# PART 4 ASP.NET MVC Lab – Events

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 2017** with the latest available updates. Let's start building it step by step.

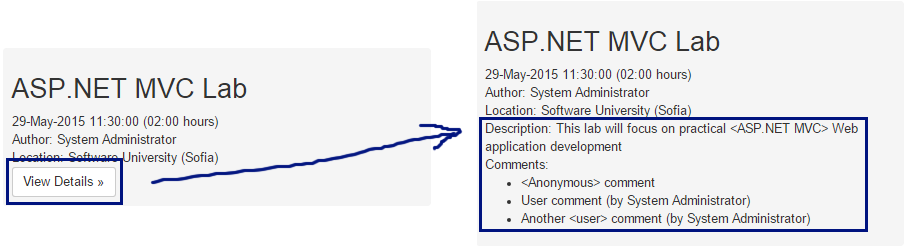
## Project Assignment

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

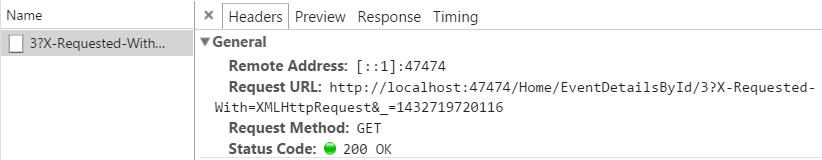
* **Events** have **title**, **start date** and optionally **start time**. Events may have also (optionally) **duration**, **description**, **location** and **author**. Events can be **public** (visible by everyone) and **private** (visible to their owner author only). Events may have **comments**. **Comments** 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 **validation** data validation.
* A special user "admin@admin.com" should have the role "Administrator" and should have full permissions to **edit** / **delete** **events** and **comments**.

PART 4 (continue from PART 3)

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:



## **Create New Event**

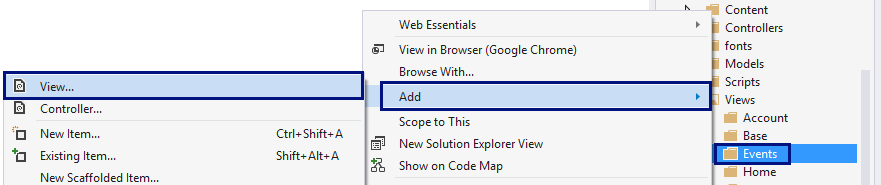
The next feature to be implemented in the Events management system is "**Create New Event**". This will require **creating a "New Event"** form (Razor view) + input **model** for the form + **controller** **action** to handle the submitted form data.

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:

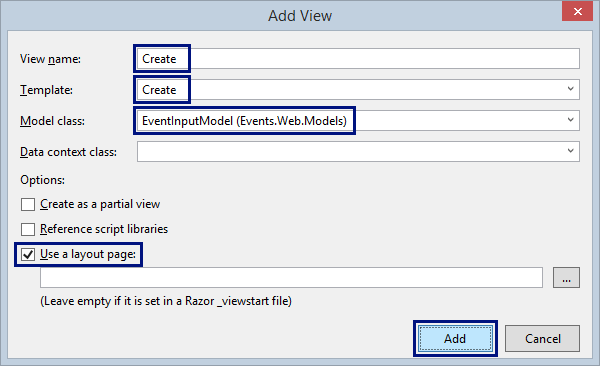


It is a good idea to attach **validation annotations** like [Required] and [MaxLength] for each property to simplify the validation of the form.

