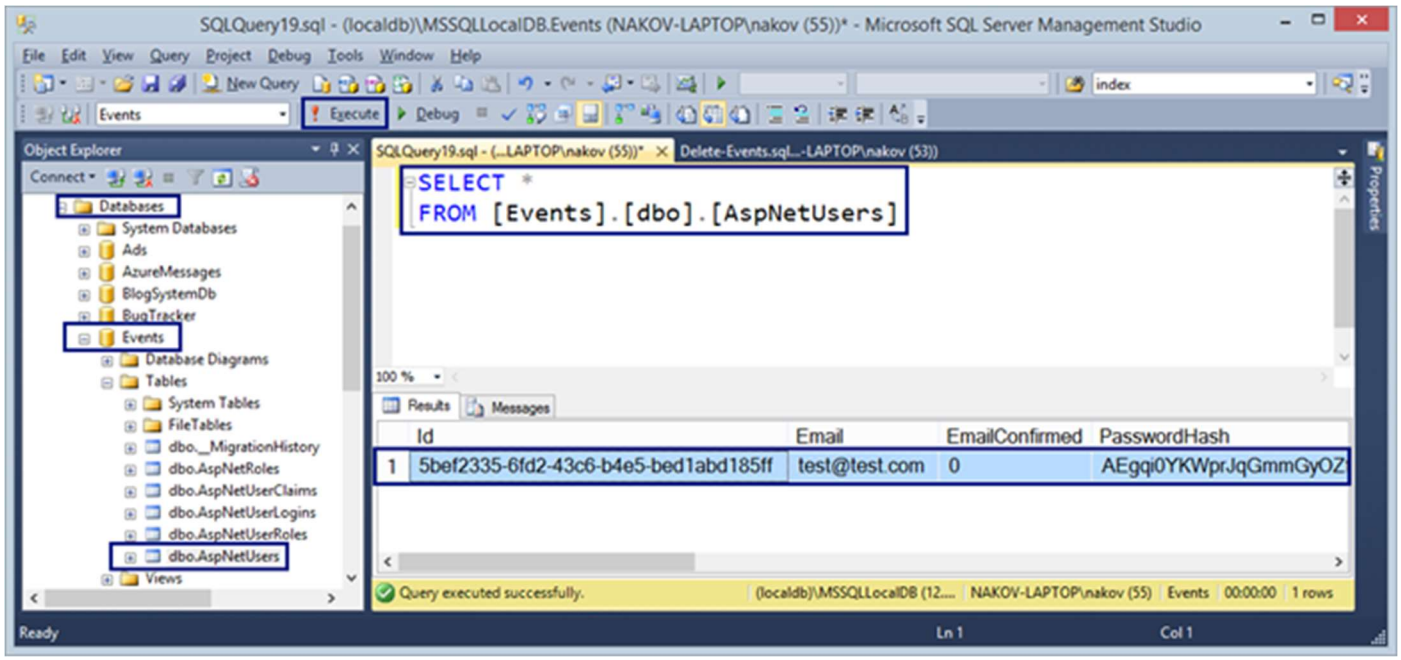
6. Recompile and run the application again. **Test the user registration**, **login** and **logout** again. E.g. try to register user "**`test@test.com`**" with password "**`1`**":



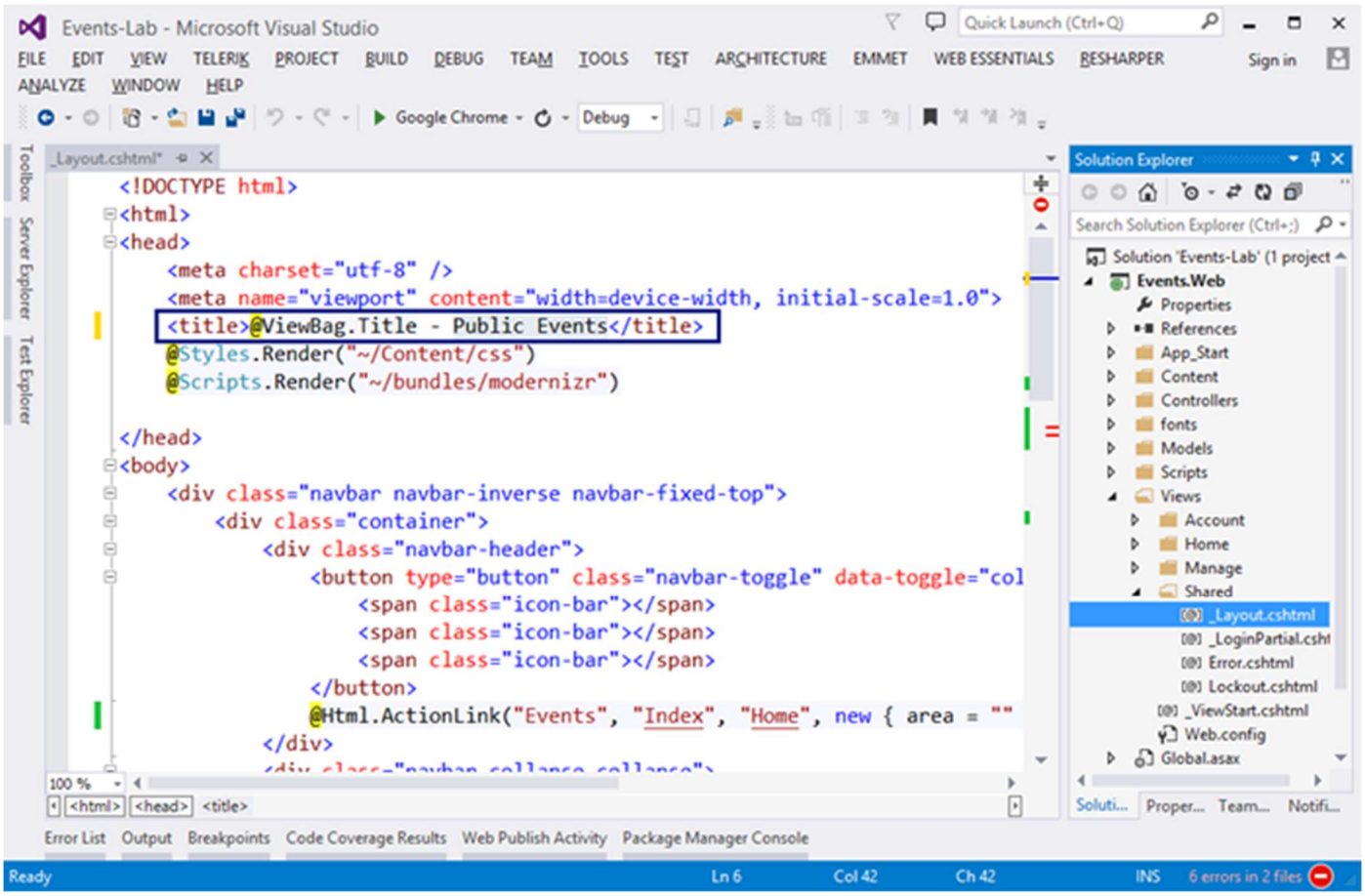
After a bit of waiting, Entity Framework will **create the database schema** in SQL Server and the MVC application will **register the user** in the database and will login as the new user:



7. Open the **database** to ensure it works as expected. You should have a database "**`Events`**" in the MS SQL Server Local DB, holding the **`AspNetUsers`** table, which should hold your registered **user account**:



8. Now let's edit the **application layout** view. Edit the file **`\Views\Shared\_Layout.cshtml`**.
   - Change the **application's title**:



- Change the **navigation menus**. Change the home page link. **Replace** the "**About**" and "**Contact**" links with the code shown below. The goal is to put "**My Events**" and "**Create Event**" links for logged-in users only. Anonymous site visitors will have only "Events" link to the home page, while logged in users will be able to see their events and create new events:

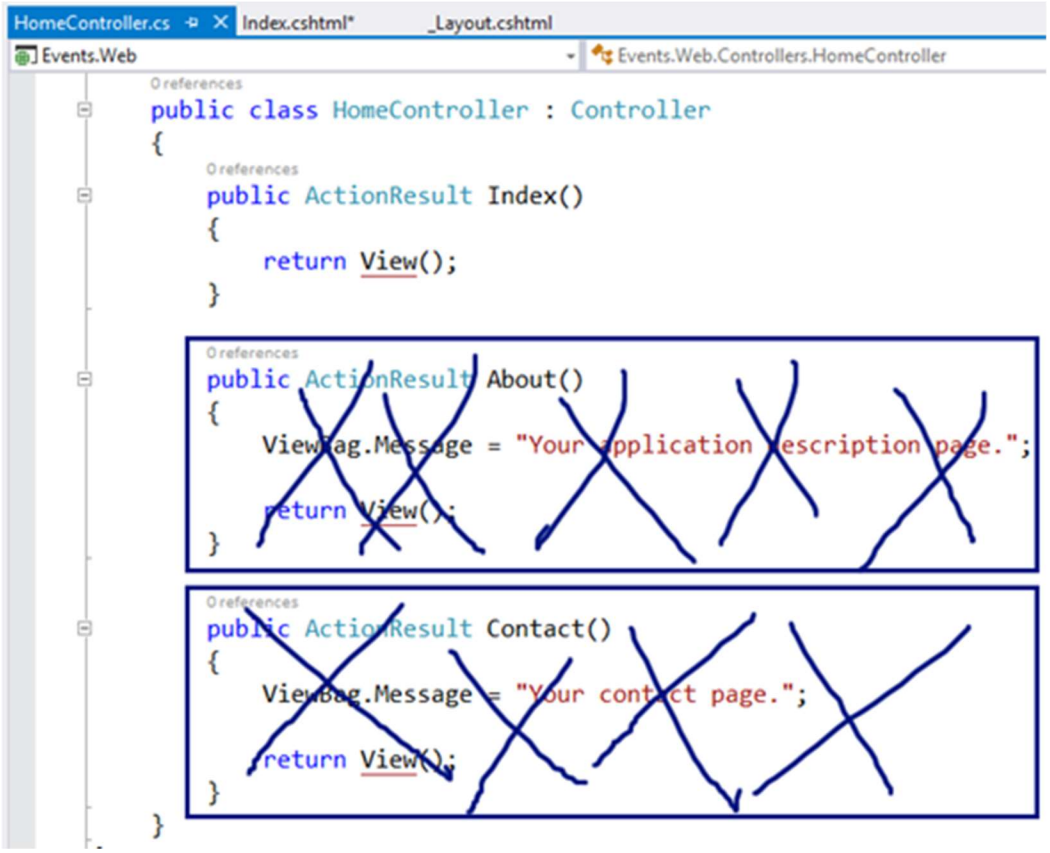
```
_Layout.cshtml
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Events", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    @if (Request.IsAuthenticated)
                    {
                        <li>@Html.ActionLink("My Events", "My", "Events")</li>
                        <li>@Html.ActionLink("Create Event", "Create", "Events")</li>
                    }
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
```

- Change the **application footer** text as well:

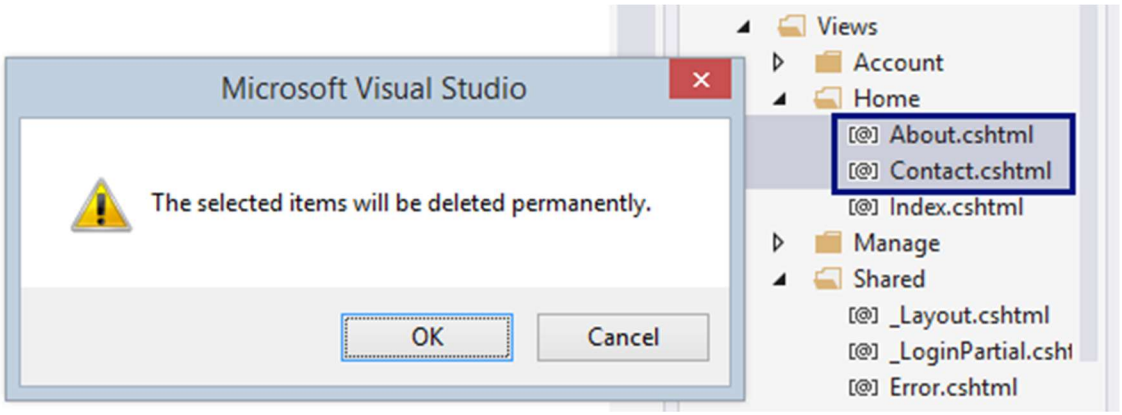
```
_Layout.cshtml*
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - Events Application</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

- Remove the junk code from the home controller's default view **`\Views\Home\Index.cshtml`**:

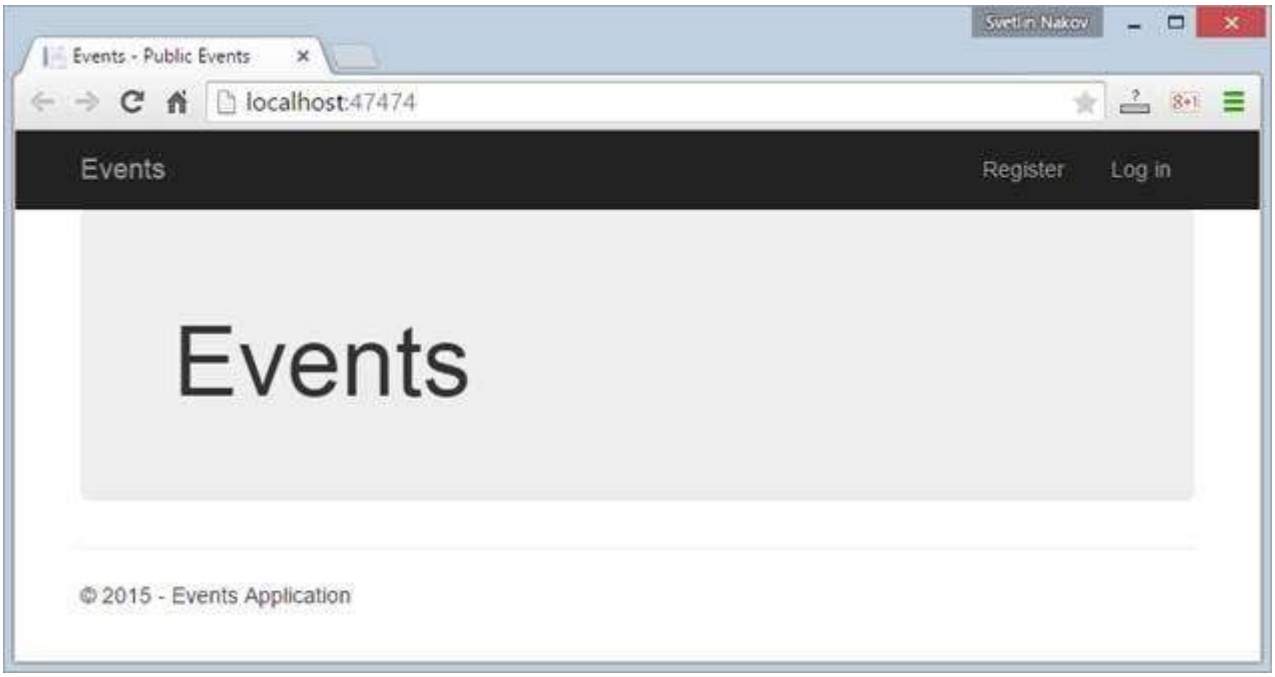
- **Remove** the unneeded **`HomeController`**'s actions "**About**" and "**Contacts**":



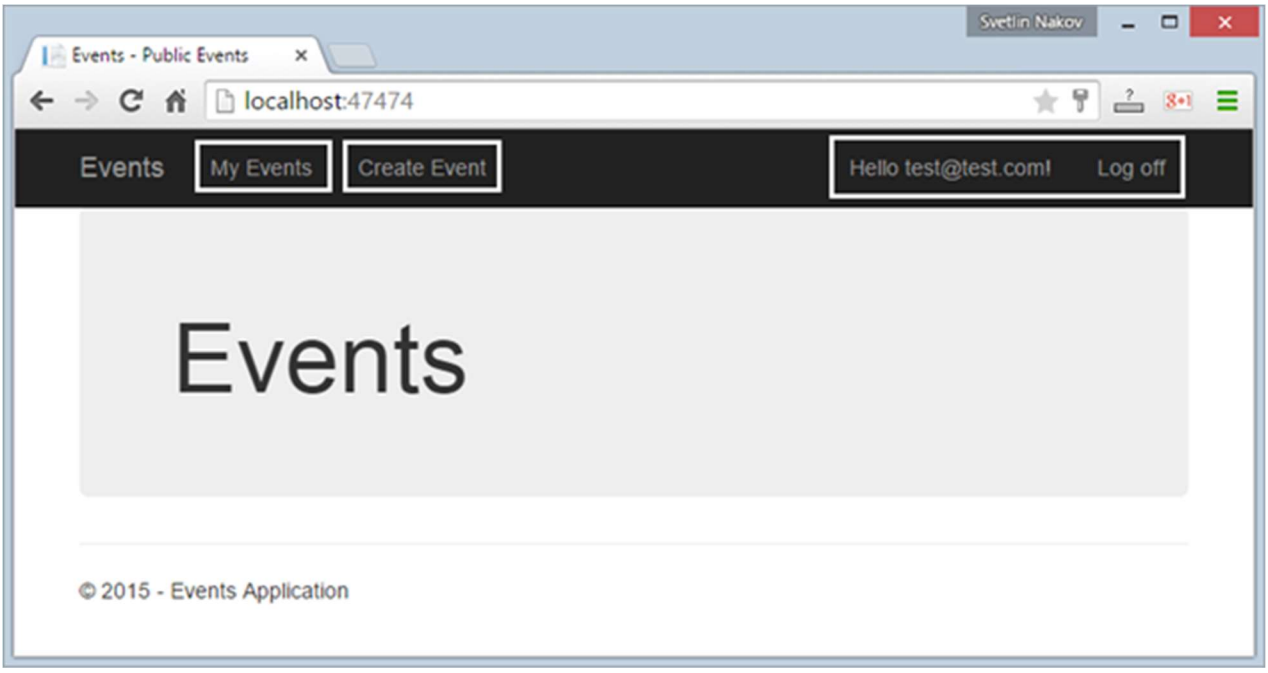
- **Delete** the unneeded Home controller's views **`About.cshtml`** and **`Contact.cshtml`**:



9. Now **run the application** to test it again. It should look like this for anonymous users:



After login, the application should display the **user's navigation menus**. It should look like this:

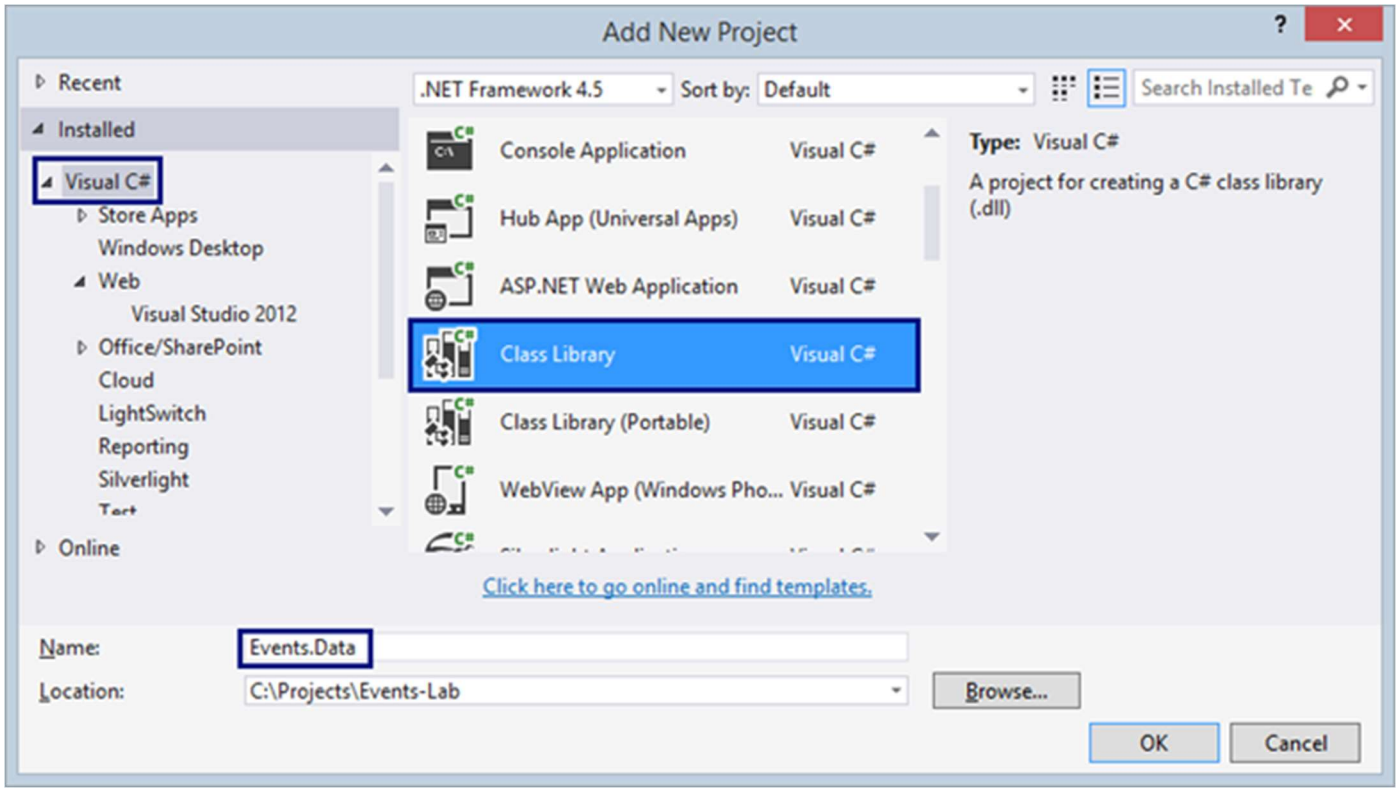


Now you are ready for the real development on the Events management system. Let's go.

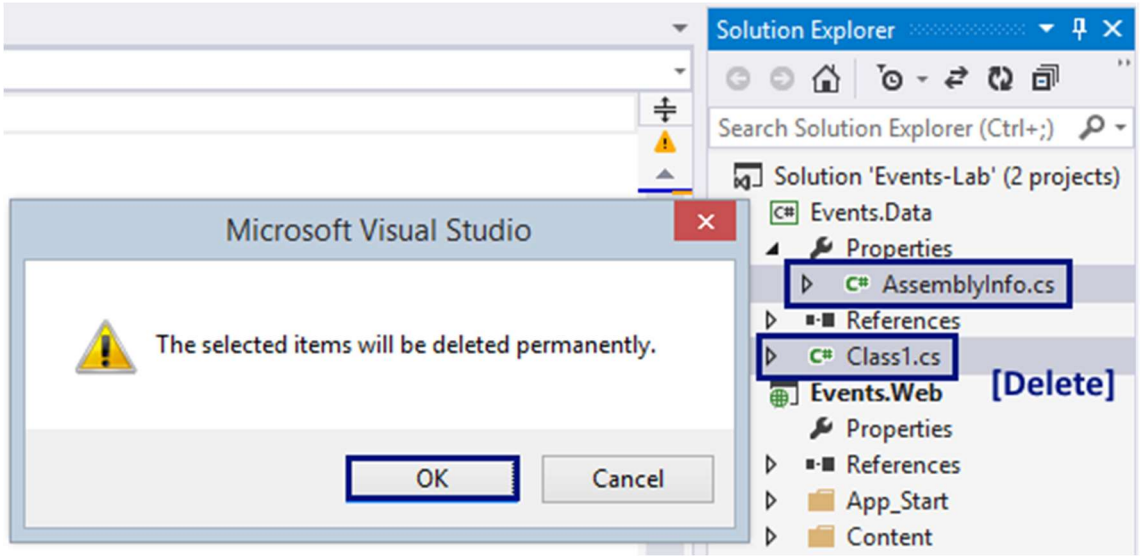
Step 4. Separate the EF Data Model from the MVC Application

The **application structure** is not very good. It mixes the entity **data models** with the Entity Framework data context and the MVC **view models** for the views and MVC **input models** for mapping the forms. Let's restructure this.

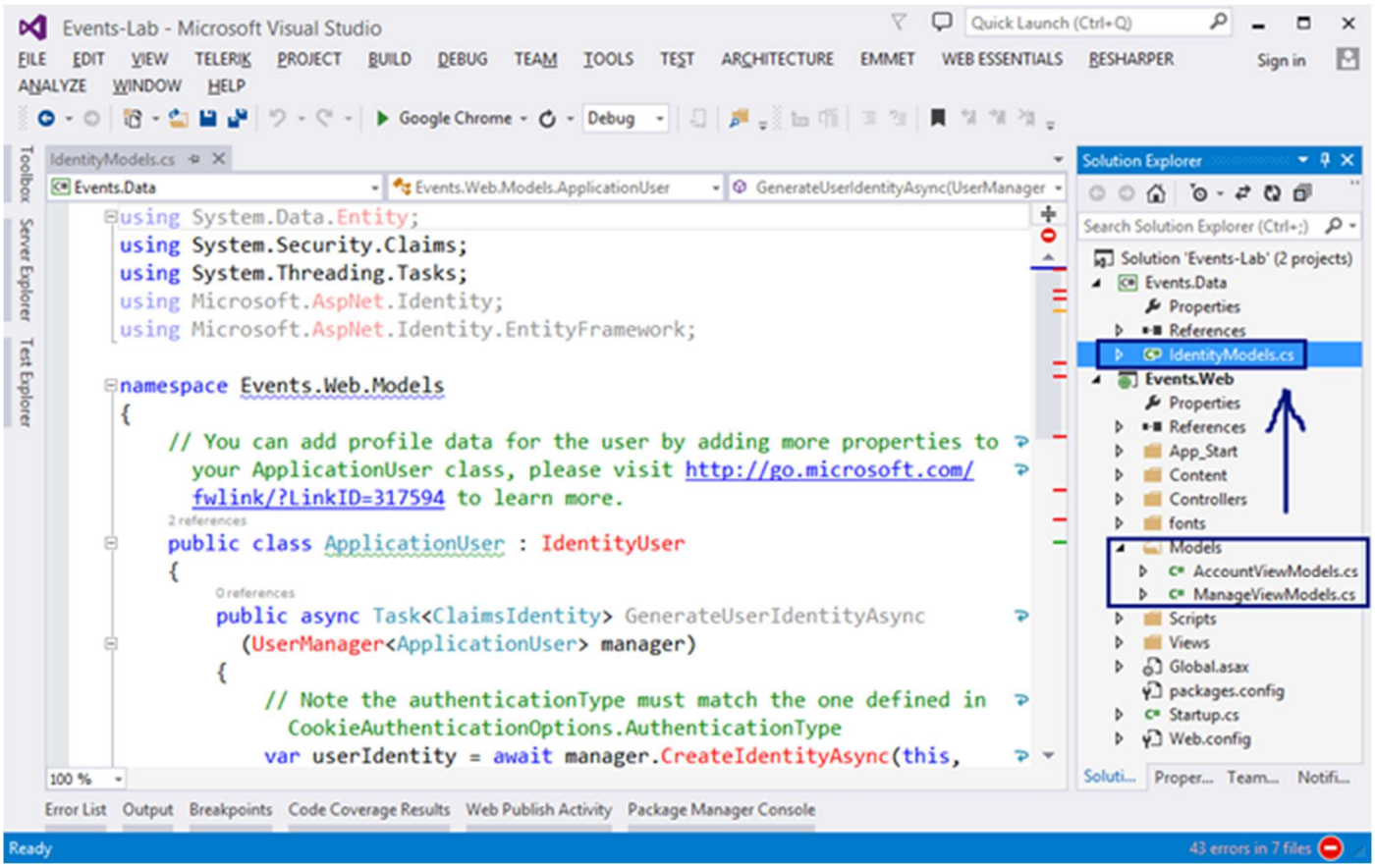
1. Create a new Class Library project "**`Events.Data`**" in your solution in Visual Studio:



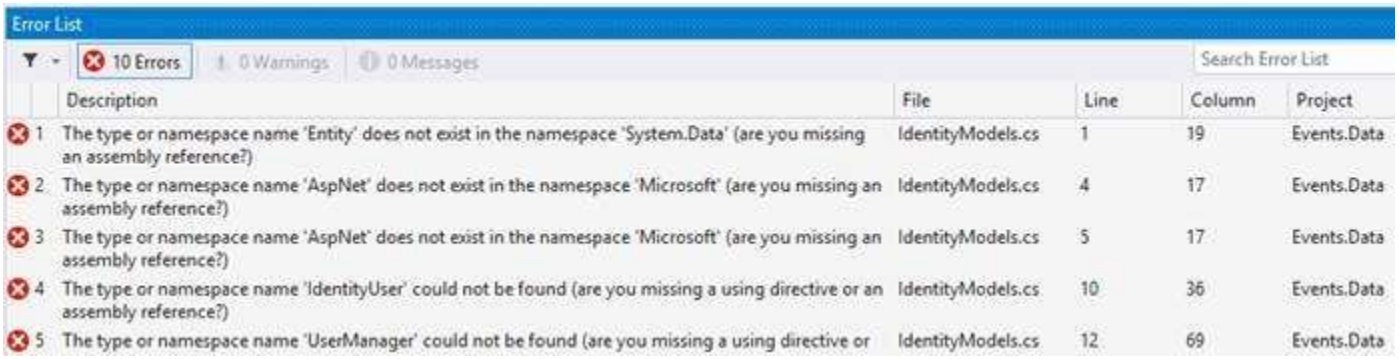
2. **Delete the unneeded files** from the **`Events.Data`** project:



3. Move the file **`Events.Web\Models\IdentityModels.cs`** to the new project **`Events.Data`**. You can use **[Edit] → [Cut] → [Paste]**.

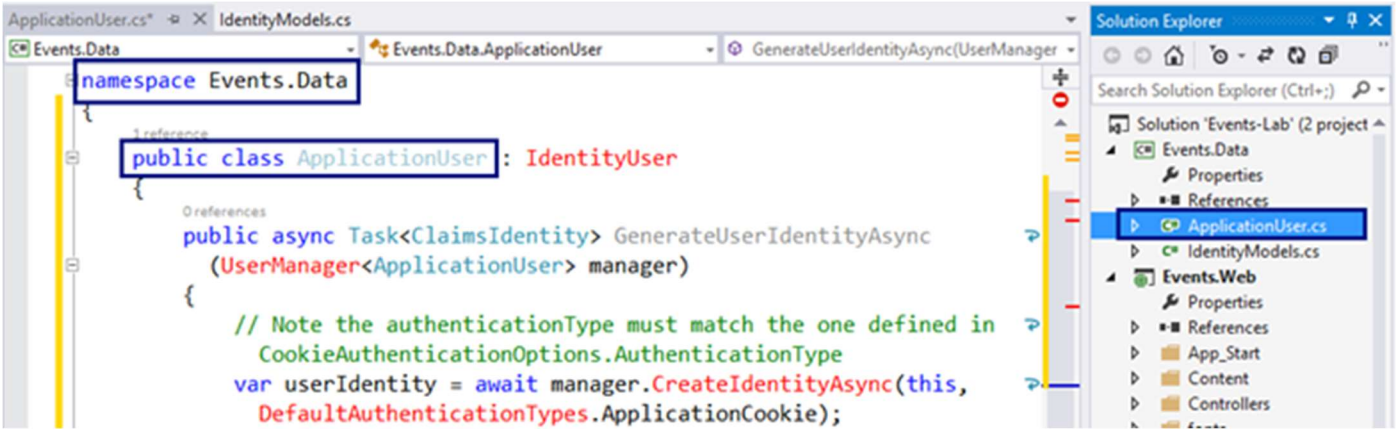


4. Now the Visual Studio Solution **will not compile**. Many errors will be shown if we rebuild with **[F6]**:

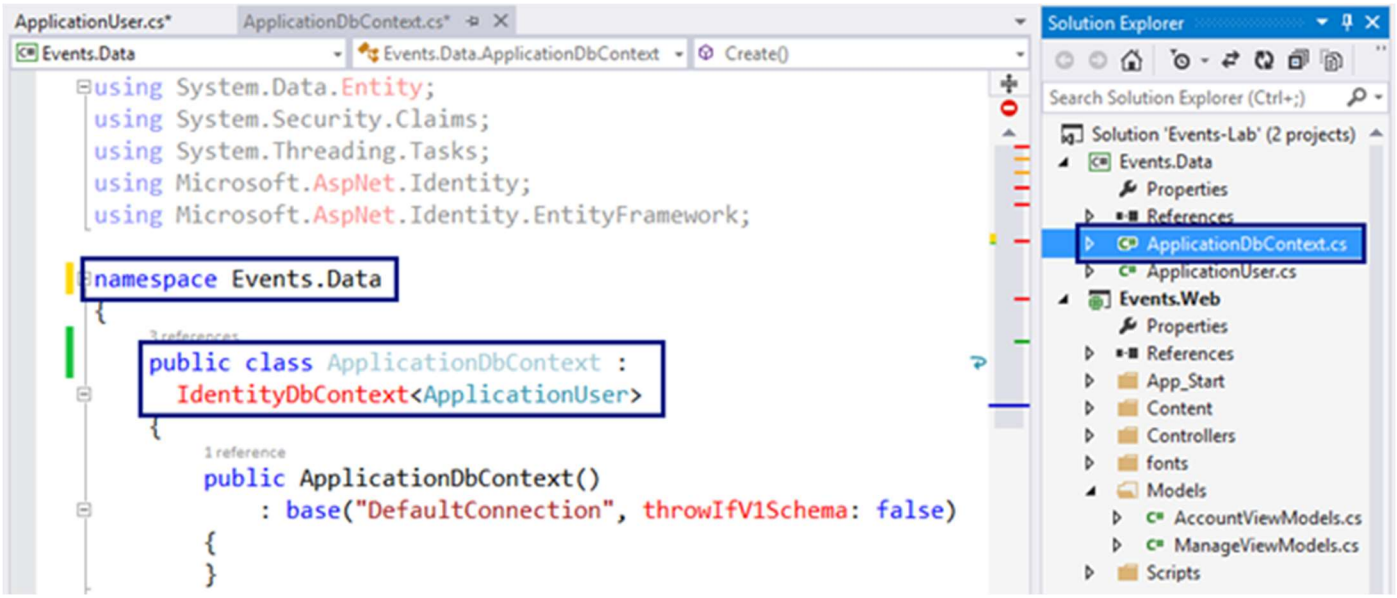


We need to **fix some issues**: class names, file names, library references, project references, etc.

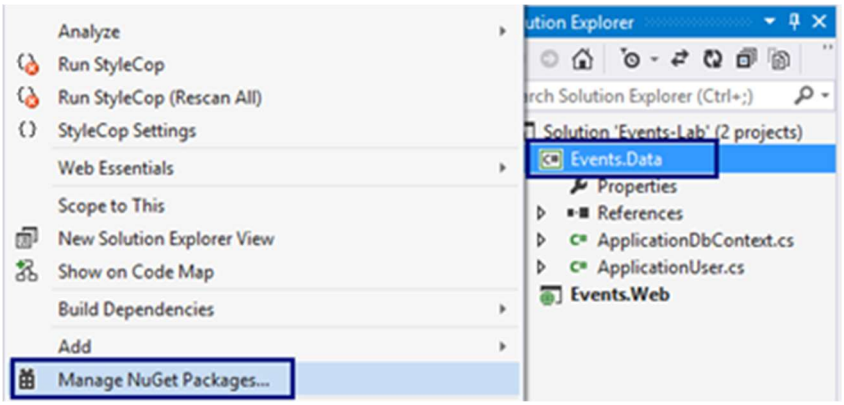
5. Extract the class **`ApplicationUser`** into a file named "**`ApplicationUser.cs`**". Change its **namespace** to "**`Events.Data`**". Your code should look like this:

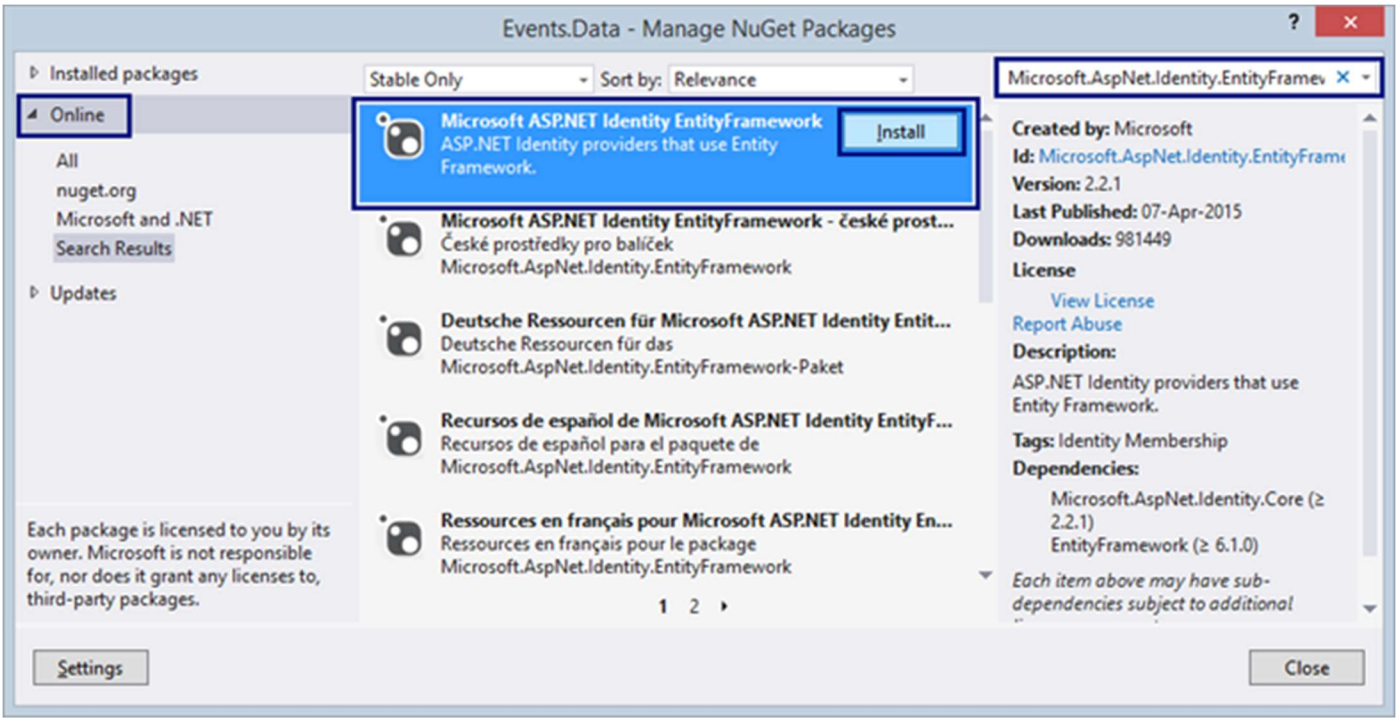


6. Rename the **`IdentityModel.cs`** file to **`ApplicationDbContext.cs`**. This file will hold your Entity Framework database context – the class**`ApplicationDbContext`**. Change its namespace to **`Events.Data`**. Your code should look like this:



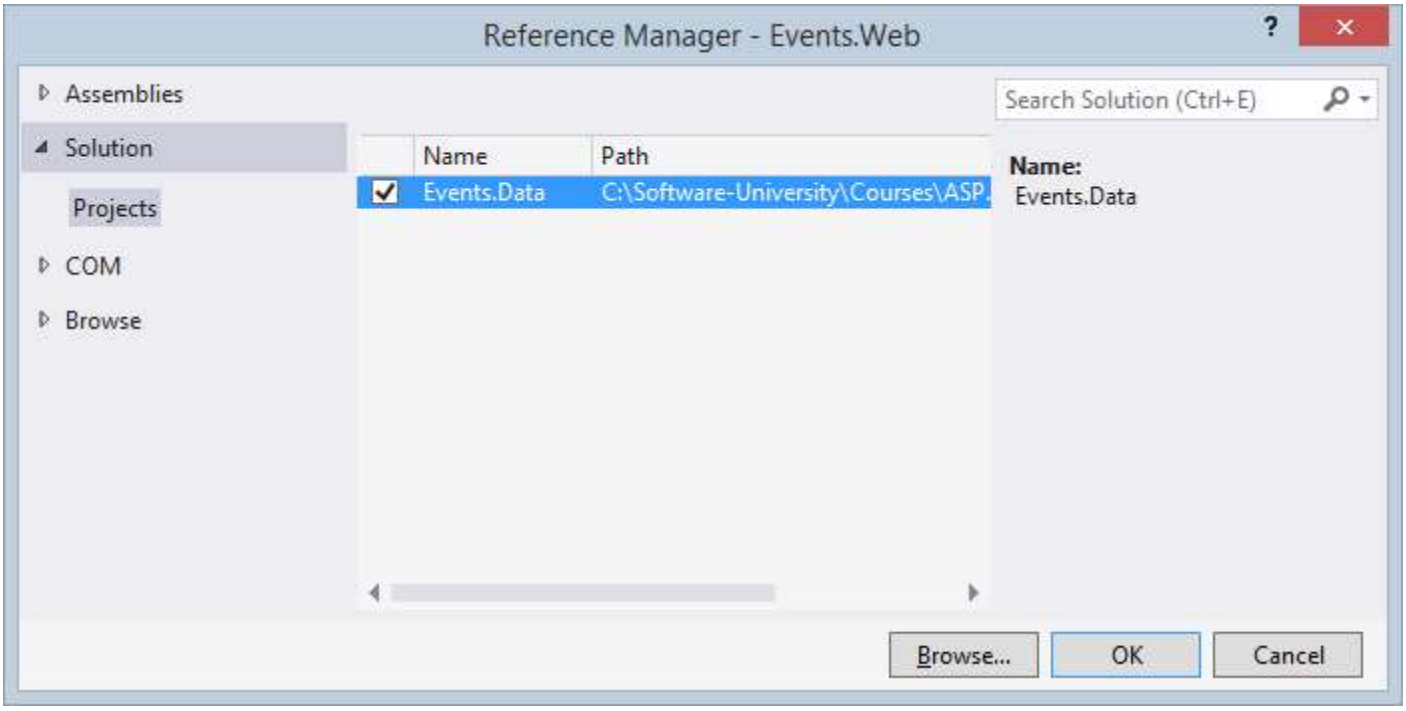
7. Your code will **still not compile**. Now you should **add NuGet references** to the missing libraries: **Entity Framework**, **Microsoft.AspNet.Identity.Core** and**Microsoft.AspNet.Identity.EntityFramework**:





It is enough to reference the package "**`Microsoft.AspNet.Identity.EntityFramework`**". The others depend on it and will be installed by dependency.

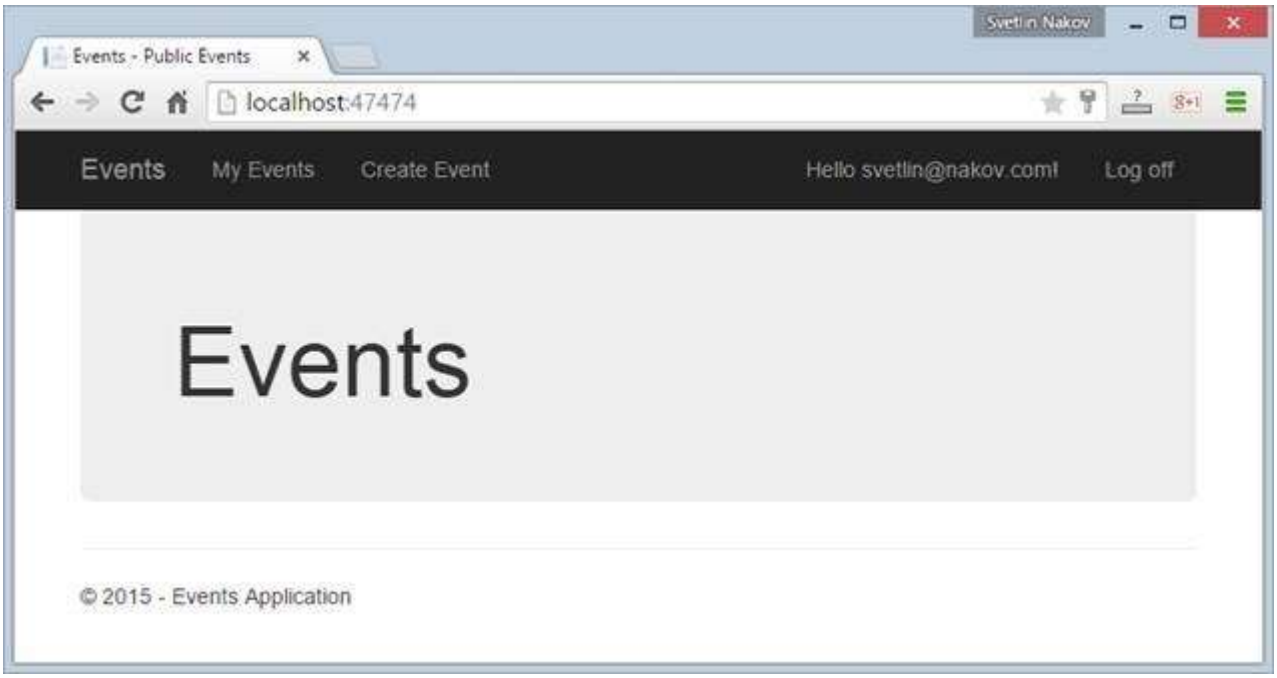
8. Fix the **missing** "**`using …`**" in the code. The project **`Events.Data`** should compile without errors.
9. The project **`Events.Web`** will not compile due to missing classes: **`ApplicationUser`** and **`ApplicationDbContext`**. Let's fix this.
10. **Add project reference** from **`Events.Web`** project to **`Events.Data`** project:



11. Fix the **missing** "`using Events.Data;`" statements in the **`Events.Web`** project.

Now the project should compile correctly. If not, check your library and project references and "**using**" statements.

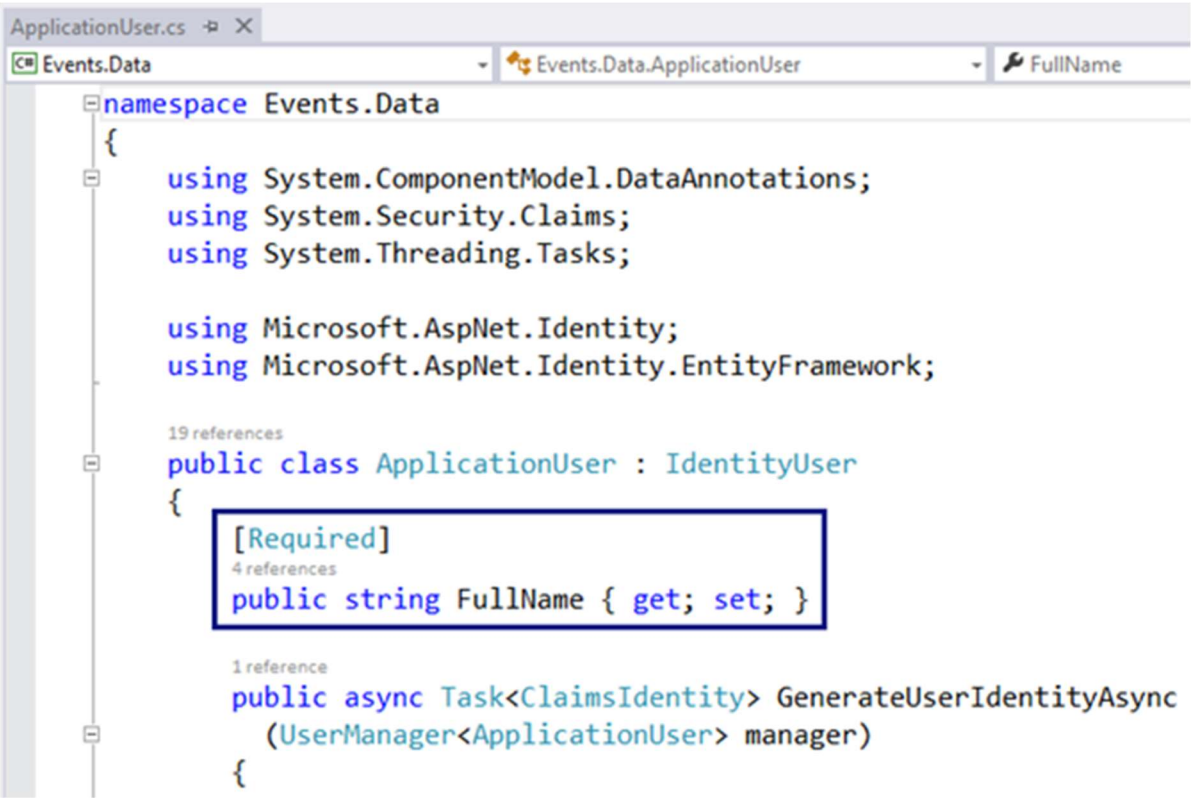
12. **Test the MVC application**. It should work correctly. Test the Home Page, Register, Login and Logout:



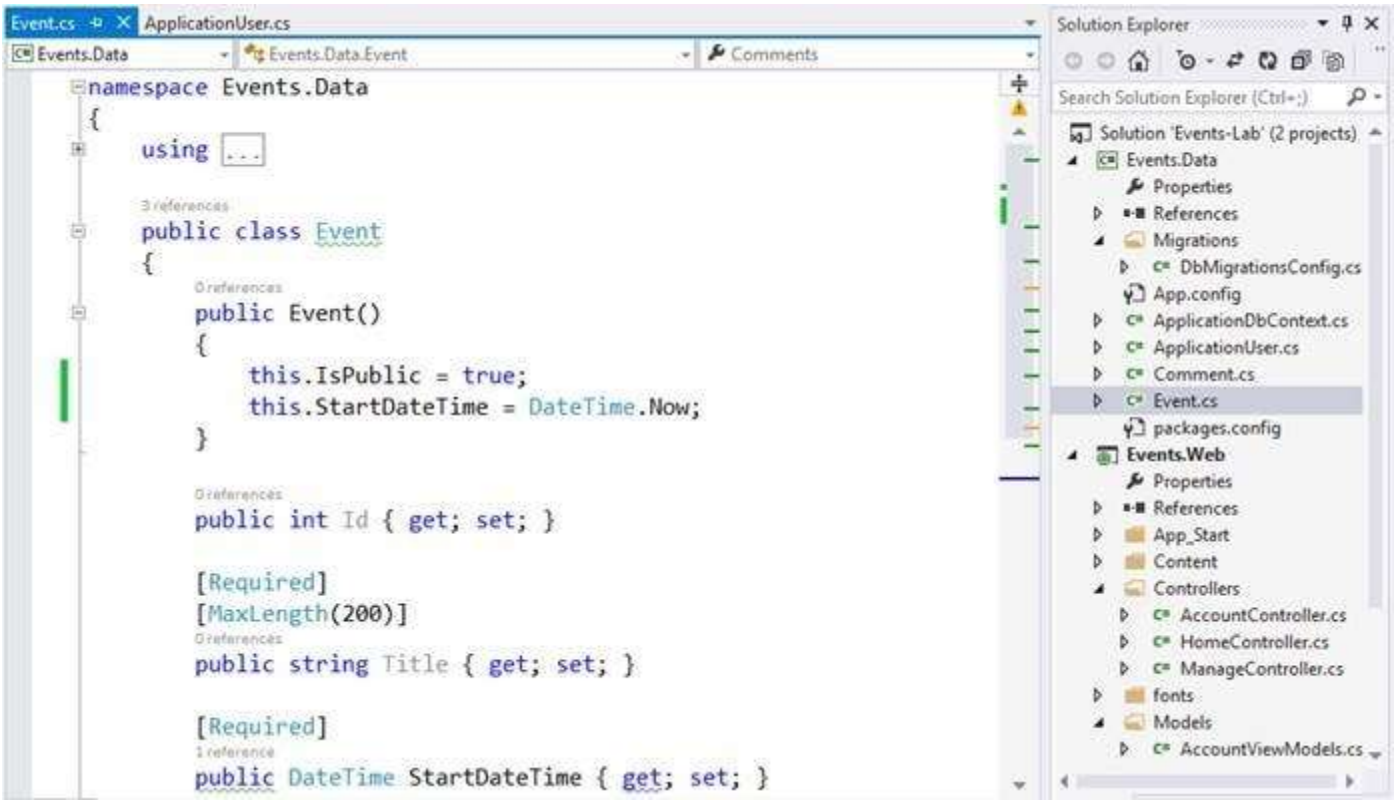
Step 5. Define the Data Model

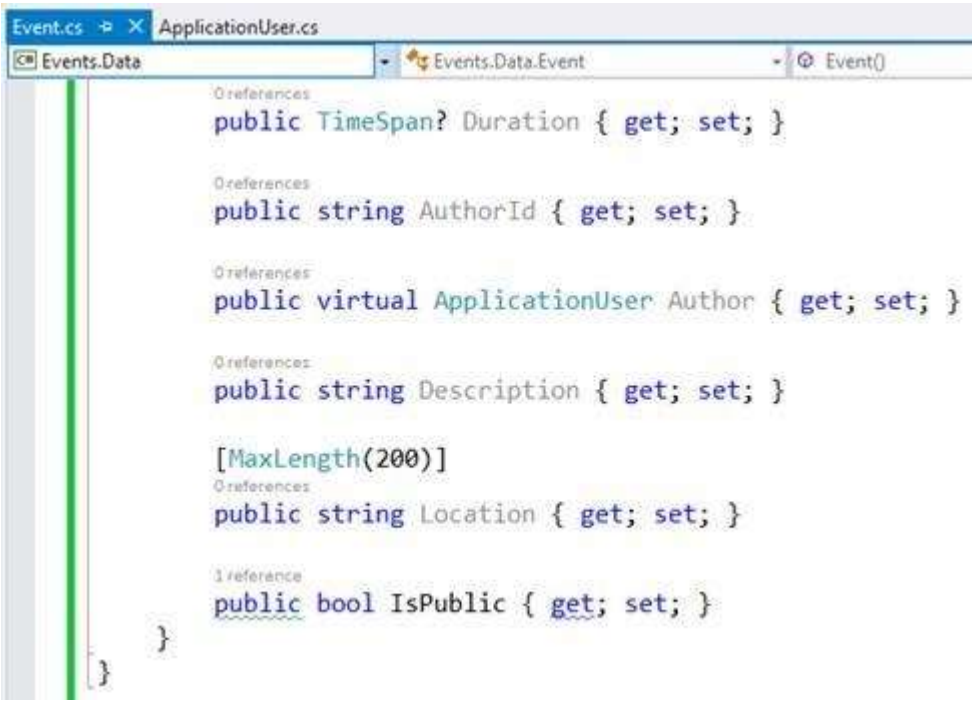
Now you are ready to **define the data model** for the Events management system: **entity classes** + Entity Framework **database context** + data **migration classes**. Let's define the entity classes:

1. First, add a **full name** in the **`ApplicationUser`** class. It will hold the name of the event's author:



2. Next, create your **`Event`** class. It will hold the events and their properties:





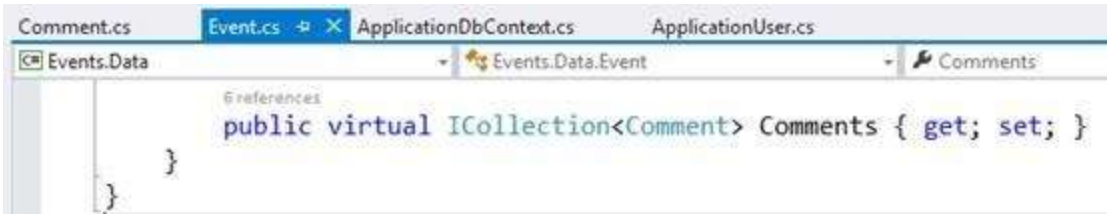
Compile your code. It should compile without errors.

3. Create your **`Comment`** class. It will hold the comments for each event:

```
public class Comment
{
    0 references
    public Comment()
    {
        this.Date = DateTime.Now;
    }

    0 references
    public int Id { get; set; }

    [Required]
    0 references
    public string Text { get; set; }

    [Required]
    1 reference
    public DateTime Date { get; set; }

    0 references
    public string AuthorId { get; set; }

    0 references
    public virtual ApplicationUser Author { get; set; }

    0 references
    public int EventId { get; set; }

    [Required]
    0 references
    public virtual Event Event { get; set; }
}
```

Compile your code. It should compile without errors.

4. **Add comments** to the **Event** class:



```
Event.cs | Comment.cs | ApplicationUser.cs
Events.Data | Events.Data.Event | StartDateTime

namespace Events.Data
{
    using ...

    3 references
    public class Event
    {
        0 references
        public Event()
        {
            this.IsPublic = true;
            this.StartDateTime = DateTime.Now;
            this.Comments = new HashSet<Comment>();
        }
```

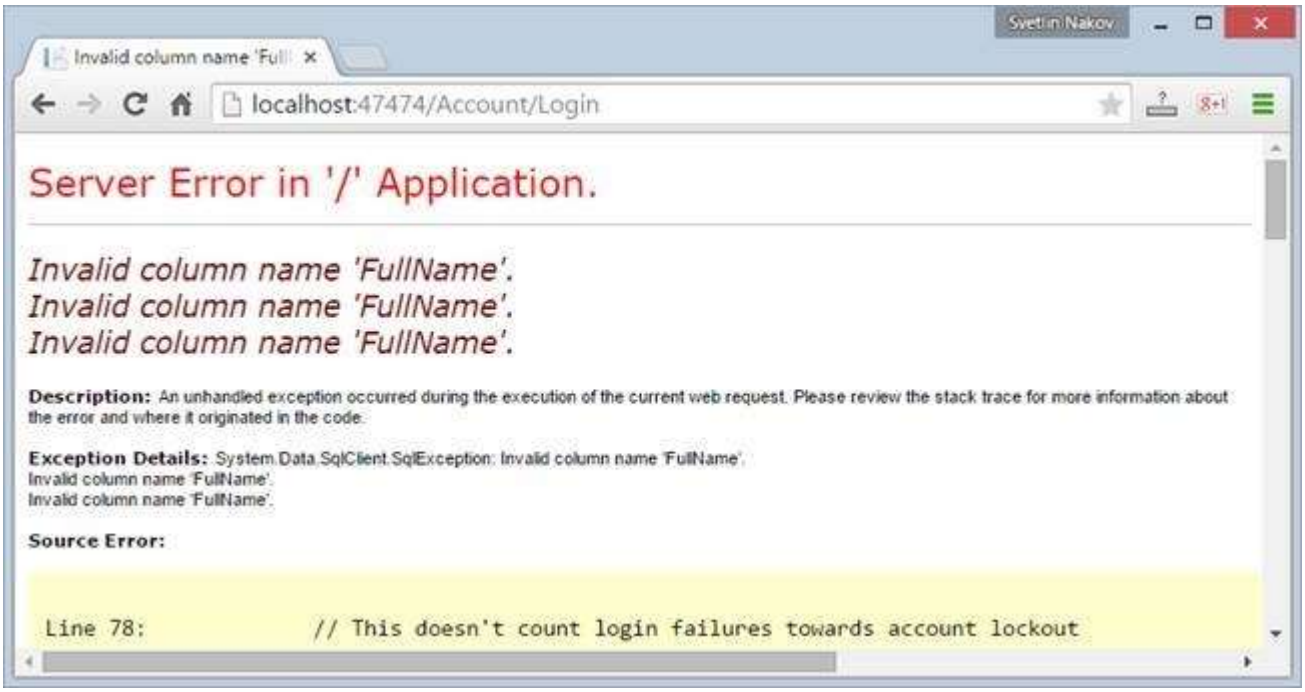
Compile again your code. It should compile without errors.

5. Modify the **`ApplicationDbContext`** class to include the events and comments as **`IDbSet<T>`**:

```
ApplicationDbContext.cs* | ApplicationUser.cs
Events.Data | Events.Data.ApplicationDbContext

namespace Events.Data
{
    using System.Data.Entity;
    using Microsoft.AspNet.Identity.EntityFramework;

    5 references
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        0 references
        public IDbSet<Event> Events { get; set; }

        0 references
        public IDbSet<Comment> Comments { get; set; }
```

Compile again your code. It should compile without errors.

6. **Run** the MVC application. **It will fail at the Login action**. Think why!

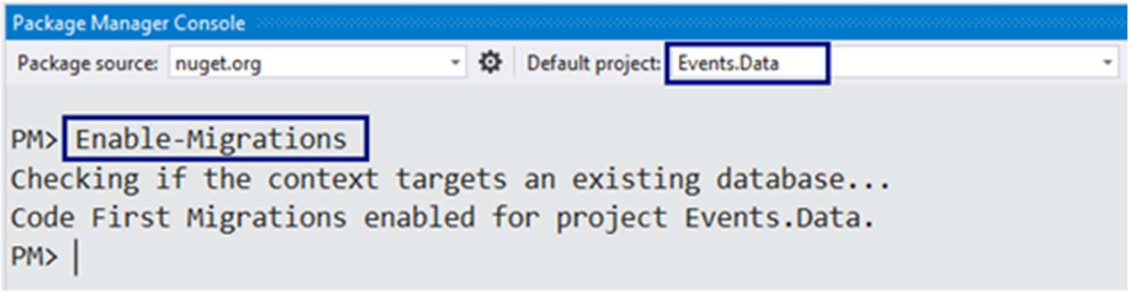


You need a **migration strategy** for the database, right? How shall changes in the DB schema be handled?

Step 6. Configure Database Migrations

Now you will **configure automatic migration strategy** for the database to simplify changes in the database schema during the project development.

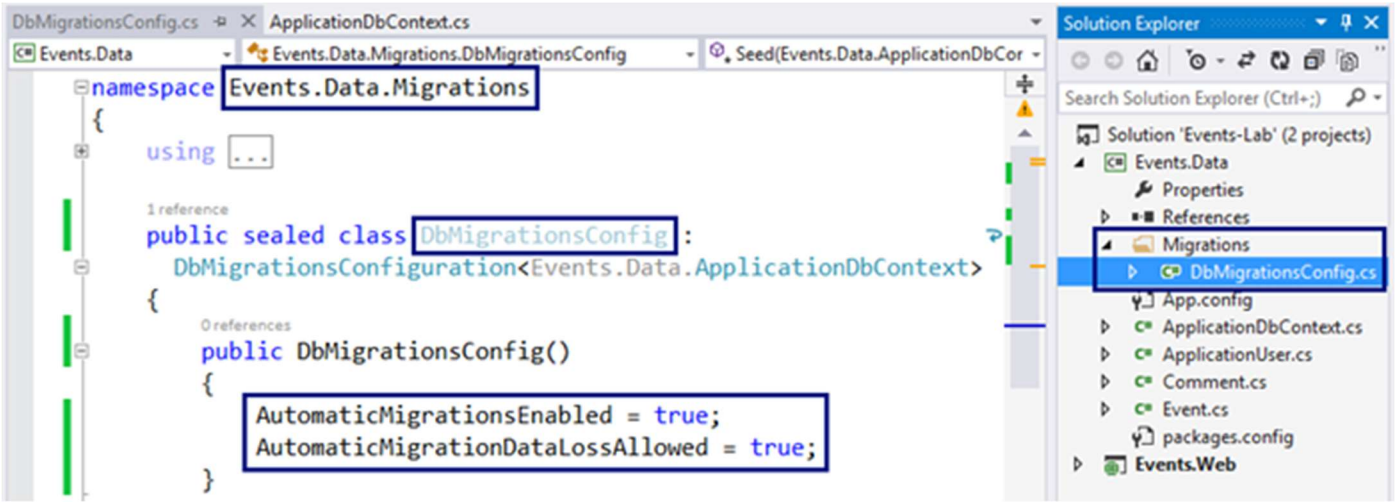
1. First, **enable the database migrations** from the Package Manager console for the **`Events.Data`** project:



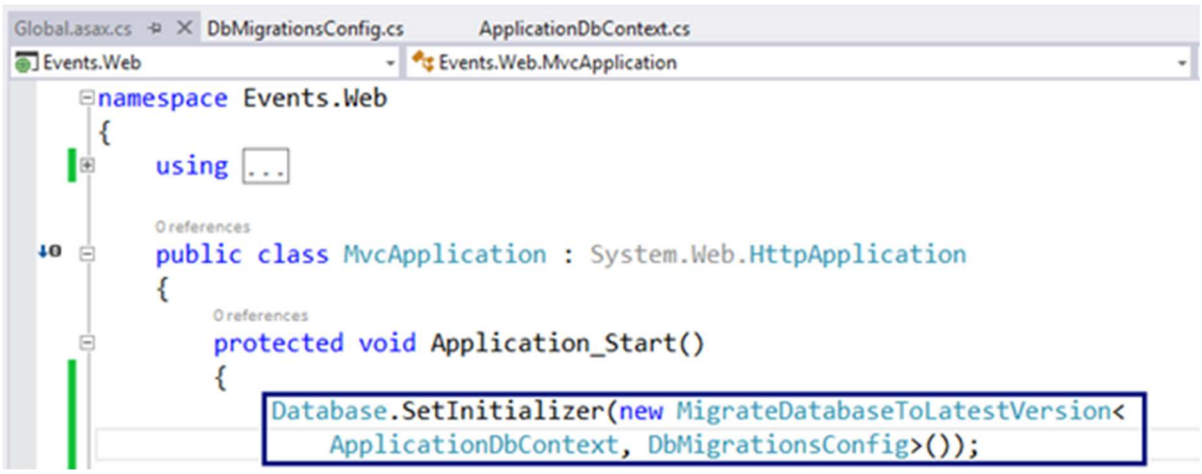
The command "**`Enable-Migrations`**" will generate a file **`Migrations\Configuration.cs`** in the **`Events.Data`** project:

2. The generated DB migration class has **non-informative name**. It is a good practice to **rename it**: **`Configuration` → `DbMigrationsConfig`**. Rename the file that holds this class as well: **`Configuration.cs` →`DbMigrationsConfig.cs`**.

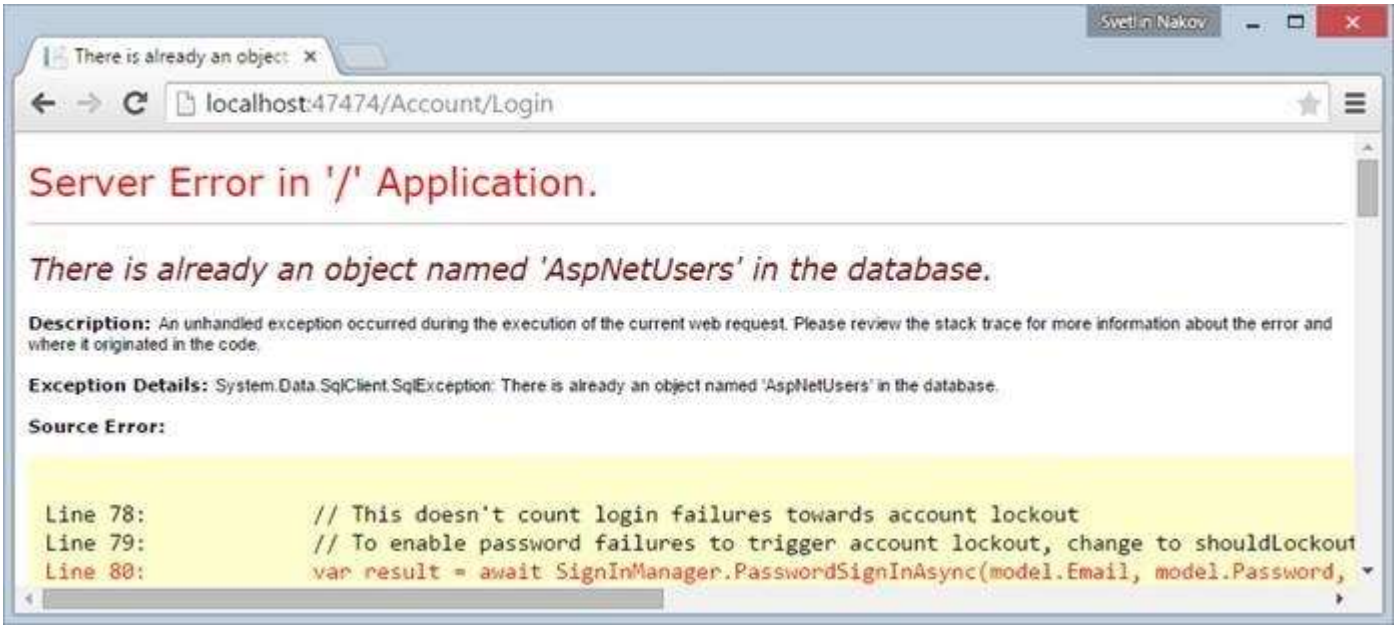
3. **Edit** the DB migration configuration: make the class **`public`** (it will be used by the MVC project); enable **automatic migrations**; enable **data loss migrations**:



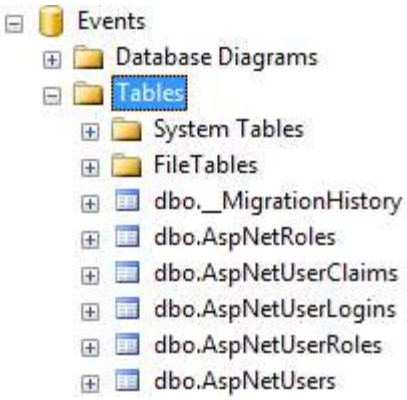
4. Set the database migration configuration in the **database initializer** in your **`Global.asax`** file in your MVC Web application:



5. Now **run the MVC application** to **test** the new database migration strategy. Rebuild the application and run it. The application will fail!

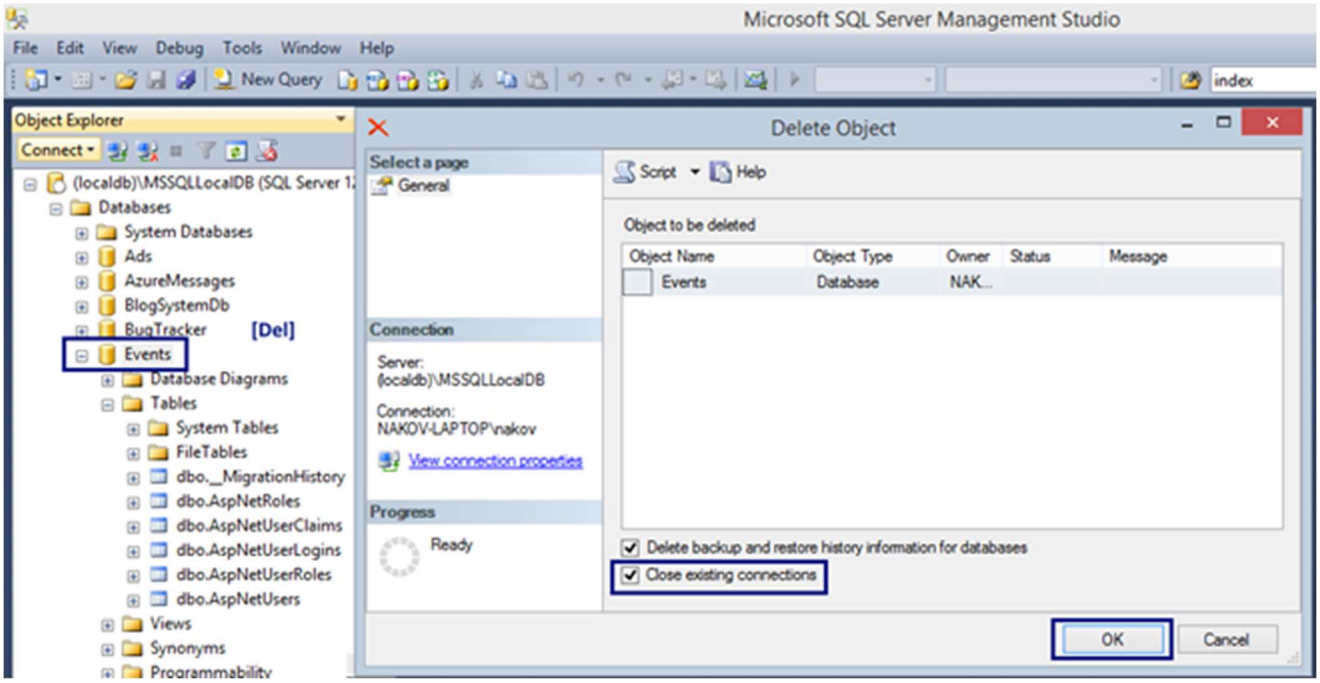


Seems like the **database migration has not been executed**, right? You could also check the database:

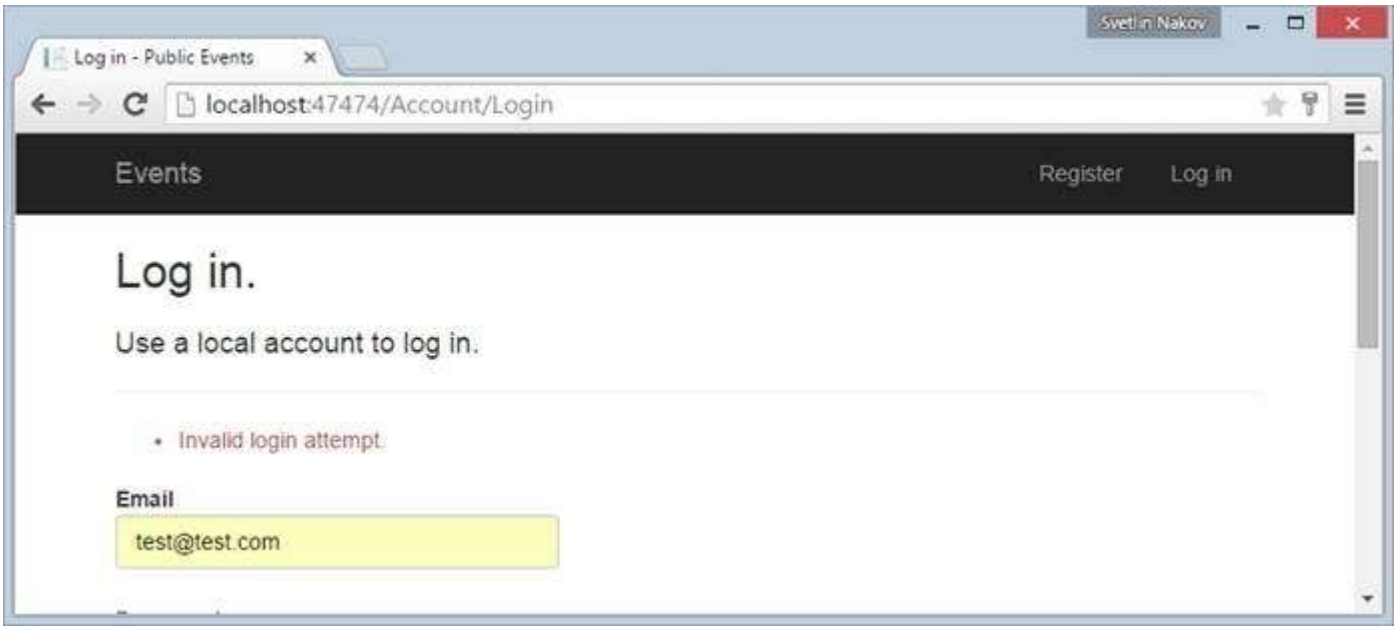


The new tables (**`Events`** and **`Comments`**) are missing.

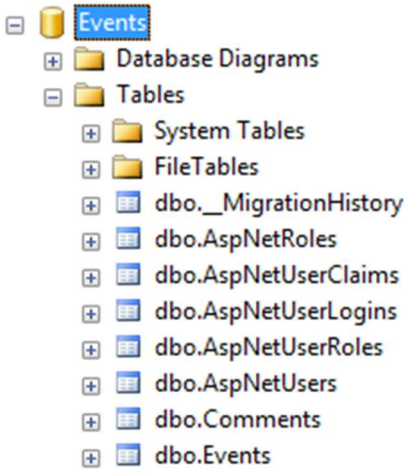
6. Let's fix this problem. The easiest way is just to **drop the database**:



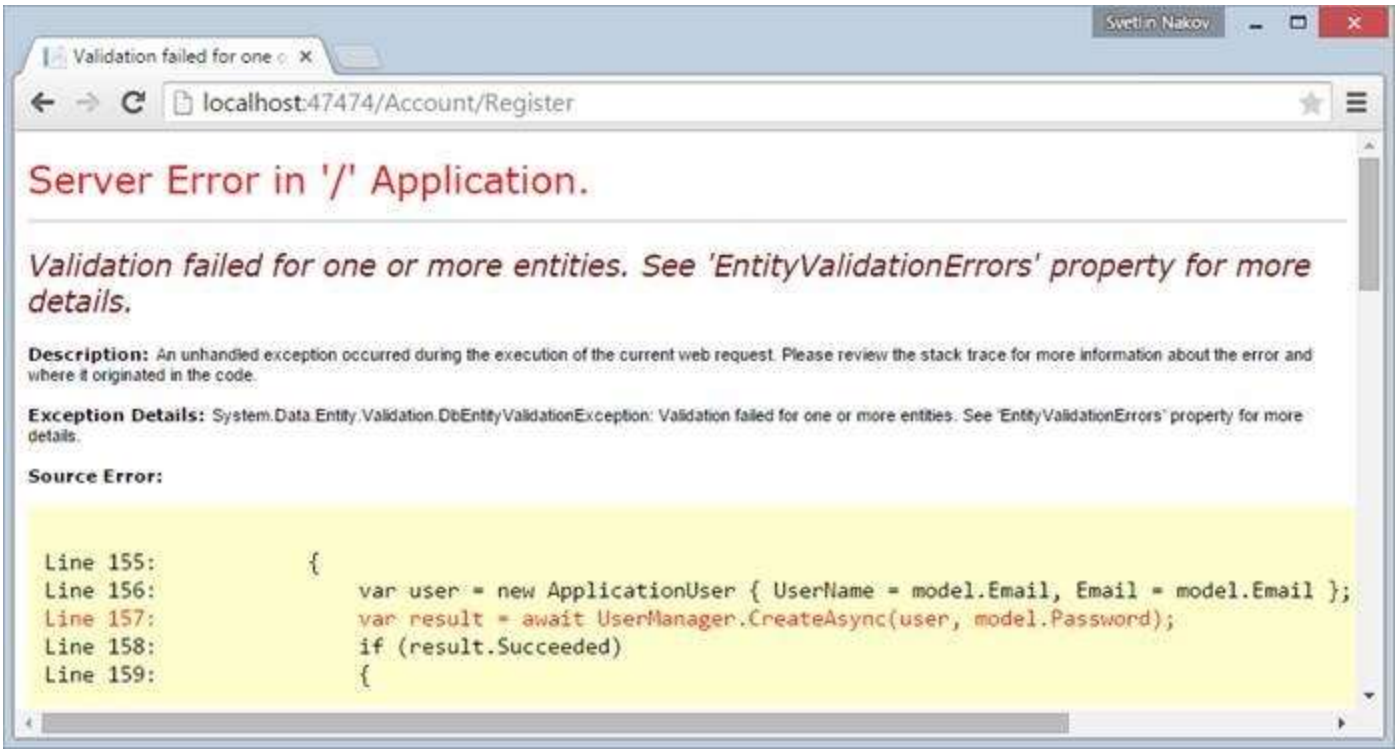
7. Let's **run the MVC Web application** again and try to login. It will say "**Invalid login**":



This is quite normal: we **dropped the entire database**, right? Let's see what the database holds now:

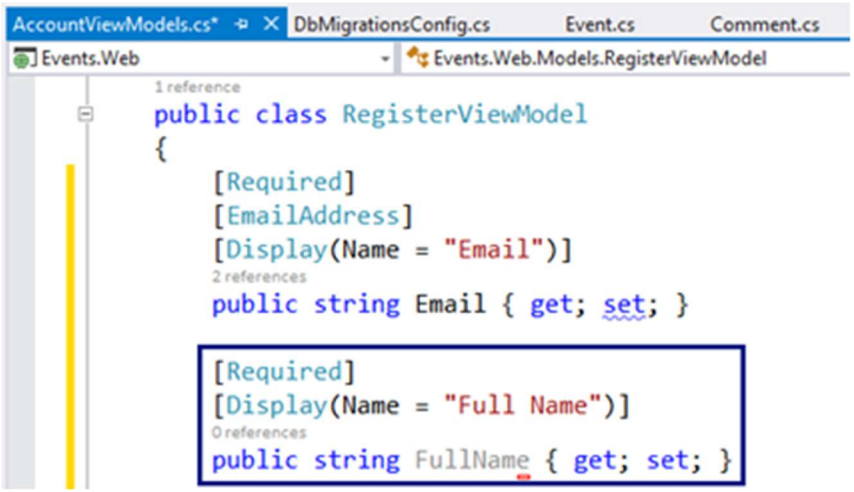


8. Let's **register** a new account. The application will fail with "**Entity Validation Errors**":



The problem comes from the field "**FullName**" in the **ApplicationUser** class. It is required but is left empty.

9. We need to change the **RegisterViewModel**, the **Register.cshtml** view and the **Register** action in the **AccountController** to add the **FullName** property:



Register.cshtml* | AccountViewModels.cs | DbMigrationsConfig.cs | Event.cs | Comment.cs | ApplicationUser.cs

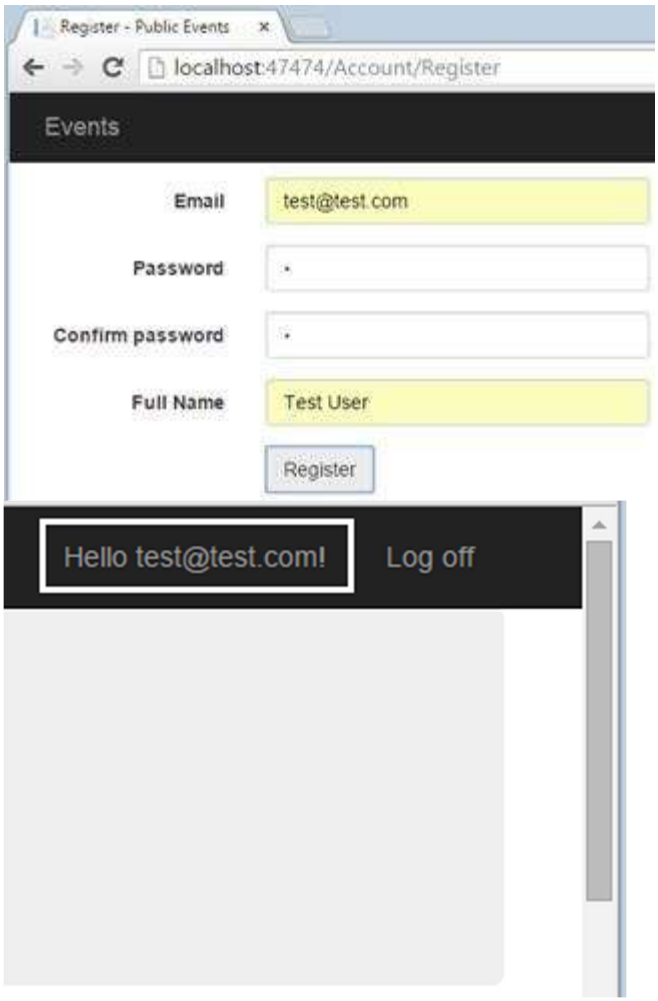
```
<div class="form-group">
    @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.FullName, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.FullName, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" class="btn btn-default" value="Register" />
    </div>
</div>
}
```

AccountController.cs* | Register.cshtml | AccountViewModels.cs | DbMigrationsConfig.cs | Event.cs

Events.Web | Events.Web.Controllers.AccountController

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = model.Email,
            Email = model.Email,
            FullName = model.FullName
        };
```

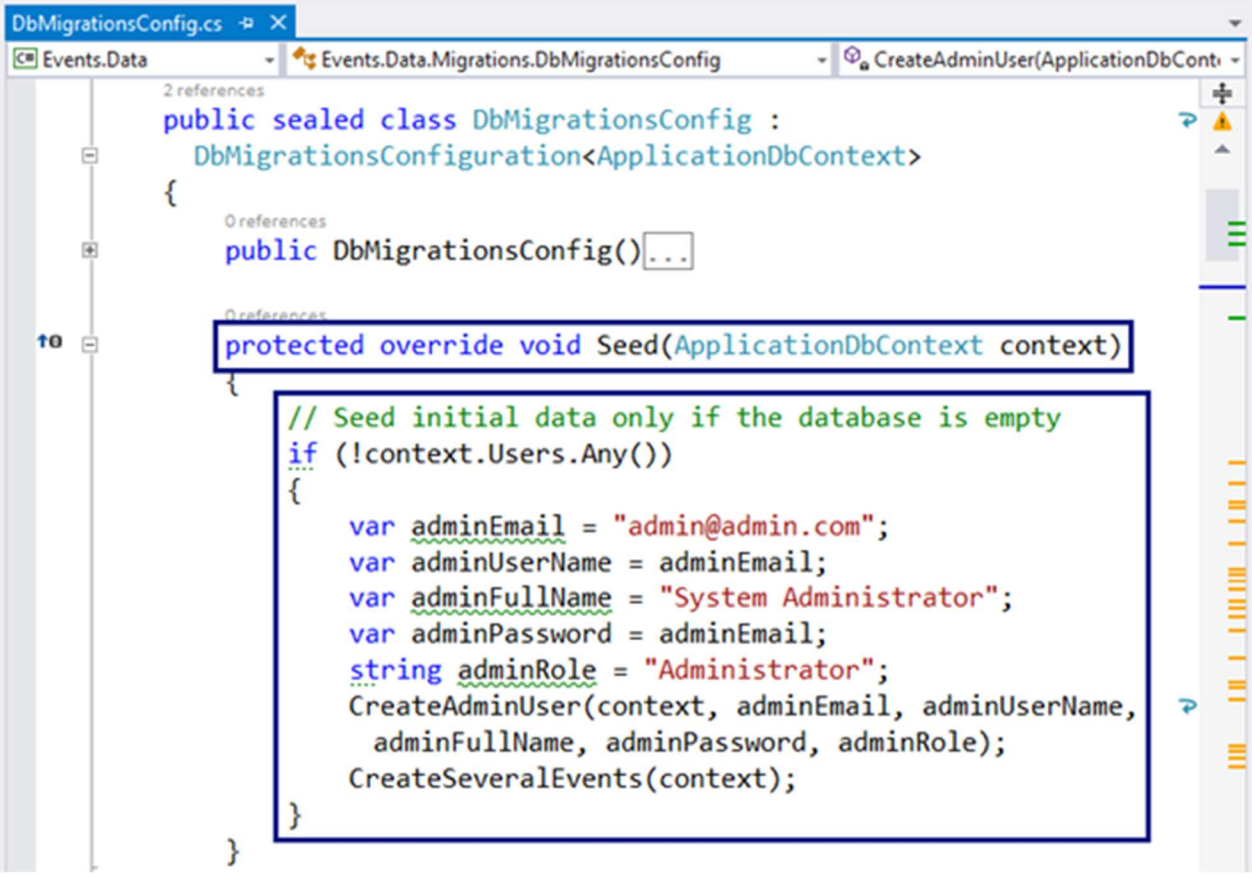
10. Now **run** the application and try to **register a new user account**. It should work correctly:



Step 7. Fill Sample Data in the DB (Seed the Database)

You are ready to **fill the database with sample data**. Sample data will help during the development to simplify testing. Now let's fill some data: create **admin account** and a **few events** with **few comments**. The easiest way to do this is to create a `Seed()` method in the database migration strategy.

1. In the `Seed()` method check whether the database is empty. If it is empty, **fill sample data**. The `Seed()` method will run every time when the application starts, after a database schema change. You can use sample code like this:



In this example, the default administrator user will be "**admin@admin.com**" with the same password. This will simplify testing, because some actions require administrator privileges in the system.

2. Now, let's create the **admin user** and the **administrator role** and **assign administrator role** to the admin user:

```
        // Create the "admin" user
        var adminUser = new ApplicationUser
        {
            UserName = adminUserName,
            FullName = adminFullName,
            Email = adminEmail
        };
        var userStore = new UserStore<ApplicationUser>(context);
        var userManager = new UserManager<ApplicationUser>(userStore);
        userManager.PasswordValidator = new PasswordValidator
        {
            RequiredLength = 1,
            RequireNonLetterOrDigit = false,
            RequireDigit = false,
            RequireLowercase = false,
            RequireUppercase = false,
        };
        var userCreateResult = userManager.Create(adminUser, adminPassword);
        if (!userCreateResult.Succeeded)
        {
            throw new Exception(string.Join("; ", userCreateResult.Errors));
        }
// Create the "Administrator" role
var roleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));
var roleCreateResult = roleManager.Create(new IdentityRole(adminRole));
if (!roleCreateResult.Succeeded)
{
    throw new Exception(string.Join("; ", roleCreateResult.Errors));
}

// Add the "admin" user to "Administrator" role
var addAdminRoleResult = userManager.AddToRole(adminUser.Id, adminRole);
if (!addAdminRoleResult.Succeeded)
{
    throw new Exception(string.Join("; ", addAdminRoleResult.Errors));
}
```

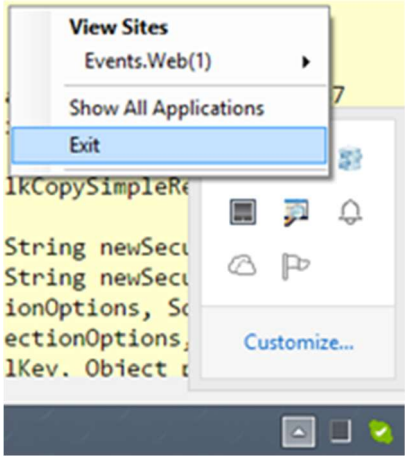
3. Next, **create some events** with comments:

```
private void CreateSeveralEvents(ApplicationDbContext context)
{
    context.Events.Add(new Event()
    {
        Title = "Party @ SoftUni",
        StartDateTime = DateTime.Now.Date.AddDays(5).AddHours(21).AddMinutes(30),
        Author = context.Users.First(),
    });

    context.Events.Add(new Event()
    {
        Title = "Passed Event <Anonymous>",
        StartDateTime = DateTime.Now.Date.AddDays(-2).AddHours(10).AddMinutes(30),
        Duration = TimeSpan.FromHours(1.5),
        Comments = new HashSet<Comment>() {
            new Comment() { Text = "<Anonymous> comment" },
            new Comment() { Text = "User comment", Author = context.Users.First() }
        }
    });
```

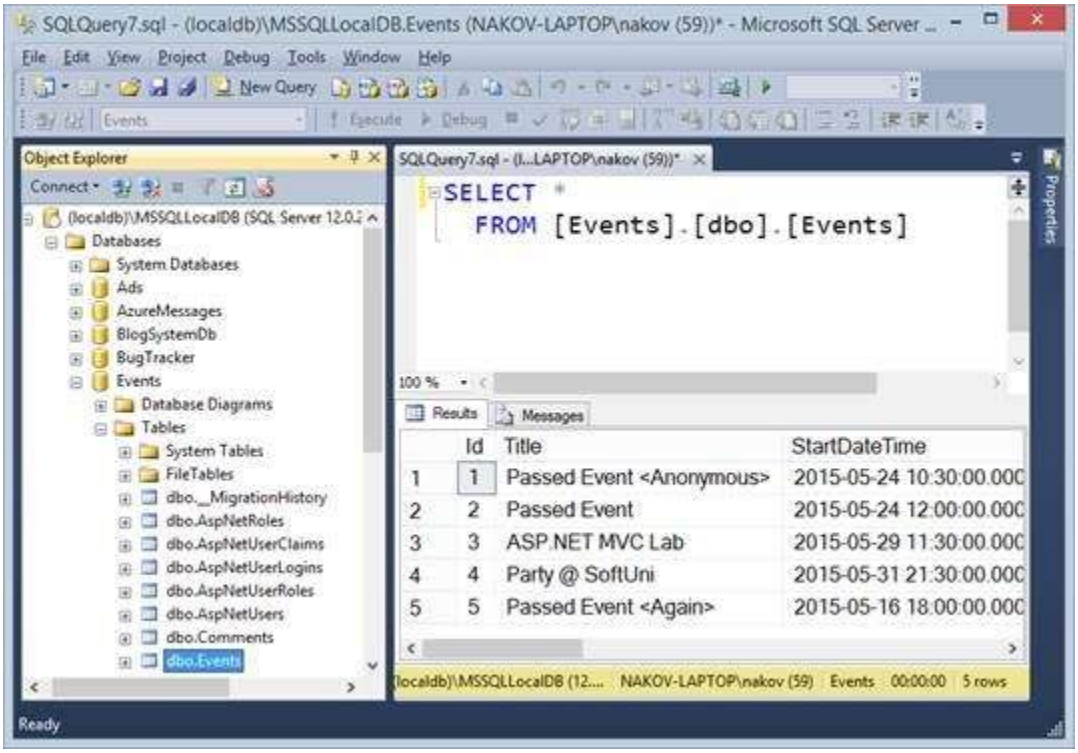
4. Add some code to **create a few more events**:
    - Few upcoming events
    - Few passed events
    - Few anonymous events (no author)
    - Events with / without comments
5. **Drop the database** and **restart the Web application** to re-create the DB.

If you get "**Cannot open database**" error, **stop the IIS Server**, **rebuild** and run the MVC application again:

Stopping the IIS Express web server (Internet Information Services):



6. **Check the data** in the **Events** database. It should hold few records in the `Events`, `Comments`, `AspNetUsers`, `AspNetRoles` and `AspNetUserRoles`:



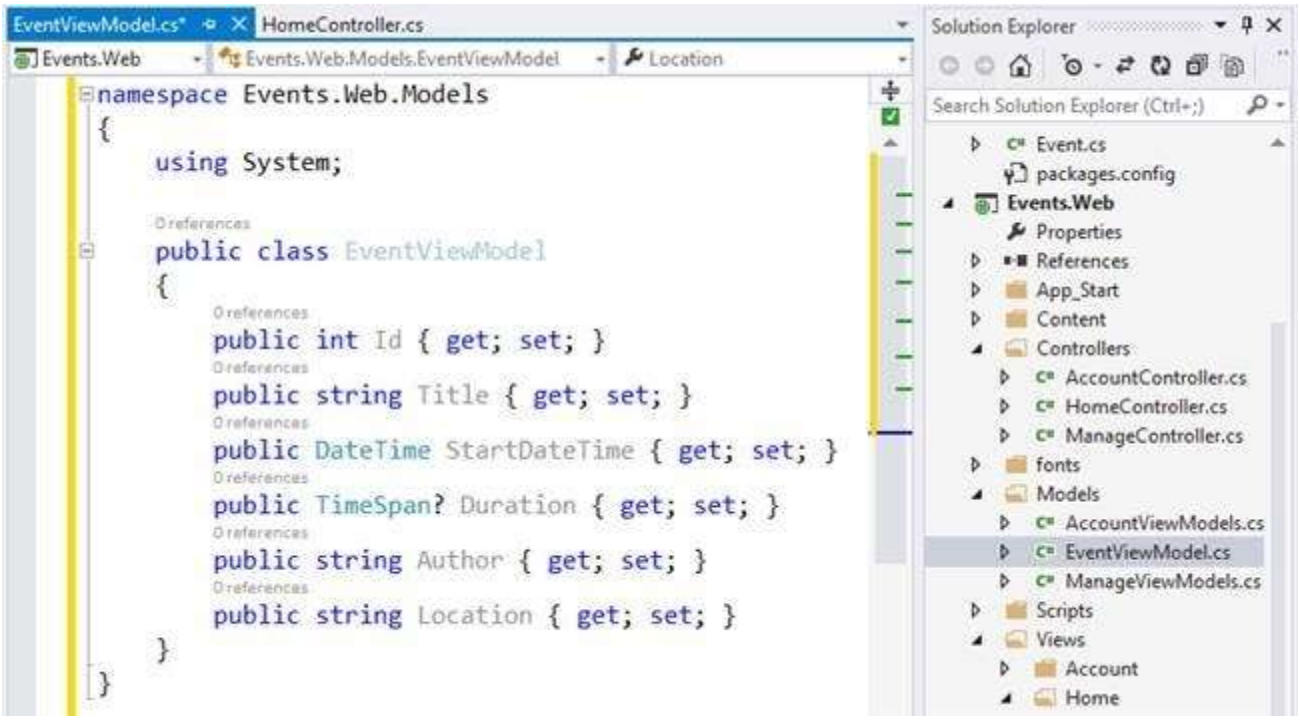
Now you have sample data in the `Events` database and you are ready to continue working on the Events management MVC Web application.

Step 8. List All Public Events at the Home Page

Listing all public events at the home page passes through several steps:

- Create `EventViewModel` to hold event data for displaying on the home page.
- Create `UpcomingPassedEventsViewModel` to hold lists of upcoming and passed events.
- Write the logic to **load the upcoming and passed events** in the default action of the `HomeController`.
- Write a **view** to display the events at the home page.

1. Create a class `Models\EventViewModel.cs` to hold event data for displaying on the home page:



2. Create a class **`Models\UpcomingPassedEventsViewModel.cs`** to hold lists of upcoming and passed events:

```
namespace Events.Web.Models
{
    using System.Collections.Generic;

    0 references
    public class UpcomingPassedEventsViewModel
    {
        0 references
        public IEnumerable<EventViewModel> UpcomingEvents { get; set; }

        0 references
        public IEnumerable<EventViewModel> PassedEvents { get; set; }
    }
}
```

3. Create a class **`Controllers\BaseController.cs`** to hold the Entity Framework data context. It will be used as base (parent) class for all controllers in the MVC application:

```
BaseController.cs | HomeController.cs
Events.Web | Events.Web.Controllers.BaseController

namespace Events.Web.Controllers
{
    using System.Web.Mvc;
    using Events.Data;

    0 references
    public class BaseController : Controller
    {
        protected ApplicationDbContext db = new ApplicationDbContext();
    }
}
```

4. First extend the **`BaseController`** from the **`HomeController`** to inherit the DB context:

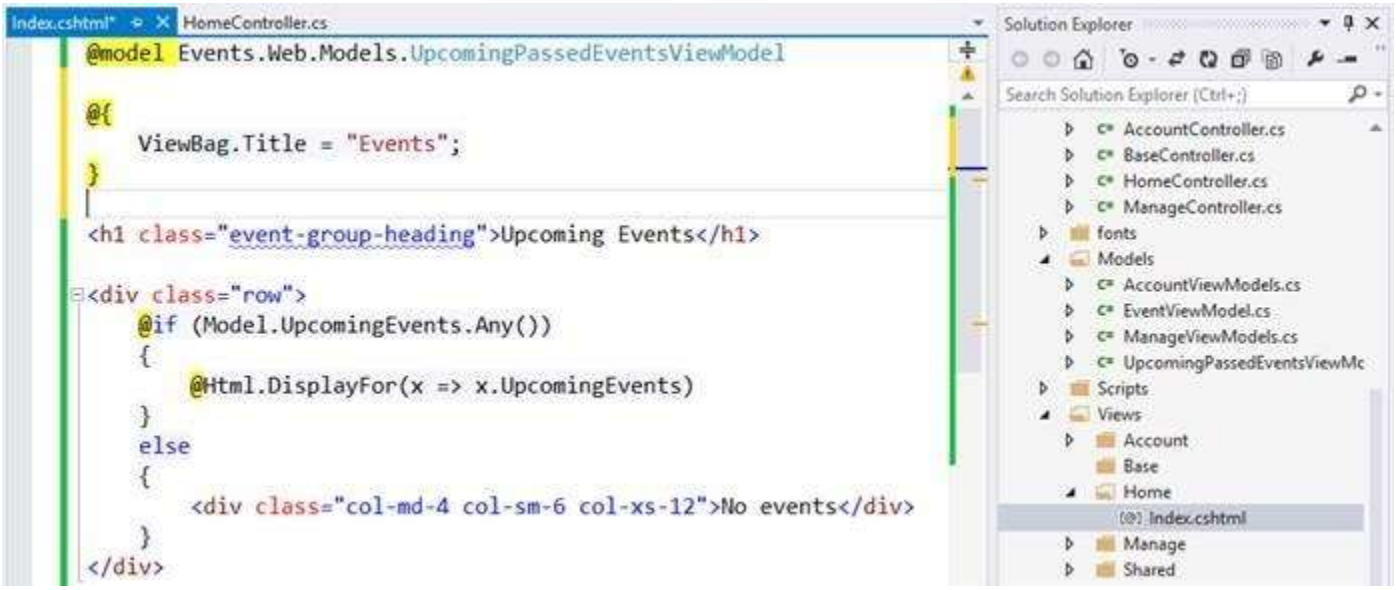
```
BaseController.cs | HomeController.cs*
Events.Web | Events.Web.Controllers.HomeController

namespace Events.Web.Controllers
{
    0 references
    public class HomeController : BaseController
    {
        0 references
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

5. Now write the code in the default action **`Index()`** of the **`HomeController`** to load all public events:

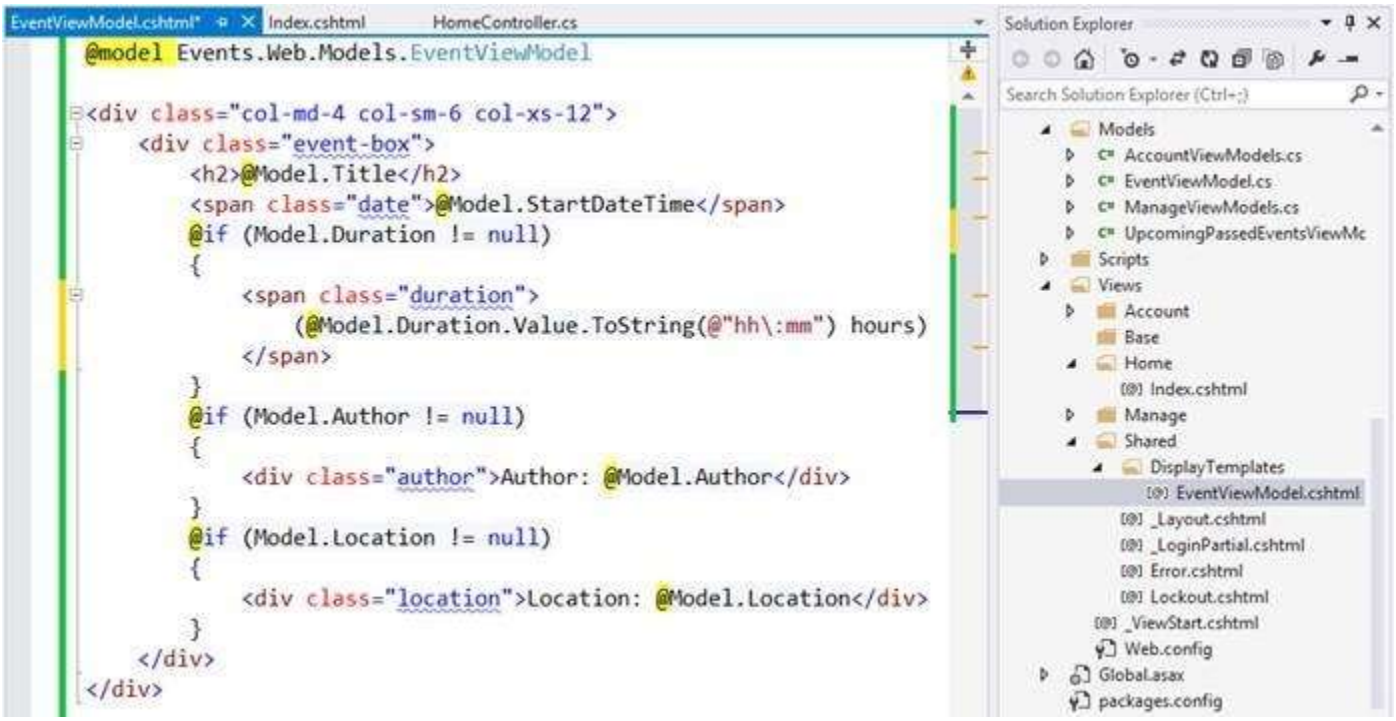
```
public class HomeController : BaseController
{
    0 references
    public ActionResult Index()
    {
        var events = this.db.Events
            .OrderBy(e => e.StartDateTime)
            .Where(e => e.IsPublic)
            .Select(e => new EventViewModel()
            {
                Id = e.Id,
                Title = e.Title,
                StartDateTime = e.StartDateTime,
                Duration = e.Duration,
                Author = e.Author.FullName,
                Location = e.Location
            });

        var upcomingEvents = events.Where(e => e.StartDateTime > DateTime.Now);
        var passedEvents = events.Where(e => e.StartDateTime <= DateTime.Now);
        return View(new UpcomingPassedEventsViewModel()
        {
            UpcomingEvents = upcomingEvents,
            PassedEvents = passedEvents
        });
    }
}
```

The idea of the above code is to select all public events, ordered by start date, then transform them from entity class **`Event`** to view model class **`EventViewModel`**and split them into two collections: upcoming and passed events.

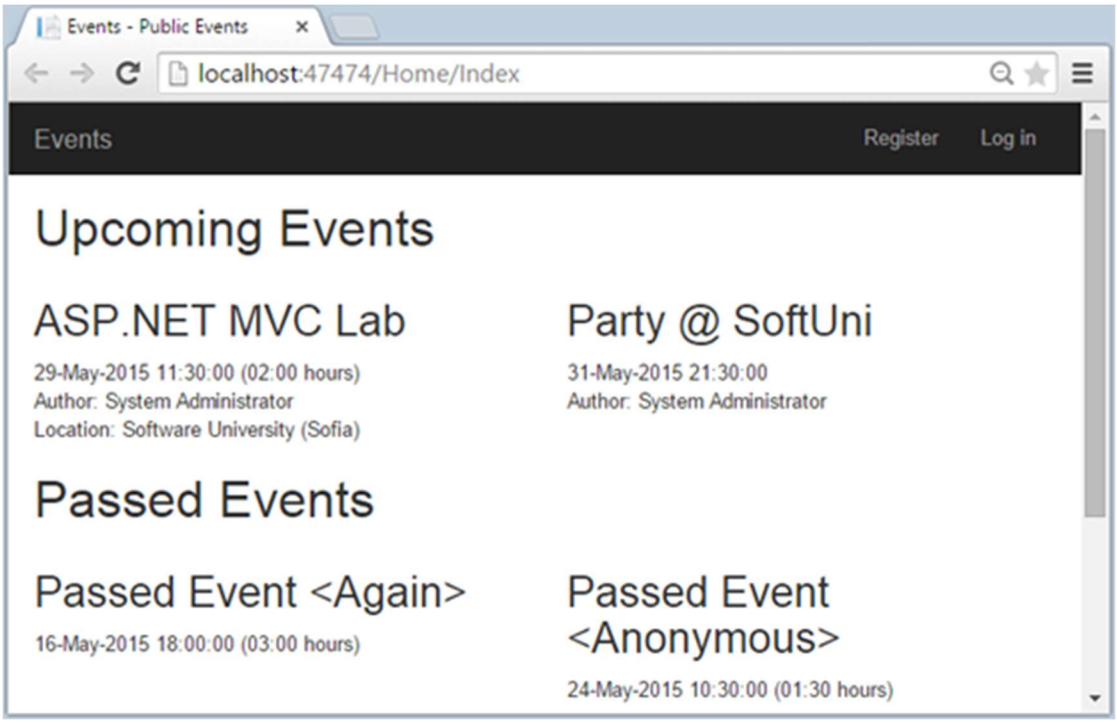
6. Finally, create the **view** to display the events at the home page – **`\Views\Home\Index.cshtml`**:



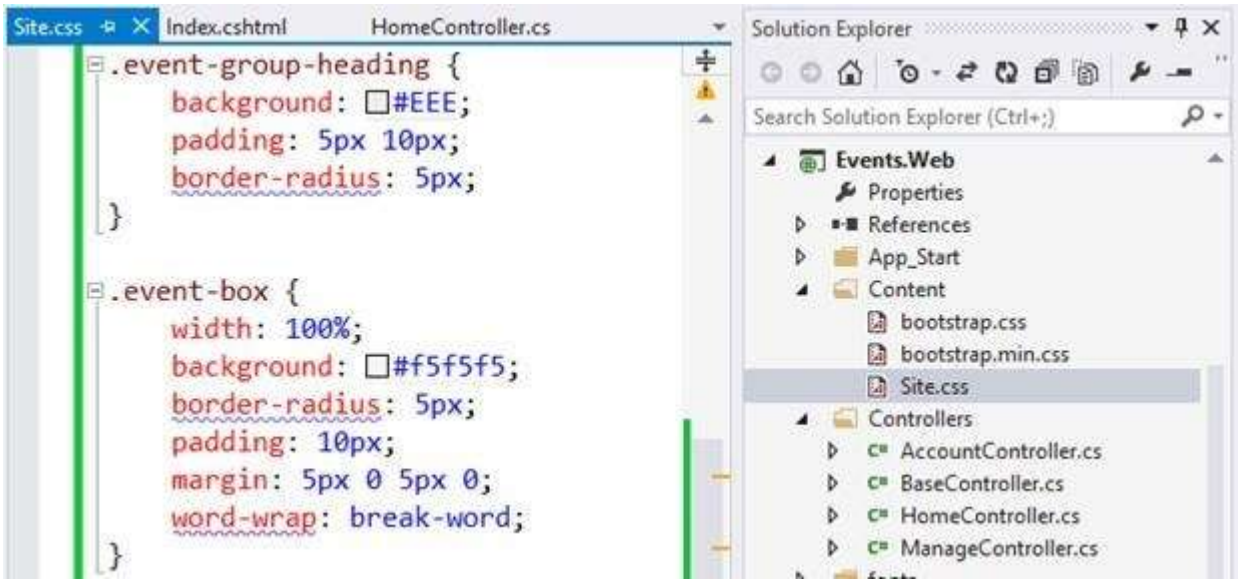
7. The above code shows the upcoming events only. Add similar code to **list the passed events** as well.
8. You need to define a **display template** for the `EventViewModel` class. It will be used when the Razor engine renders this code: `@Html.DisplayFor(x =>x.UpcomingEvents)`. Create the display template in the file `\Views\Shared\DisplayTemplates\EventViewModel.cshtml`:



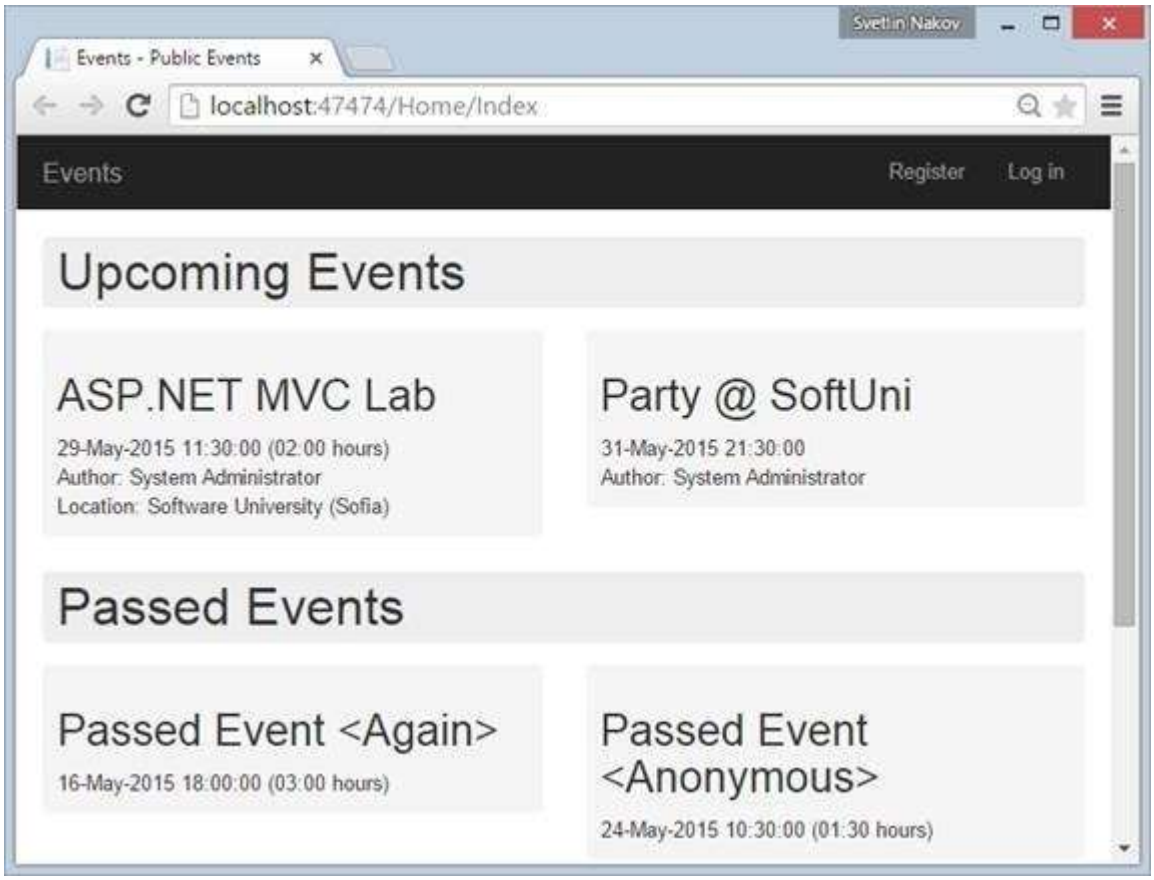
9. **Build and run the project** to test whether it works correctly. If you have a luck, the code will be correct at your first attempt. The expected result may look like this:



10. Now let's **add some CSS styles** to make the UI look better. Append the following CSS style definitions at the end of the file **`\Content\Site.css`**:



11. **Refresh** the application home page to see the styled events:



12. Good job! Events are displayed at the home page. Now let's **optimize a bit** the **`HomeController`**. It holds a code that loads an **`Event`** object into a new**`EventViewModel`** object:

```
.Select(e => new EventViewModel()
{
    Id = e.Id,
    Title = e.Title,
    StartDateTime = e.StartDateTime,
    Duration = e.Duration,
    Author = e.Author.FullName,
    Location = e.Location
});
```

This code can be moved to the **`EventViewModel`** class to make the **`HomeController`** cleaner:

```
EventViewModel.cs*   Index.cshtml   HomeController.cs
Events.Web                          Events.Web.Models.EventViewModel
0 references
public static Expression<Func<Event, EventViewModel>> ViewModel
{
    get
    {
        return e => new EventViewModel()
        {
            Id = e.Id,
            Title = e.Title,
            StartDateTime = e.StartDateTime,
            Duration = e.Duration,
            Location = e.Location,
            Author = e.Author.FullName
        };
    }
}
```

Looks complex, but it is not really. This static property is used in **`.Select(…)`** queries to transform **`Event`** into **`EventViewModel`**. Such transformations are boring and work better encapsulated in the view model class. Now the **`HomeController`** can use this static property as follows:

```
public ActionResult Index()
{
    var events = this.db.Events
        .OrderBy(e => e.StartDateTime)
        .Where(e => e.IsPublic)
        .Select(EventViewModel.ViewModel);
```

13. **Build and test** the project again to ensure it woks correctly after the refactoring.

Step 9. List Events Details with AJAX

The next step in the development of the Events management application is to **display event details dynamically with AJAX**. The goal is to achieve the following functionality:

1. First, let's define the **`EventDetailsViewModel`** class that will hold the event details:

```
public class EventDetailsViewModel
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Description { get; set; }
    1 reference
    public string AuthorId { get; set; }
    1 reference
    public IEnumerable<CommentViewModel> Comments { get; set; }
    1 reference
    public static Expression<Func<Event, EventDetailsViewModel>> ViewModel
    {
        get
        {
            return e => new EventDetailsViewModel()
            {
                Id = e.Id,
                Description = e.Description,
                Comments = e.Comments.AsQueryable().Select(CommentViewModel.ViewModel),
                AuthorId = e.Author.Id
            };
        }
    }
}
```

It will internally refer the **`CommentViewModel`** class:

```
public class CommentViewModel
{
    0 references
    public string Text { get; set; }
    1 reference
    public string Author { get; set; }
    1 reference
    public static Expression<Func<Comment, CommentViewModel>> ViewModel
    {
        get
        {
            return c => new CommentViewModel()
            {
                Text = c.Text,
                Author = c.Author.FullName
            };
        }
    }
}
```

2. Next, let's write the **AJAX controller action**, that will return the event details when users click the **[View Details]** button:

```
public ActionResult EventDetailsById(int id)
{
    var currentUserId = this.User.Identity.GetUserId();
    var isAdmin = this.IsAdmin();
    var eventDetails = this.db.Events
        .Where(e => e.Id == id)
        .Where(e => e.IsPublic || isAdmin || (e.AuthorId != null && e.AuthorId == currentUserId))
        .Select(EventDetailsViewModel.ViewModel)
        .FirstOrDefault();

    var isOwner = (eventDetails != null && eventDetails.AuthorId != null &&
        eventDetails.AuthorId == currentUserId);
    this.ViewBag.CanEdit = isOwner || isAdmin;

    return this.PartialView("_EventDetails", eventDetails);
}
```

The above logic is not quite straightforward, because event details should be shown only when the user has **permissions to access** them:

- The event author can view and edit its own events (even private).
- System administrators can view and edit all events (even private).
- Public events are visible by everyone, but not editable by everyone.

3. It is quite good idea to extract the check whether the current user is administrator in the **`BaseController`**:

```
public class BaseController : Controller
{
    protected ApplicationDbContext db = new ApplicationDbContext();

    1 reference
    public bool IsAdmin()
    {
        var currentUserId = this.User.Identity.GetUserId();
        var isAdmin = (currentUserId != null && this.User.IsInRole("Administrator"));
        return isAdmin;
    }
}
```

4. Next, let's create the **partial view** that will be returned when the AJAX request for event detail is made. The partial view for the above AJAX action should stay in **`\Views\Home\_EventDetails.cshtml`**.

First, display the event **description**:

```
@model Events.Web.Models.EventDetailsViewModel

@if (Model.Description != null)
{
    <div class="description">Description: @Model.Description</div>
}
```

Next, list the event **comments**:

```
@if (Model.Comments.Any())
{
    @:Comments:
    <ul>
        @foreach (var comment in Model.Comments)
        {
            <li>
                @comment.Text
                @if (@comment.Author != null)
                {
                    @: (by @comment.Author)
                }
            </li>
        }
    </ul>
}
else
{
    <p>No comments</p>
}
```

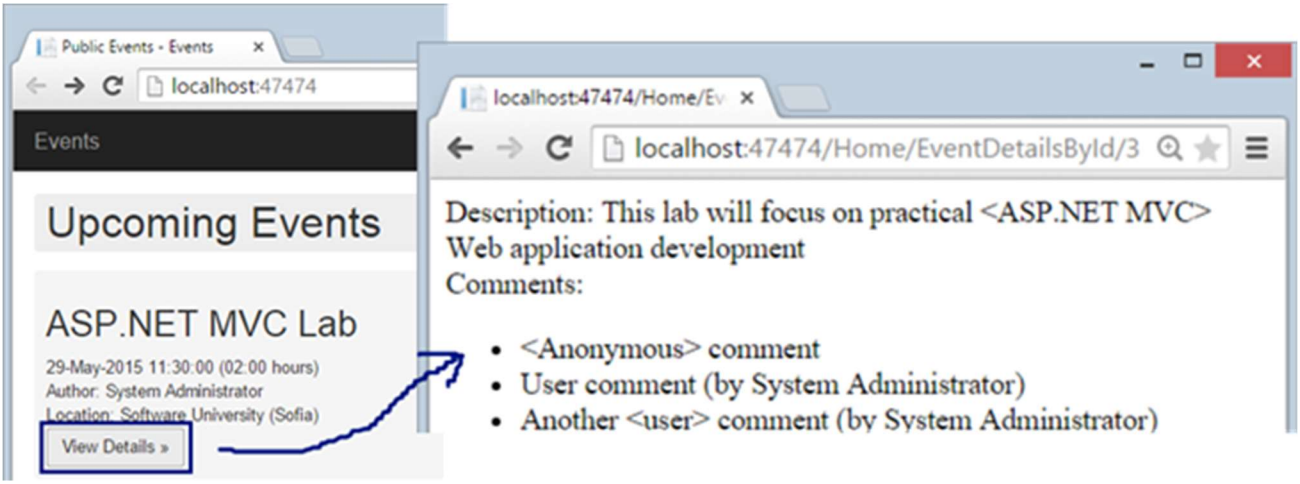
Finally, show **[Edit] and [Delete] buttons** when the user has edit permissions:

```
@if (ViewBag.CanEdit)
{
    @Html.ActionLink("Edit", "Edit", "Events", new { id = Model.Id },
        new { @class = "btn btn-default" })
    <span></span>
    @Html.ActionLink("Delete", "Delete", "Events", new { id = Model.Id },
        new { @class = "btn btn-default" })
}
```

5. Now, let's edit the events display template **`\Views\Shared\DisplayTemplates\EventViewModel.cshtml`** and add the AJAX call in it:

```
EventViewModel.cshtml*  Index.cshtml  HomeController.cs
@model Events.Web.Models.EventViewModel

<div class="col-md-4 col-sm-6 col-xs-12">
    <div class="event-box">
        ...
        <div id="event-details-@Model.Id">
            @Ajax.ActionLink("View Details »", "EventDetailsById", "Home",
                new { id = Model.Id },
                new AjaxOptions
                {
                    InsertionMode = InsertionMode.Replace,
                    UpdateTargetId = "event-details-" + Model.Id
                }, new { @class = "btn btn-default" })
        </div>
    </div>
</div>
```

6. Let's **test** the new functionality. Everything works fine, except that the AJAX action link is **executed as normal link without AJAX**:

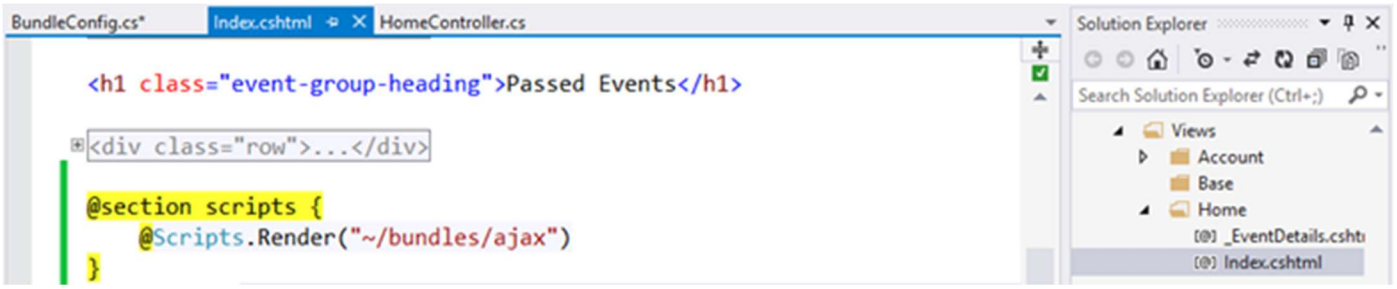


7. What is missing? The **script that handle the AJAX calls** are not loaded in the page. Let's add it.
   - First, install the NuGet package **`Microsoft.jQuery.Unobtrusive.Ajax`** in the project **`Events.Web`**:

```
Package Manager Console
Package source: nuget.org    Default project: Events.Web
PM> Install-Package Microsoft.jQuery.Unobtrusive.Ajax
Attempting to resolve dependency 'jQuery (≥ 1.8)'.
Installing 'Microsoft.jQuery.Unobtrusive.Ajax 3.2.3'.
You are downloading Microsoft.jQuery.Unobtrusive.Ajax from Microsoft, the l
available at http://www.microsoft.com/web/webpi/eula/net_library_eula_ENU.h
dependencies, which may come with their own license agreement(s). Your use
constitutes your acceptance of their license agreements. If you do not acce
delete the relevant components from your device.
Successfully installed 'Microsoft.jQuery.Unobtrusive.Ajax 3.2.3'.
Adding 'Microsoft.jQuery.Unobtrusive.Ajax 3.2.3' to Events.Web.
Successfully added 'Microsoft.jQuery.Unobtrusive.Ajax 3.2.3' to Events.Web.
```

- **Register the Unobtrusive AJAX script** in the bundles configuration file **`App_Start\BundleConfig.cs`**, just after the jQuery bundle registration:

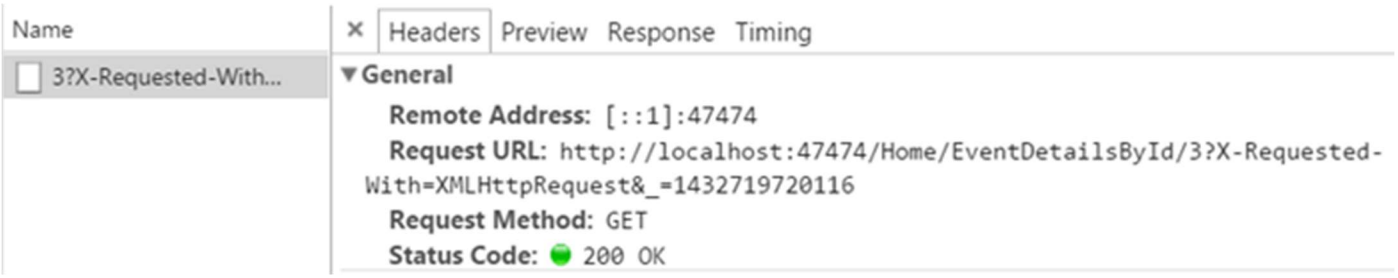
- **Include** the unobtrusive AJAX script in the view which needs AJAX: **`\Views\Home\Index.cshtml`**



This code will render the "**`~/bundles/ajax`**" bundle in the section "scripts" in the site layout:

8. Now the AJAX functionality should **work as expected**:

The **network requests** in the Web browser developer tools shows that the AJAX call was successfully executed:



Step 10. Create New Event

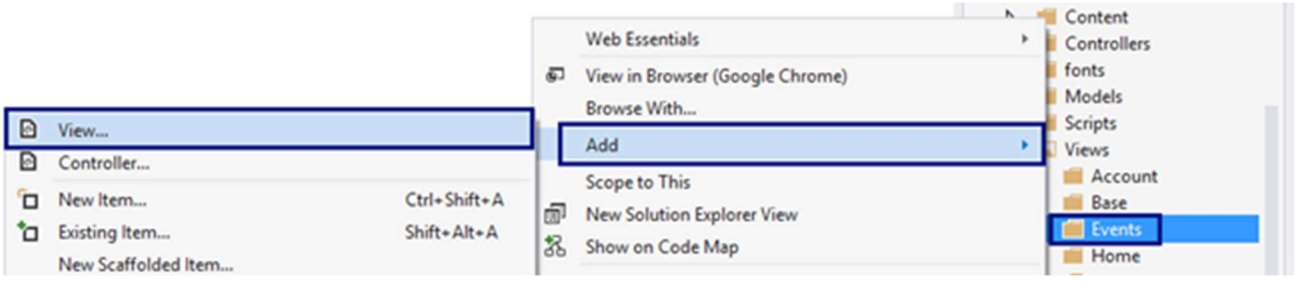
The next feature to be implemented in the Events management system is "**Create New Event**". This will require **creating a "New Event"** form (Razor view) + input **model** for the form + **controller action** to handle the submitted form data.

1. First, let's define the **input model** for the create or edit event form. It will hold all event properties that the user will fill when creating or editing an event. Let's create the class **`Models\EventInputModel.cs`**:

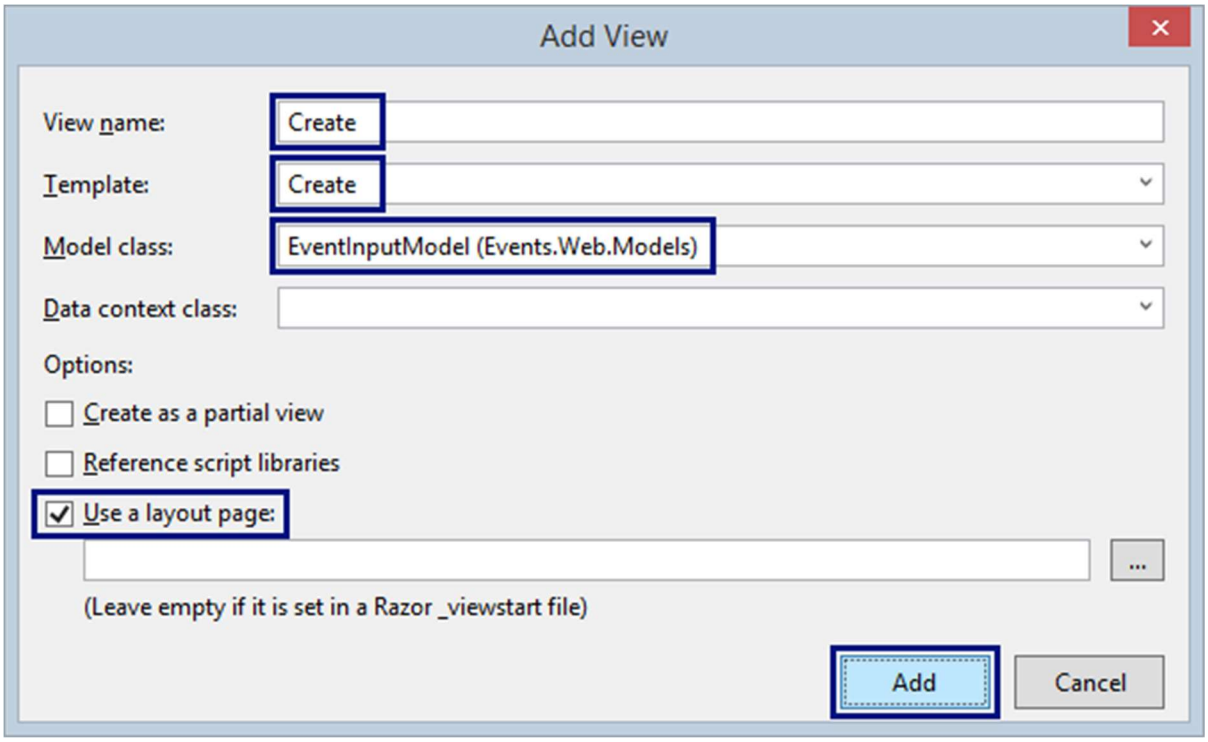
```
public class EventInputModel
{
    [Required(ErrorMessage = "Event title is required.")]
    [StringLength(200, ErrorMessage = "The {0} must be between {2} and {1} characters long.",
        MinimumLength = 1)]
    [Display(Name = "Title *")]
    public string Title { get; set; }

    [DataType(DataType.DateTime)]
    [Display(Name = "Date and Time *")]
    public DateTime StartDateTime { get; set; }

    public TimeSpan? Duration { get; set; }

    public string Description { get; set; }

    [MaxLength(200)]
    public string Location { get; set; }

    [Display(Name = "Is Public?")]
    public bool IsPublic { get; set; }
```

It is a good idea to attach **validation annotations** like **`[Required]`** and **`[MaxLength]`** for each property to simplify the validation of the form.

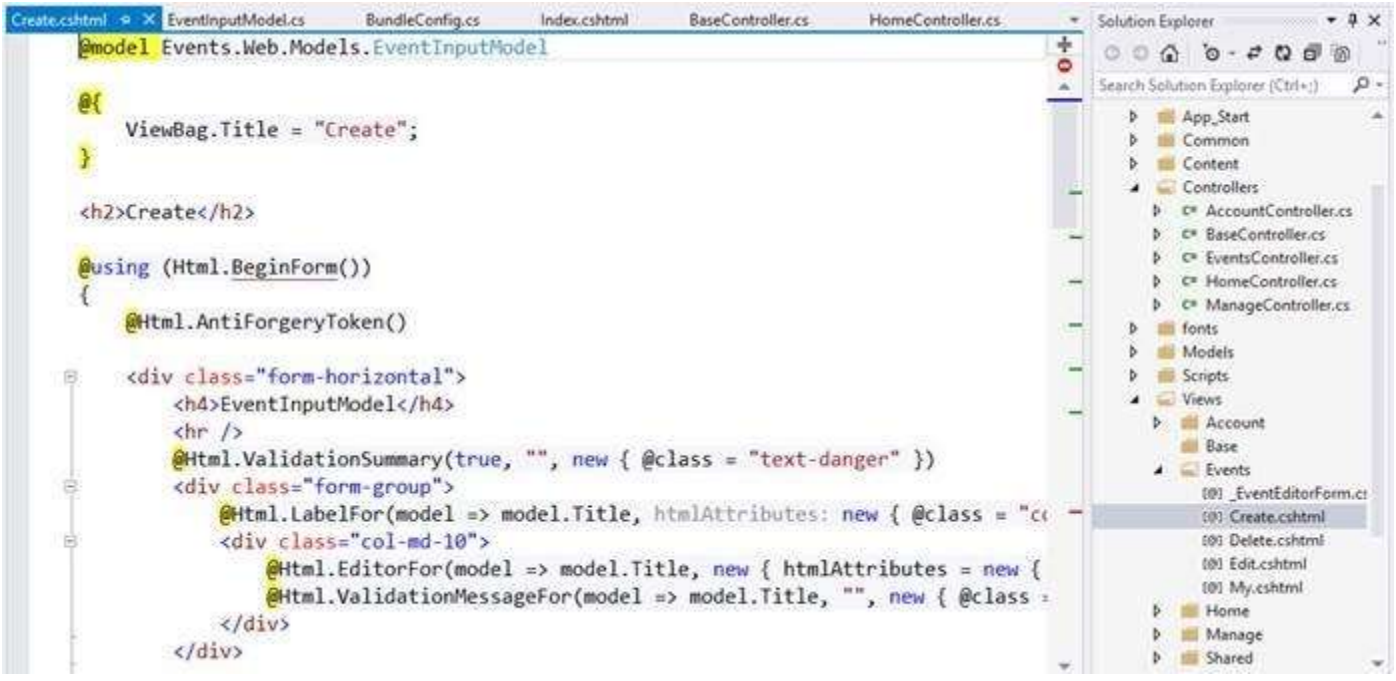
2. Next, let's **create the "New Event" form**. The easiest way to start is by using the **Razor view generator** in Visual Studio. Create a folder "**\Views\Events**". Right click on the "**`Events`**" folder and choose [Add] → [View…]:



Enter a **view name** "**`Create`**". Select **template** "**`Create`**". Select the **model class** "**`EventInputModel (Events.Web.Models)`**". Click **`[Add]`** to generate the view:



Visual Studio will **generate the "Create Event" form**:



3. Now **customize the generated form**. Change several things:
   - Change the Title → "Create Event"

```
@{
    ViewBag.Title = "Create New Event";
}

<h2>@ViewBag.Title</h2>
<hr />
```

- Remove the “Back to List” link:

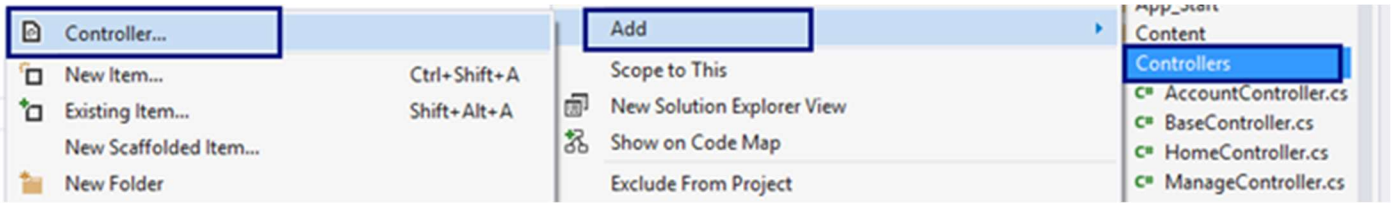
```
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

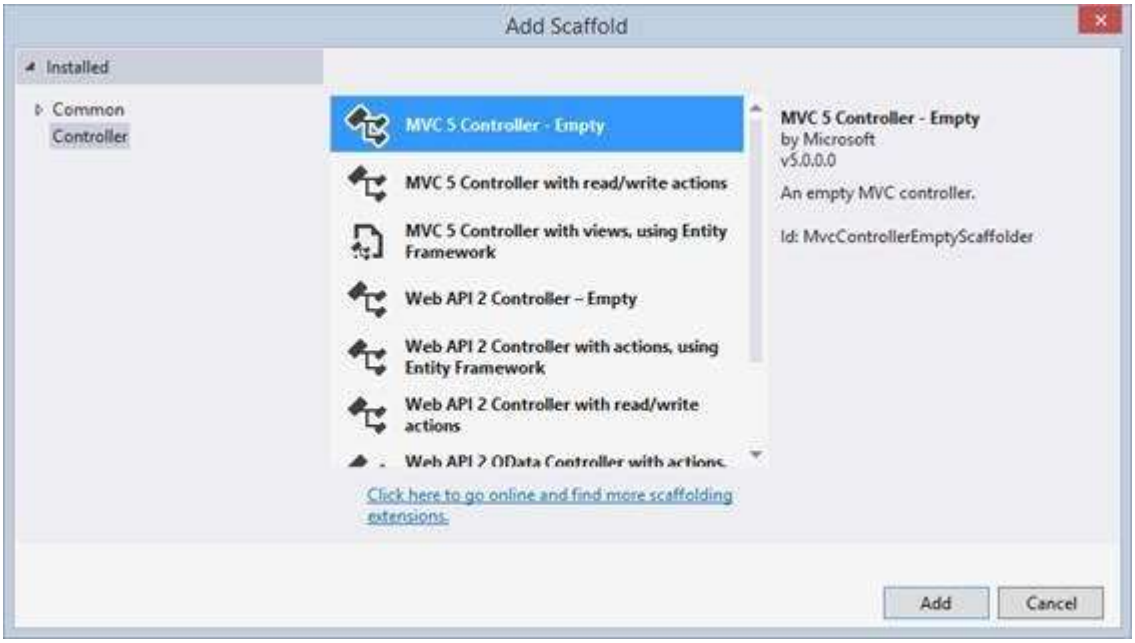
- Add **[Cancel]** link to “My Events”, just after the **[Create]** button:

Create.cshtml* | EventInputModel.cs | EventViewModel.cs | BundleConfig.cs | Index.cshtml | HomeController.cs

```
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
                @Html.ActionLink("Cancel", "My", null, htmlAttributes: new { @class = "btn" })
            </div>
        </div>
    </div>
}
```

4. Create **`EventsController`** to handle the actions related to events: **create** / **edit** / **delete** / **list** events.

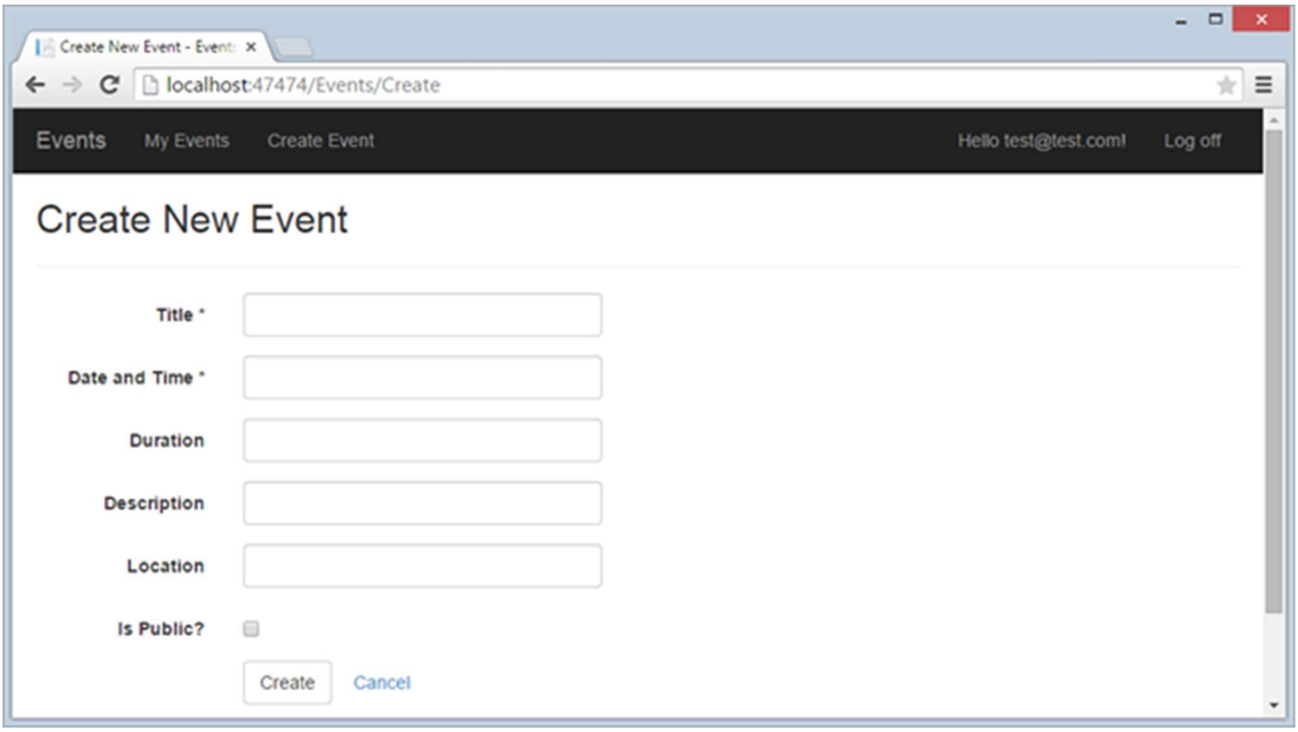




5. Add the **"Create" action** (HTTP GET) to display the "New Event" form:

```
namespace Events.Web.Controllers
{
    0 references
    public class EventsController : Controller
    {
        // GET: Events/Create
        0 references
        public ActionResult Create()
        {
            return View();
        }
    }
}
```

6. Build the project, run it and **test** the new functionality (**login** and click **[Create Event]**). The "New Event" form should be rendered in the Web browser:



Create New Event Logic

After the "Create New Event" form is ready, it is time to write the **logic** behind it. It should create a new event and save it in the database. The correct way to handle form

submissions in ASP.NET MVC is by **HTTP POST** action in the controller behind the form. Let's write the "Create New Event" logic.

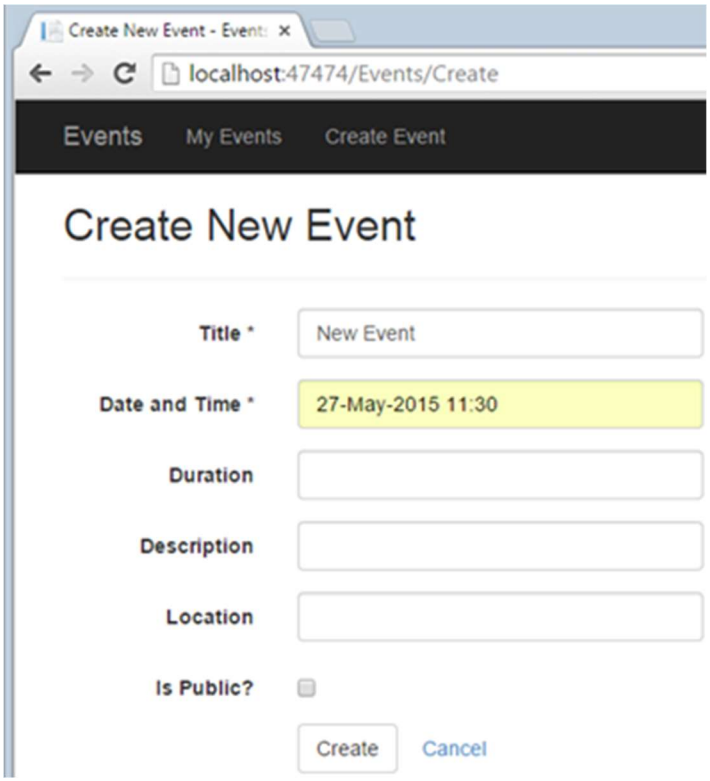
1. First, extend the **`BaseController`** to inherit the database context:

```
public class EventsController : BaseController
{
    // GET: Events/Create
    -references
    public ActionResult Create()...
```

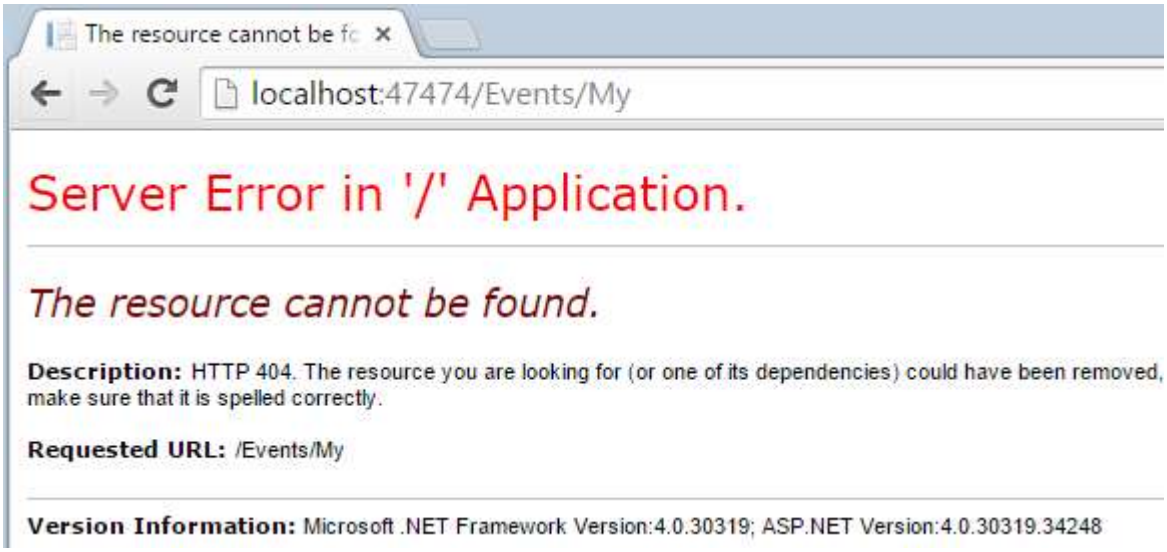
2. Next, write the action to handle **`POST \Events\Create`**:

```
[HttpPost]
[ValidateAntiForgeryToken]
-references
public ActionResult Create(EventInputModel model)
{
    if (model != null && this.ModelState.IsValid)
    {
        var e = new Event()
        {
            AuthorId = this.User.Identity.GetUserId(),
            Title = model.Title,
            StartDateTime = model.StartDateTime,
            Duration = model.Duration,
            Description = model.Description,
            Location = model.Location,
            IsPublic = model.IsPublic
        };
        this.db.Events.Add(e);
        this.db.SaveChanges();
        // Display notification message "Event created."
        return this.RedirectToAction("My");
    }

    return this.View(model);
}
```

3. Run and **test the code**. Now creating events almost works:



When the **form is submitted**, the **result** will be like this:



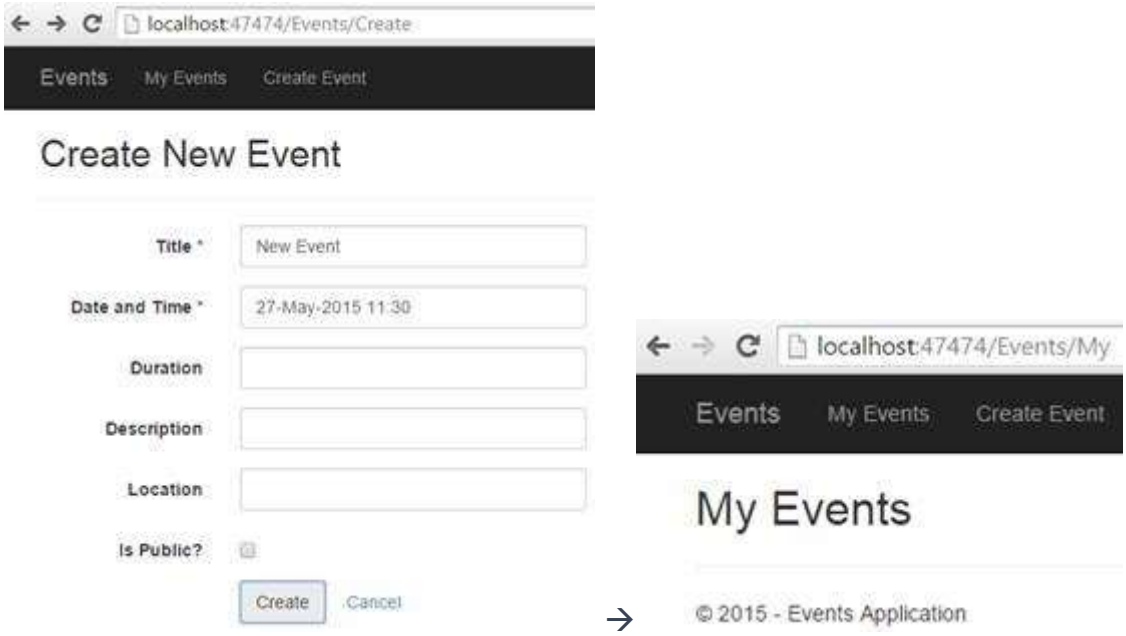
This is quite normal. The "**`My`**" action in the **`EventsController`** is not missing.

4. Let's define the action "**`My`**" and the view behind it in the **`EventsContoller`**:

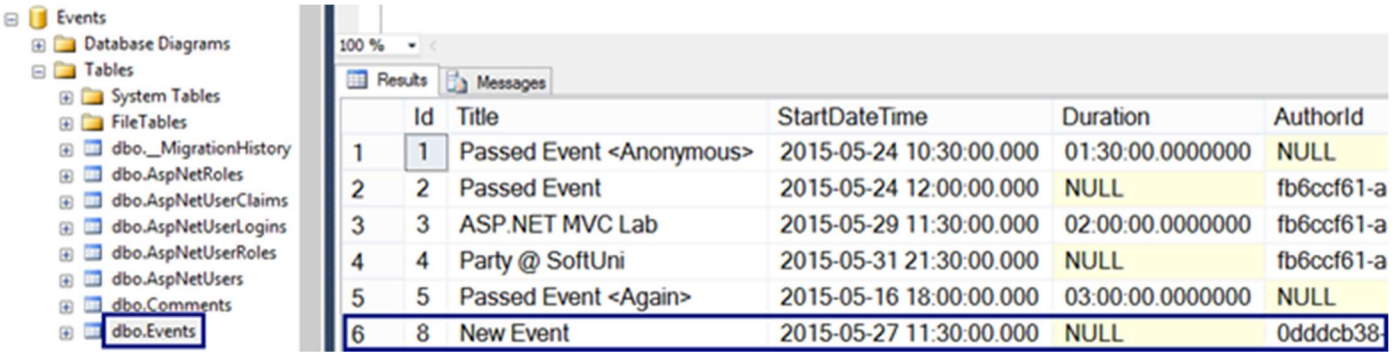
```
public class EventsController : BaseController
{
    // GET: Events/My
    0 references
    public ActionResult My()
    {
        return View();
    }
```

Create an empty view **`\Views\Events\My.cshtml`** (it will be implemented later):

5. Now run and test the application again:



Check the database to see whether the new event is created:



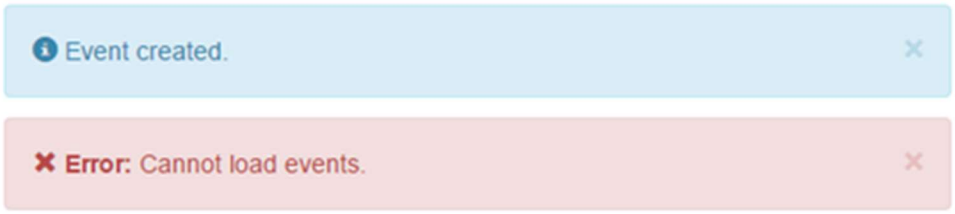
	Id	Title	StartDateTime	Duration	AuthorId
1	1	Passed Event <Anonymous>	2015-05-24 10:30:00.000	01:30:00.0000000	NULL
2	2	Passed Event	2015-05-24 12:00:00.000	NULL	fb6ccf61-a
3	3	ASP.NET MVC Lab	2015-05-29 11:30:00.000	02:00:00.0000000	fb6ccf61-a
4	4	Party @ SoftUni	2015-05-31 21:30:00.000	NULL	fb6ccf61-a
5	5	Passed Event <Again>	2015-05-16 18:00:00.000	03:00:00.0000000	NULL
6	8	New Event	2015-05-27 11:30:00.000	NULL	0dddcb38-

6. At the home page the new event is not listed. Why? The event is not public. Let's make the **"Is Public?" check box** in the "New Event" form **checked by default**:

```
<div class="form-group">
    @Html.LabelFor(model => model.IsPublic, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        <div class="checkbox">
            @Html.CheckBoxFor(model => model.IsPublic, htmlAttributes: new { @checked = "true" })
            @Html.ValidationMessageFor(model => model.IsPublic, "", new { @class = "text-danger" })
        </div>
    </div>
</div>
```

Step 11. Notification System (Info / Error Messages)

The next big step will be to **add a notification system** to display informational and error messages like these:



Typically, MVC controller actions that modify DB objects (e.g. `EventsController.Create`) work as follows:

- **Check the model state**. If the validation fails, the model state will be invalid and the controller will **re-render the form**. The form will show the errors for each incorrect field.
- If the model state is correct, **create / edit the database object** behind the form.
- Add a **notification message** to be shown at the top of the next page shown in the Web application.
- **Redirect** to another page that lists the created / modified DB object.

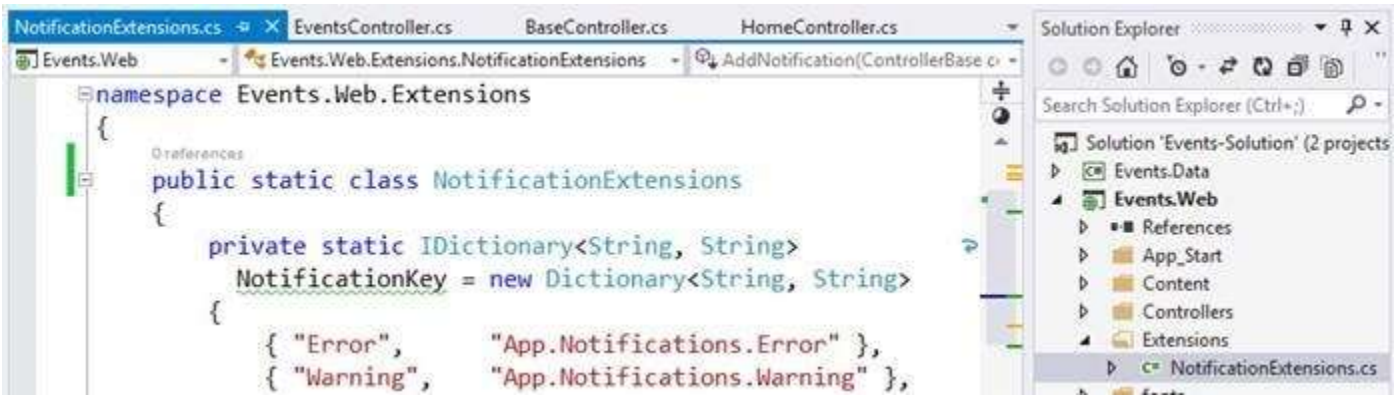
As the **"Post-Redirect-Get" pattern** says, in Web applications it is highly recommended to redirect to another page after executing an action that changes something at the server side:http://en.wikipedia.org/wiki/Post/Redirect/Get.

The missing part in ASP.NET MVC is the **notification system**, so developers should either create it, or **use some NuGet package** that provides notification messages in ASP.NET MVC. Let's install and use the NuGet package`BootstrapNotifications`:

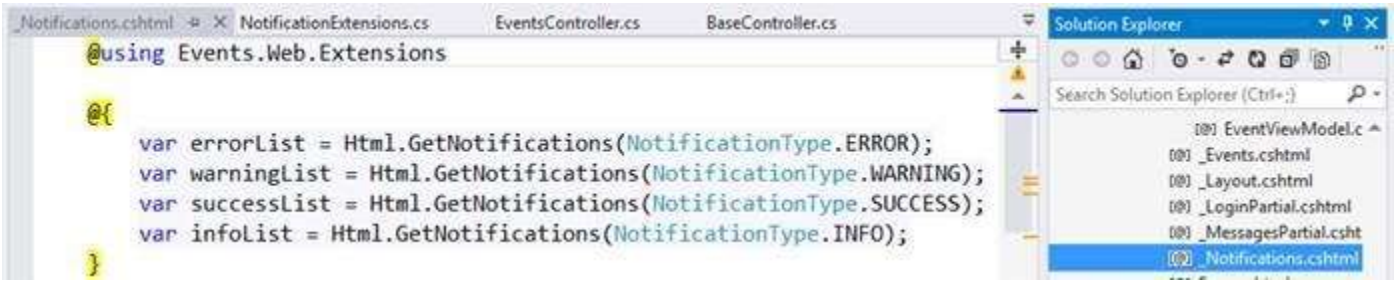
1. From the package management console in Visual Studio **install the NuGet package `BootstrapNotifications`** in the **`Events.Web`** project:

```
Package Manager Console
Package source: nuget.org    Default project: Events.Web
PM> Install-Package BootstrapNotifications
Attempting to resolve dependency 'bootstrap (≥ 3.0.0)'.
Attempting to resolve dependency 'jQuery (≥ 1.9.1)'.
Installing 'BootstrapNotifications 0.3.2'.
```

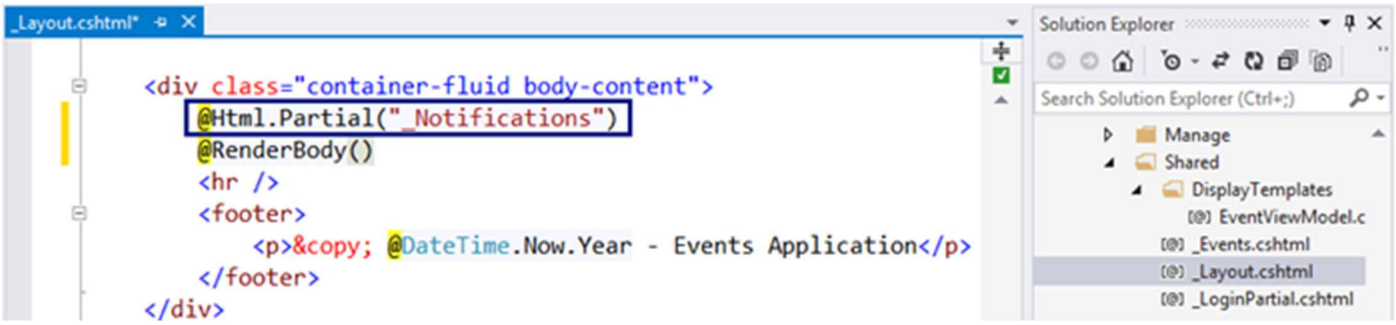
NuGet will add the class **`Extensions\NotificationExtensions.cs`**:



NuGet will also add a partial view **`\Views\Shared\_Notifications.cshtml`**:



2. To display the notification messages (when available) at the top of each page in the MVC project, render the **`_Notifications` partial view** in the site layout, just before the **`@RenderBody()`**:



3. **Put notifications** in the controller actions after successful database change.

Use **`NotificationType.INFO`** for information messages (success) and **`NotificationType.ERROR`** for error messages. Not that these notifications will

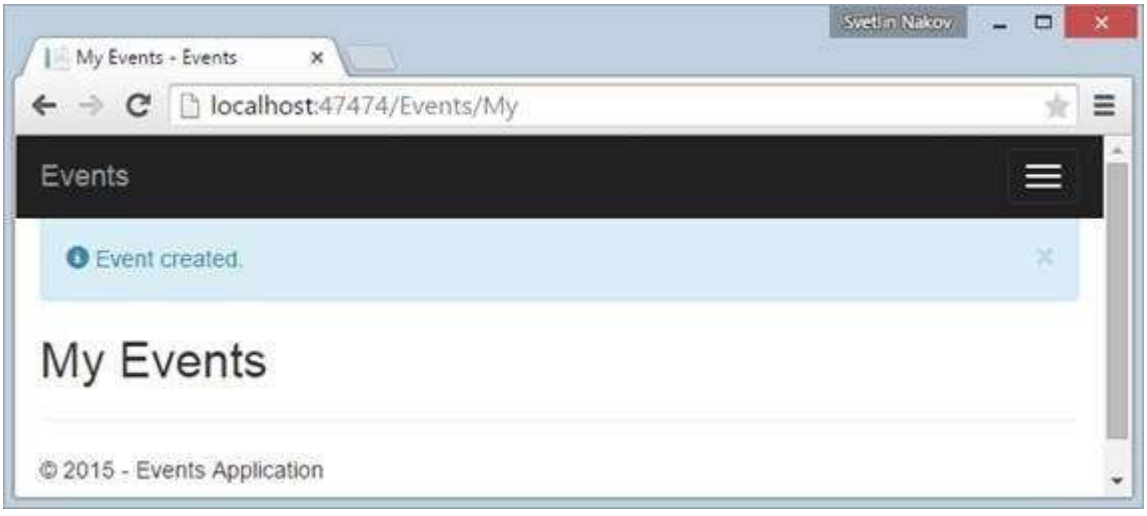
render after the first **redirect to another page**, because the implementation relies on the **`TempData`** dictionary in ASP.NET MVC.

Add notification messages after **`db.SaveChanges()`** in the actions that change the database, followed by **`RedirectToAction(…)`**:

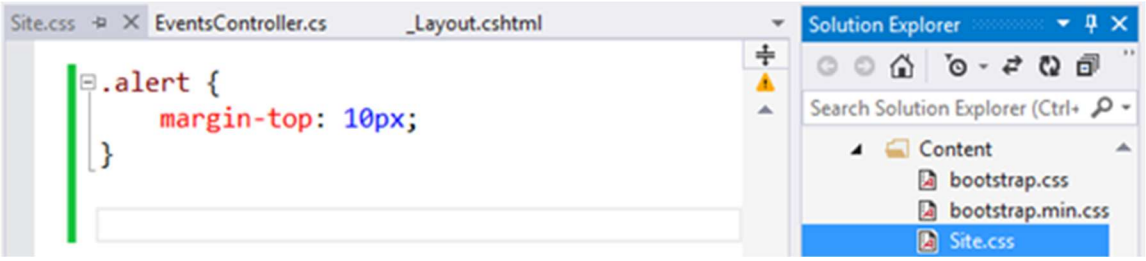
```
EventsController.cs  _Layout.cshtml
Events.Web  Events.Web.Controllers.EventsController

        // POST: Events/Create
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Create(EventInputModel model)
        {
            if (model != null && this.ModelState.IsValid)
            {
                var e = new Event()
                {
                    AuthorId = this.User.Identity.GetUserId(),
                    Title = model.Title,
                    StartDateTime = model.StartDateTime,
                    Duration = model.Duration,
                    Description = model.Description,
                    Location = model.Location,
                    IsPublic = model.IsPublic
                };
                this.db.Events.Add(e);
                this.db.SaveChanges();
                this.AddNotification("Event created.", NotificationType.INFO);
                return this.RedirectToAction("My");
            }

            return this.View(model);
        }
```

Add this "using":
`using Events.Web.Extensions;`

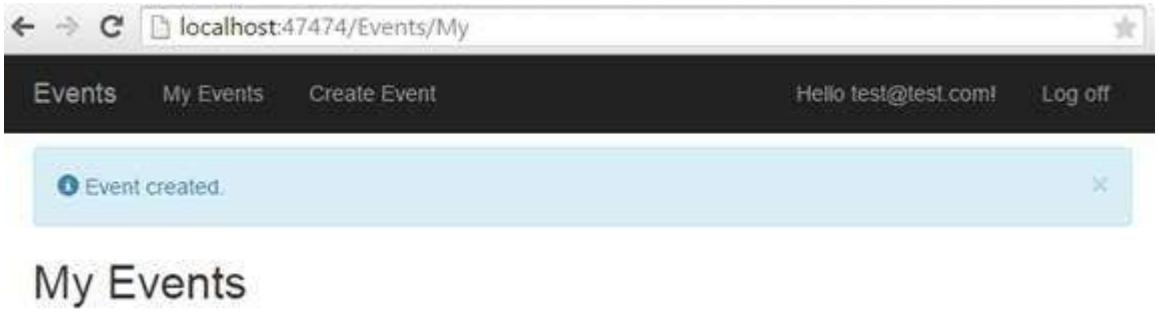
4. Run and **test the application**. Create an event to see the "**Event created.**" notification message:



This looks a bit ugly, so add a fix in the **`\Content\Site.css`**:



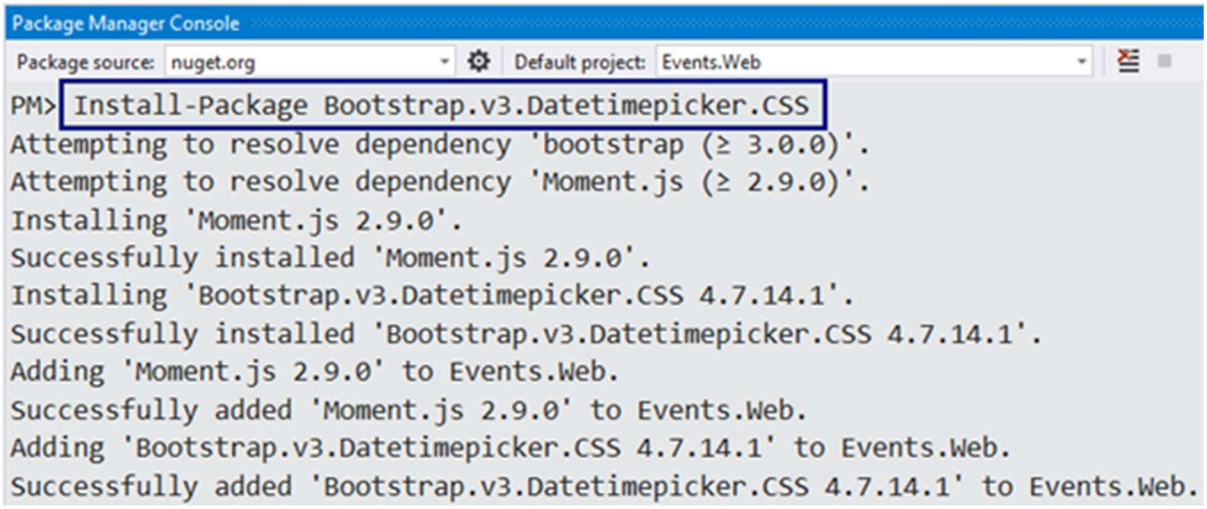
5. Save the changes, refresh the site with [Ctrl + F5] and create a new event to **test the changes**:



Step 12. Date / Time / Duration UI Controls

The "New Event" form displays **fields for entering date + time and duration**. Currently the editor for these fields is not user-friendly. The date format is unclear and there is not calendar control. Let's fix this. Let's **add "date-time picker"** for the date and duration fields:

1. Install the NuGet package "**`Bootstrap.v3.Datetimepicker.CSS`**":



It will add the following files to your MVC project:

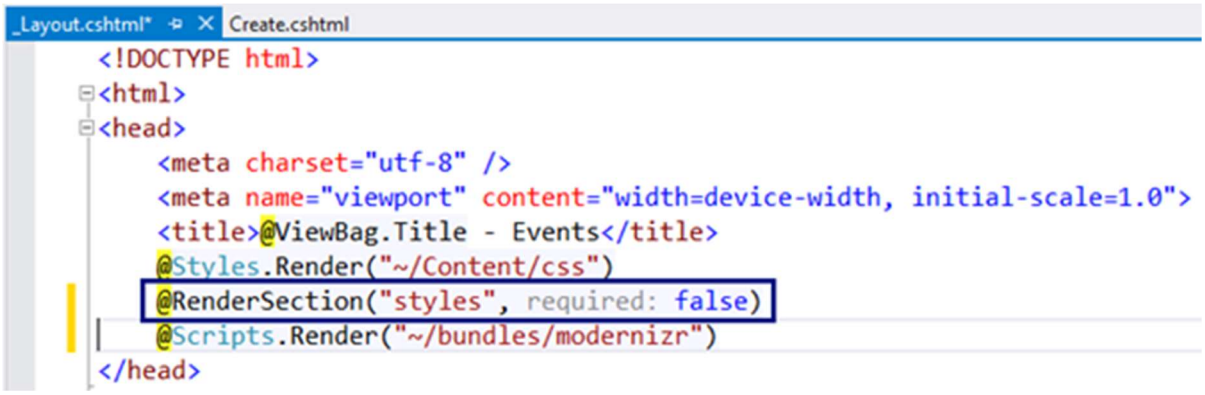
- **`\Scripts\bootstrap-datetimepicker.js`**
- **`\Scripts\moment.js`**

- **`\Content\bootstrap-datetimepicker.css`**

These files should be includes in all pages that use the date-time picker.

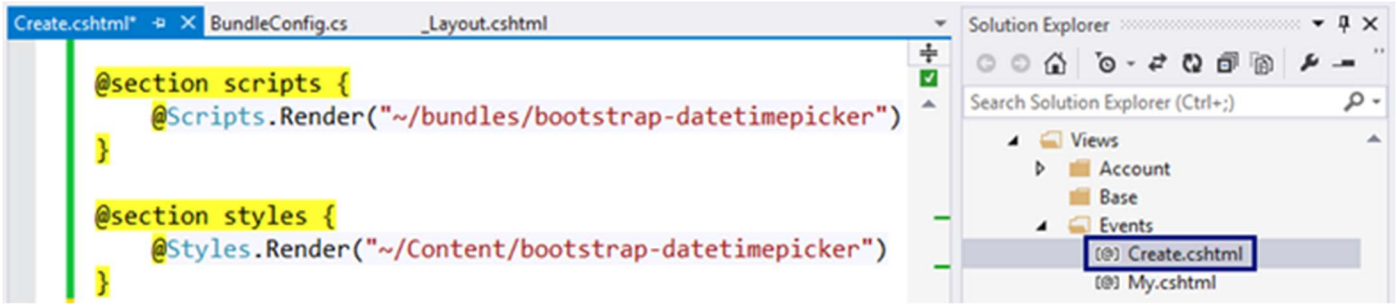
2. Create CSS and JavaScript **bundles** for the date-time picker:

3. Create a **placeholder for the CSS bundles** in the **`_Layout.cshtml`**:

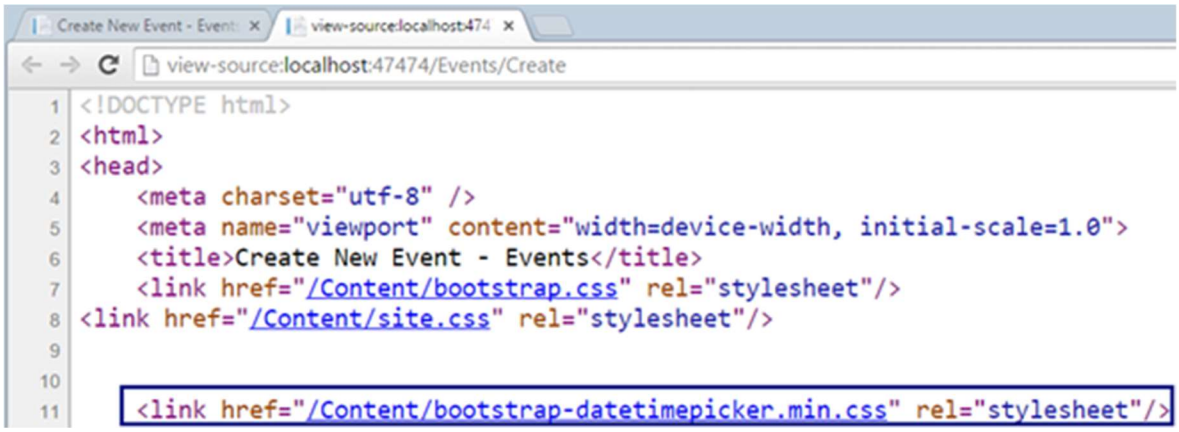


It will be used later to inject the date-time picker's CSS from the events editor form.

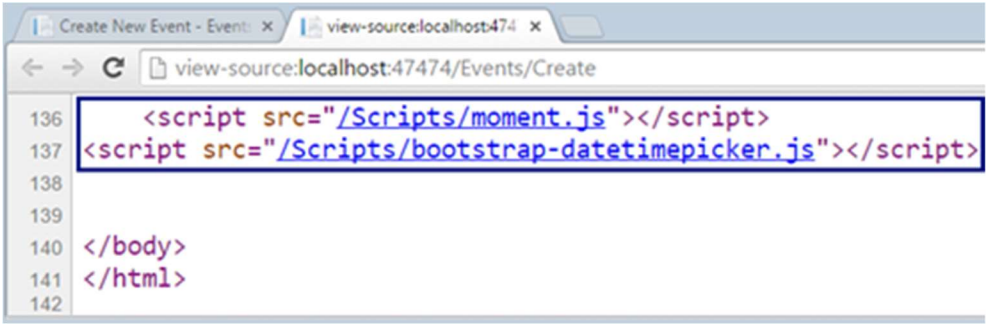
4. Add the date-time picker's scripts and CSS files in the "New Event" form, in **`\Views\Create.cshtml`**:



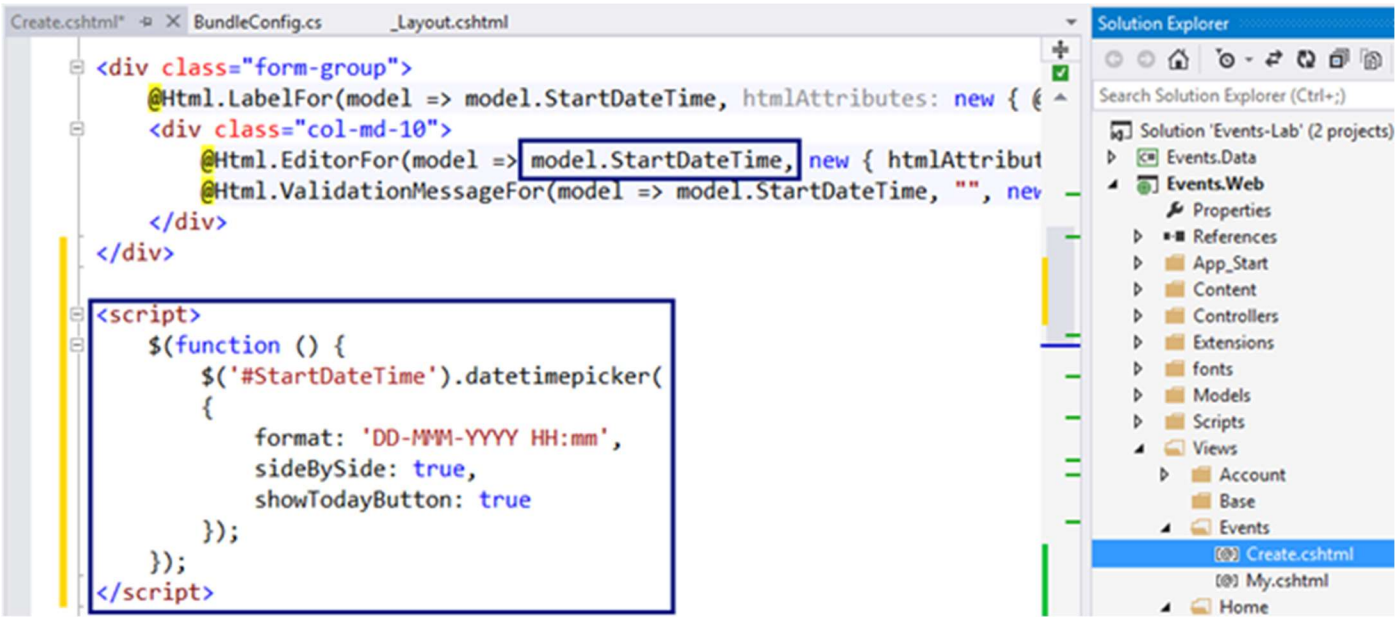
This code will inject the specified CSS from the bundles in the page header:



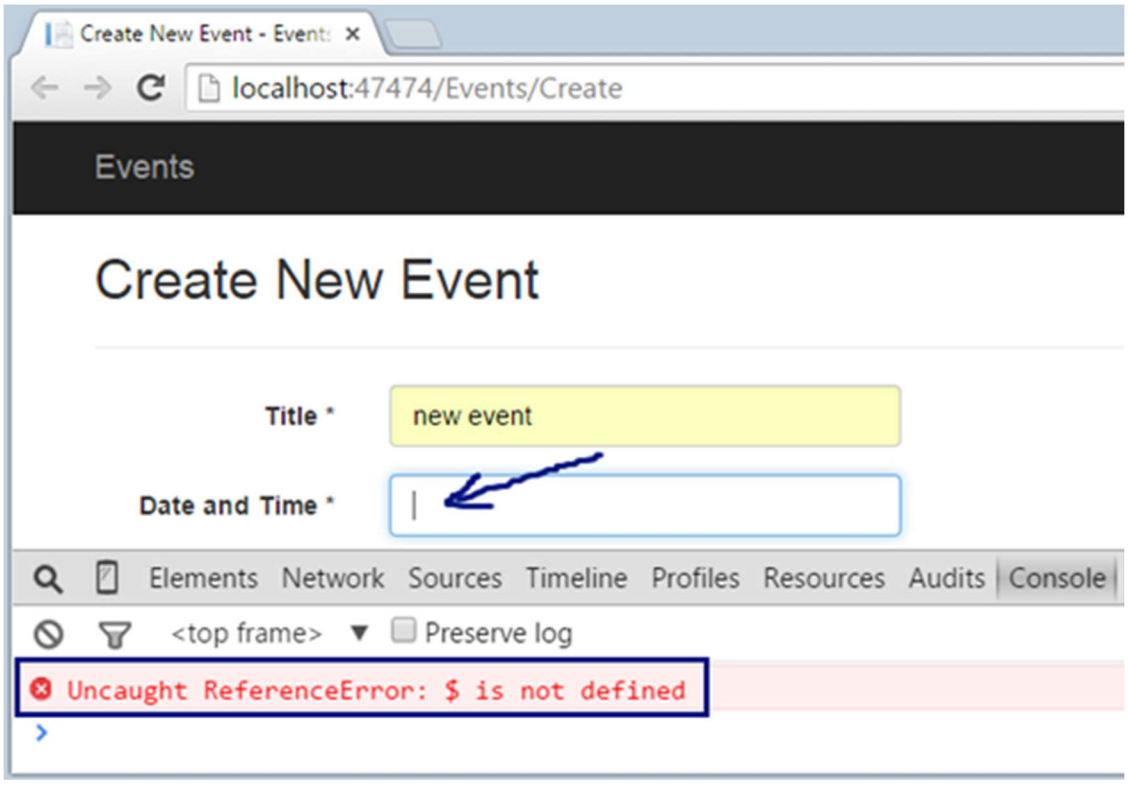
It will also inject the specified JavaScript from the bundles at the end of the page:



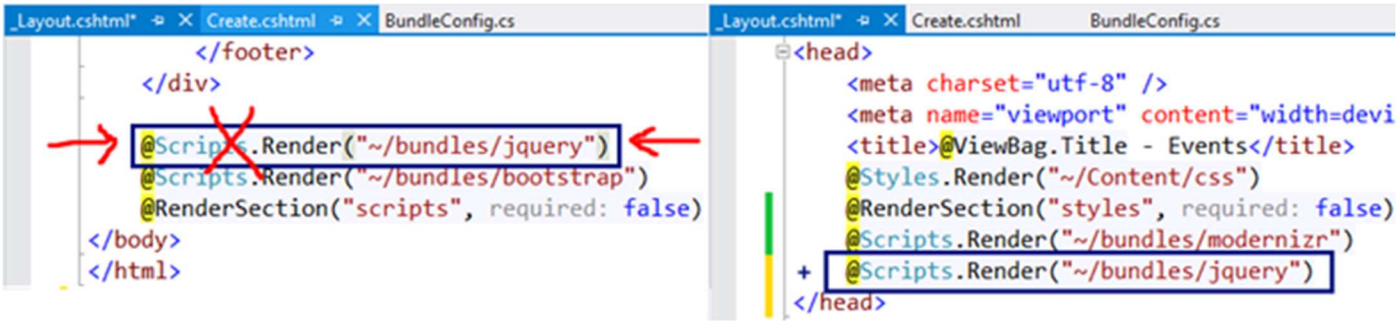
5. Now the site is ready to add the date-time picker in the "New Event" form. Let's try to attach the date-time picker for the field "**`StartDateTime`**" in**`\Views\Events\Create.cshtml`**:



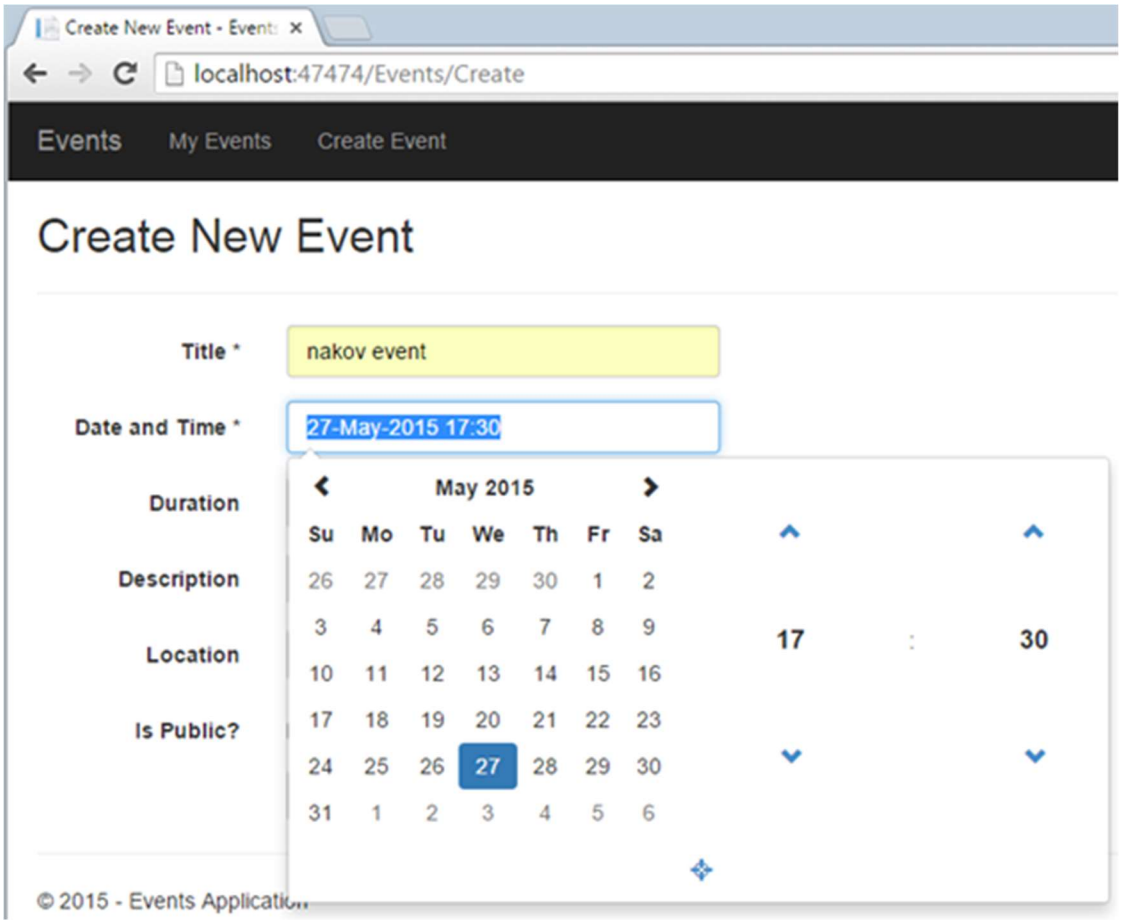
6. Now rebuild the project and run the code **to test the new functionality**. It does not work due to JavaScript error:



7. Seems like **jQuery is not loaded**. This is because the script for attaching the **`datetimepicker`** for the **`StartDateTime`** field uses jQuery, but the jQuery library loads later, at the end of the HTML document. This is easy to fix, just move the jQuery reference at the start of the HTML code in **`_Layout.cshtml`**:



8. Now rebuild the project and run the code **to test again the new functionality**. It should now work correctly:

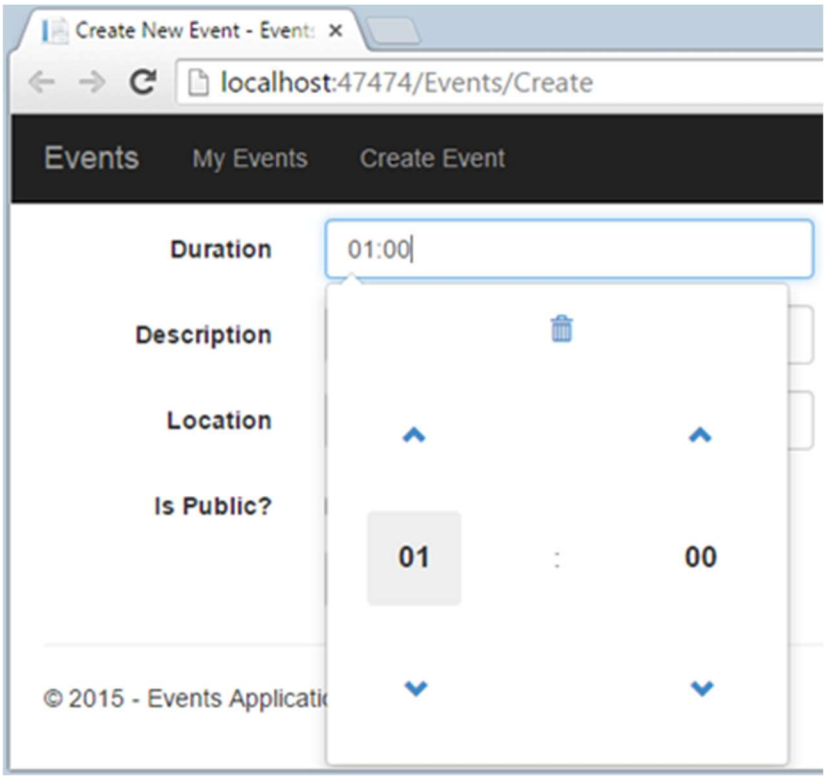


9. In similar way, add a time picker for the "**`Duration`**" field:

Create.cshtml* | BundleConfig.cs

```
<div class="form-group">
    @Html.LabelFor(model => model.Duration, htmlAttributes:
    <div class="col-md-10">
        @Html.EditorFor(model => model.Duration, new { html
        @Html.ValidationMessageFor(model => model.Duration,
    </div>
</div>

<script>
    $(function () {
        var date = new Date();
        date.setHours(1);
        date.setMinutes(0);
        $('#Duration').datetimepicker(
        {
            format: 'HH:mm',
            defaultDate: date,
            showClear: true
        });
    });
</script>
```

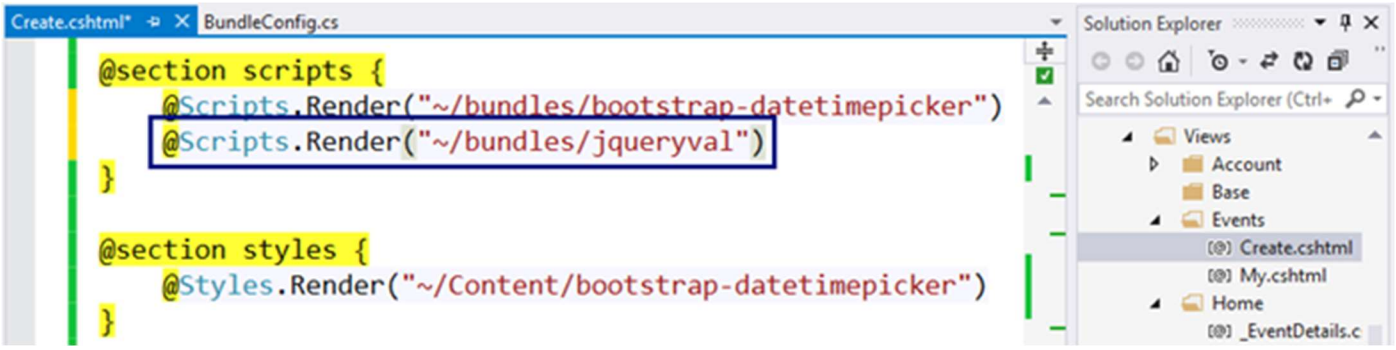
10. **Test** the new duration picker by starting the Web application:



Step 13. Client-Side Unobtrusive Validation

Now the “New Event” form works as expected. There is a **small UI problem**: when invalid data is entered, the **form validation is executed at the server side**and the user sees the validation errors after post-back, with a small delay.

Let’s try to **add client-side form validation**. This is really easy, just insert the Microsoft jQuery Unobtrusive validation JavaScript bundle at the page holding the form:



Test the “New Event” form to ensure the validation is now client side.

Step 14. List My Events

Now let’s list current user’s events at its personal events page: **`\Events\My`**.

1. First, add **`[Authorize]`** attribute in the **`EventsController`**:

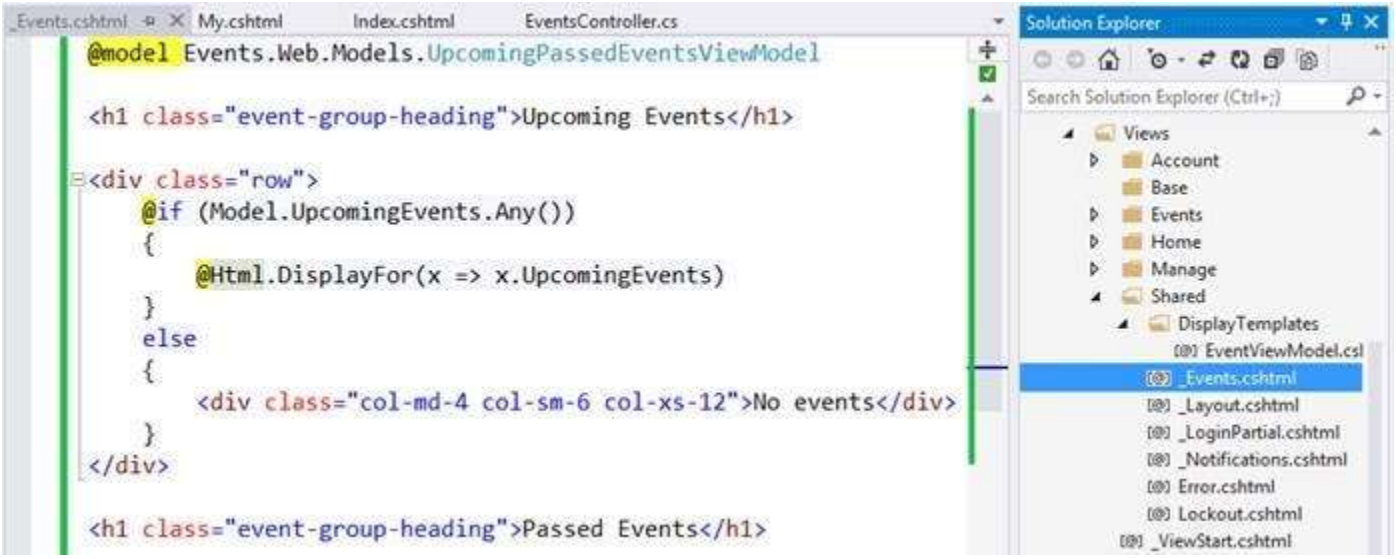
```
namespace Events.Web.Controllers
{
    using ...

    [Authorize]
    public class EventsController : BaseController
    {
```

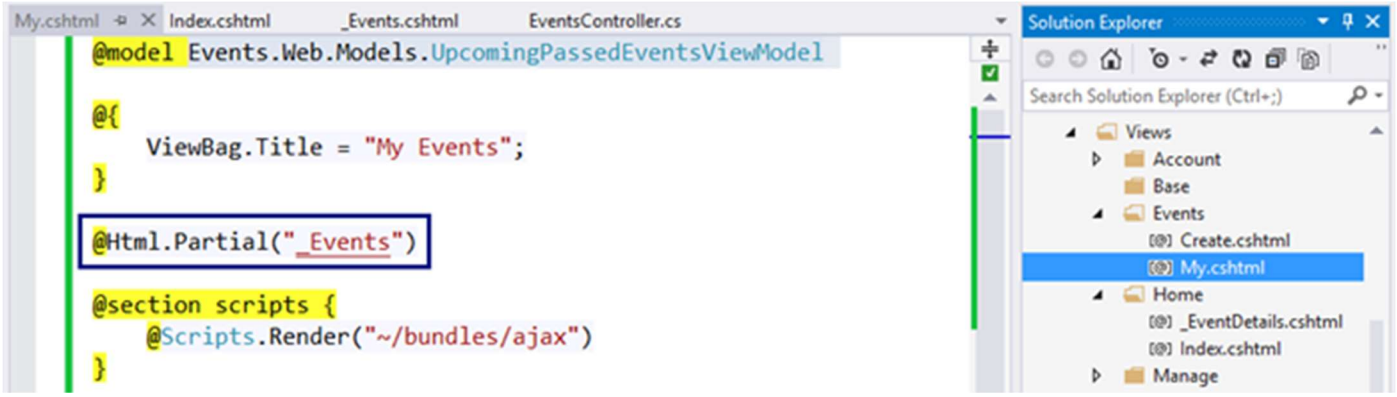
The **`[Authorize]`** attribute will redirect the anonymous users to the login form. It says that all controller actions of the **`EventsController`** can be accessed by logged-in users only.

2. Next, add the HTTP GET controller action “**`My`**” in **`EventsController`** that will display current user’s events:

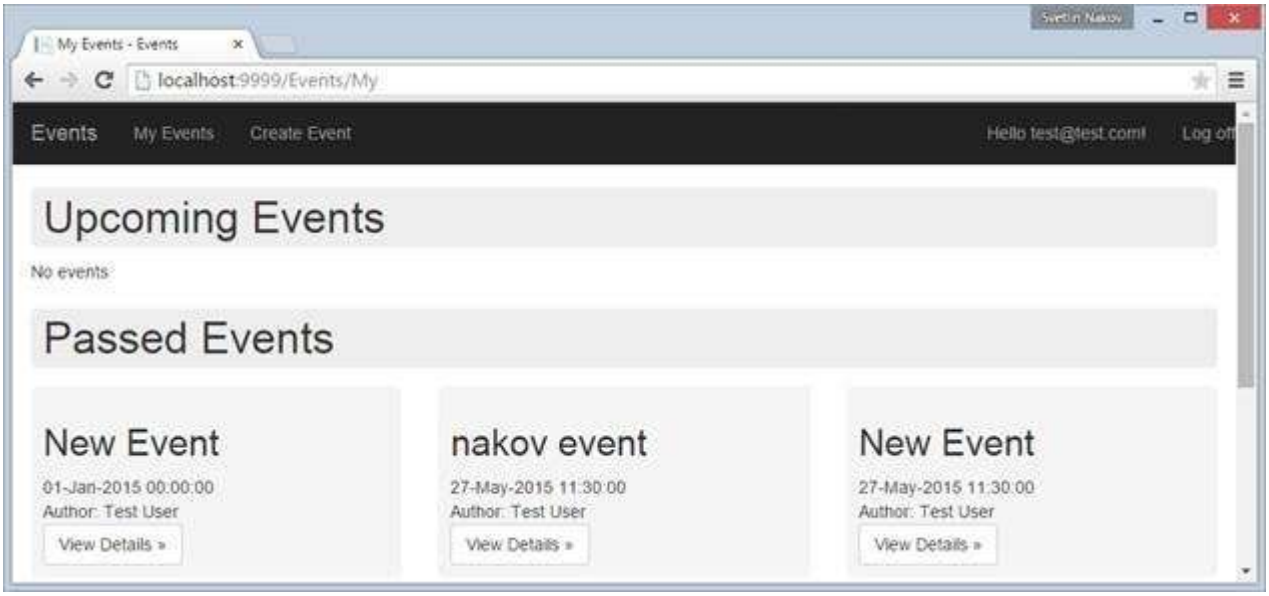
```
public ActionResult My()
{
    string currentUserId = this.User.Identity.GetUserId();
    var events = this.db.Events
        .Where(e => e.AuthorId == currentUserId)
        .OrderBy(e => e.StartDateTime)
        .Select(EventViewModel.ViewModel);

    var upcomingEvents = events.Where(e => e.StartDateTime > DateTime.Now);
    var passedEvents = events.Where(e => e.StartDateTime <= DateTime.Now);
    return View(new UpcomingPassedEventsViewModel()
    {
        UpcomingEvents = upcomingEvents,
        PassedEvents = passedEvents
    });
}
```

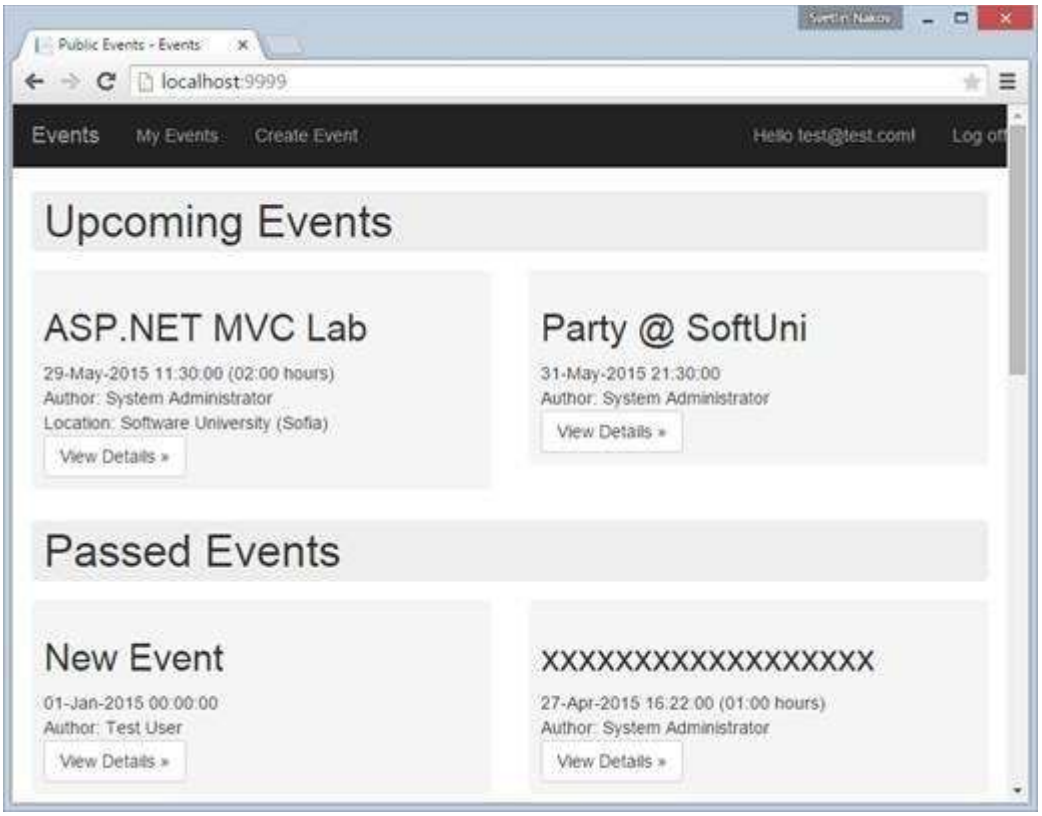
3. Finally, create the view **`My.cshtml`** behind the above action. It is essentially the same like the **`Index.cshtml`** view of the **`HomeController`**, right? Duplicating code is very bad practice, so let's **reuse the code**. First, extract a partial view **`\Views\Shared\_Events.cshtml`**, then reference it from**`\Views\Events\My.cshtml`** and again from **`\Views\Home\Index.cshtml`**:



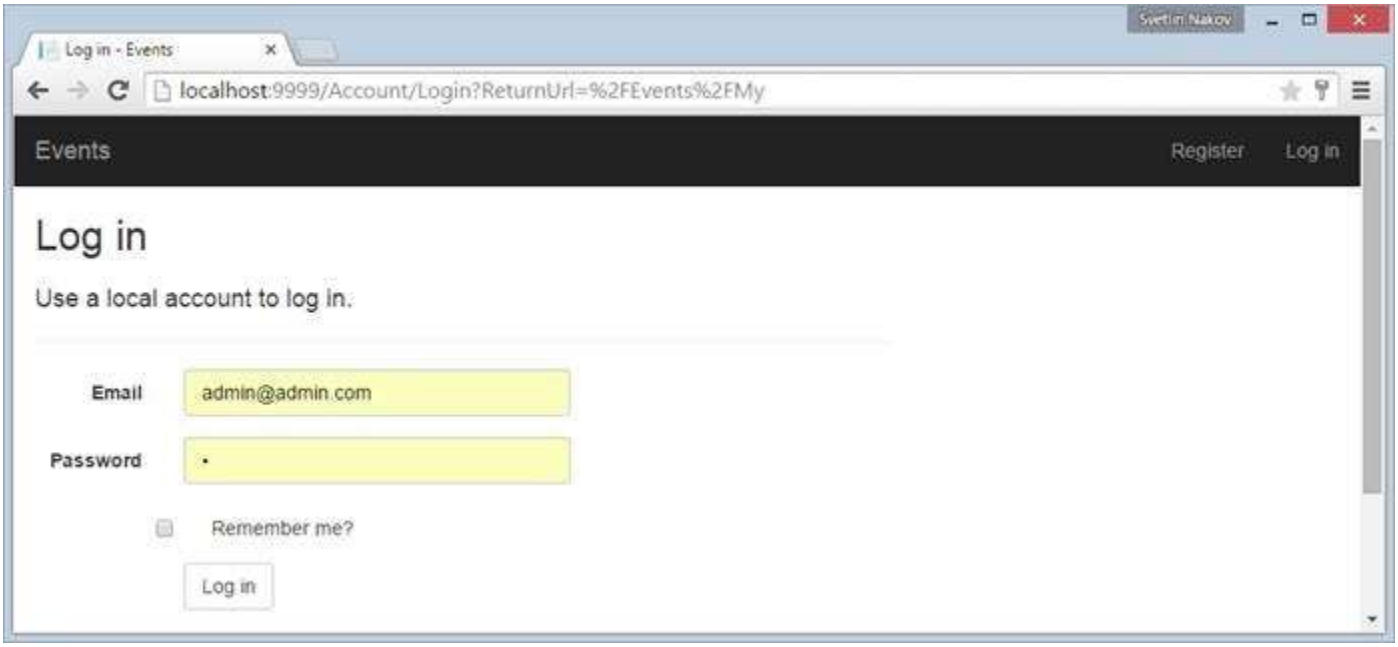


4. Test the new functionality "**My Events**", as well as the old functionality "**Home Page**":





5. Try also to access "**My Events**" for anonymous user. The application should redirect you to the login page:



Step 15. Edit Existing Event Form

The "**Edit Event**" functionality is very similar to "**Create Event**". It uses the same input model **`Events.Web.Models.UpcomingPassedEventsViewModel`**. Create a

view **`\Views\Events\Edit.cshtml`** and reuse the logic from **`\Views\Events\Create.cshtml`** by extracting a partial view **`\Views\Events\_EventEditorForm`**.

Step 16. Edit Existing Event Logic

Write the controller action for editing events in the **`EventsController`**.

1. Write a HTTP **GET action "Edit"** to prepare the form for editing event. In case of invalid event ID, show an error message and **redirect to** "**My Events**":

```
[HttpGet]
0 references
public ActionResult Edit(int id)
{
    var eventToEdit = this.LoadEvent(id);
    if (eventToEdit == null)
    {
        this.AddNotification("Cannot edit event #" + id,
          NotificationType.ERROR);
        return this.RedirectToAction("My");
    }

    var model = EventInputModel.CreateFromEvent(eventToEdit);
    return this.View(model);
}
```

2. The logic in **`LoadEvent(id)`** method loads an existing event in case the user has permissions to edit it:

```
private Event LoadEvent(int id)
{
    var currentUserId = this.User.Identity.GetUserId();
    var isAdmin = this.IsAdmin();
    var eventToEdit = this.db.Events
        .Where(e => e.Id == id)
        .FirstOrDefault(e => e.AuthorId == currentUserId || isAdmin);
    return eventToEdit;
}
```

3. Write a HTTP **POST action "Edit"** to save the changes after submitting the event editing form. It should first **check the event ID** and show an error of case of non-existing event or missing permissions. Then it **checks for validation errors**. In case of validation errors, the same form is rendered again (it will show the validation errors). Finally, the **`Edit`** method modifies the database and redirects to "**My Events**":

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public ActionResult Edit(int id, EventInputModel model)
{
    var eventToEdit = this.LoadEvent(id);
    if (eventToEdit == null)
    {
        this.AddNotification("Cannot edit event #" + id, NotificationType.ERROR);
        return this.RedirectToAction("My");
    }

    if (model != null && this.ModelState.IsValid)
    {
        eventToEdit.Title = model.Title;
        eventToEdit.StartDateTime = model.StartDateTime;
        eventToEdit.Duration = model.Duration;
        eventToEdit.Description = model.Description;
        eventToEdit.Location = model.Location;
        eventToEdit.IsPublic = model.IsPublic;

        this.db.SaveChanges();
        this.AddNotification("Event edited.", NotificationType.INFO);
        return this.RedirectToAction("My");
    }

    return this.View(model);
}
```

4. Run and test the new functionality “Edit Event” by opening **`\Events\Edit\3`**.

Step 17. Handling HTML Special Characters

Try to create an event named “**`<New Event>`**“:

It will fail, because by default ASP.NET MVC does not allow forms to send HTML tags in the fields values. This is to help protecting from Cross-Site Scripting (XSS) attacks. You will see an **ugly error** page:

This is very **easy to fix**, just add **`[ValidateInput(false)]`** attribute in the **`BaseController`**:

```
[ValidateInput(false)]
public class BaseController : Controller
{
    protected ApplicationDbContext db = new ApplicationDbContext();
```

Step 18. Delete Existing Event Form

Create a form and controller action, **similar to "Edit Event"**. The form should load the event data in read-only mode with **[Confirm]** and **[Cancel]** buttons.

Step 19. Delete Existing Event Logic

The logic behind the "Delete Event" from is **similar to "Edit Event"**. Try to implement is yourself.

Step 20. Add Comments with AJAX

For each comment implement **[Add Comment]** button that shows (with AJAX) a new comment form. After the form is submitted, it creates a comment and appends it after the other comments for the current event. Comments can be added by the currently logged-in user and anonymously (without login).

Step 21. Delete Comments

Comments can be deleted only by their **author** (owner) and by **administrators**. Display **[Delete]** button right after each comment in the events (if deleting is allowed). Clicking **[Delete]** should delete the comment after modal popup **confirmation**.

Step 22. Add Paging

Implement **paging for the events**. Display 15 events per page. At each page display the **pager** holding the number of pages as well as **[Next Page]** / **[Previous Page]** links.

Step 23. Upload and Display Event Image

Modify the “Create Event” form to **upload an image** for the event (optionally). The image should be at most 100 KB, max width 400, max height 400, min width 100 and min height 100 pixels. Images should be **gif**, **png** or**jpeg** files, stored in the file system, in files named like this: **`event-185.png`**. Hold the images in a directory “**`~/Images`**“. Protect the directory to disable direct HTTP access to it.

Create an action **`/event/image/{id}`** to download and display an event image. It should read the image from the file system and display it if the user is allowed to view it (owner or administrator or public event).

Modify the event listing views to **display the image** for each event (when available). Use max-width and max-height attributes to ensure the image will resize correctly for different sizes of its container.