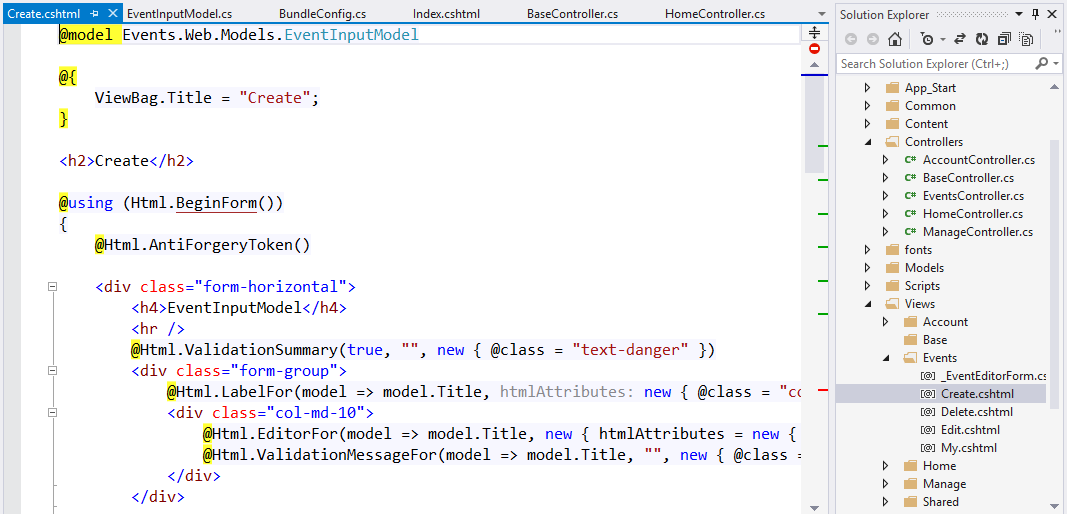
1. Next, let's **create the "New Event**" **form**. The easiest way to start is by using the **Razor view generator** in Visual Studio. Create a folder "**\Views\Events**". Right click on the "Events" folder and choose [Add] 🡪 [View…]:



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

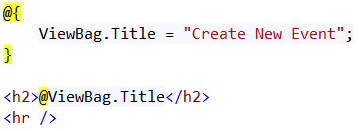


Visual Studio will **generate the "Create Event" form**:

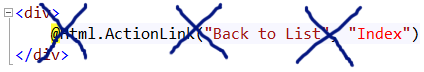


1. Now **customize the generated form**. Change several things:

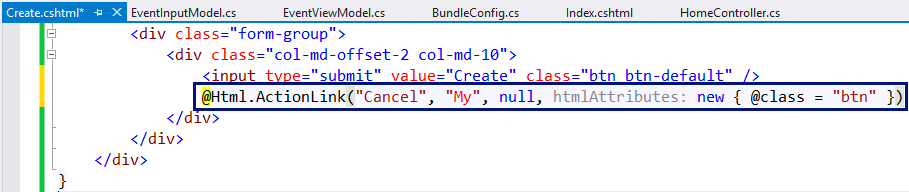
* Change the Title 🡪 "Create Event"



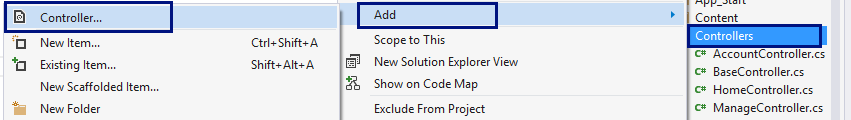
* Remove the "Back to List" link:

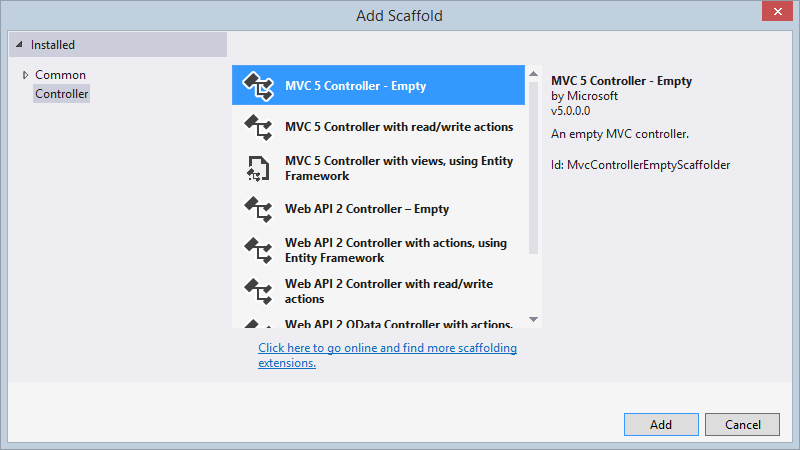


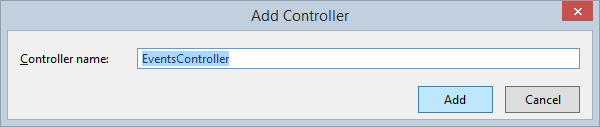
* Add **[Cancel]** link to "My Events", just after the **[Create]** button:



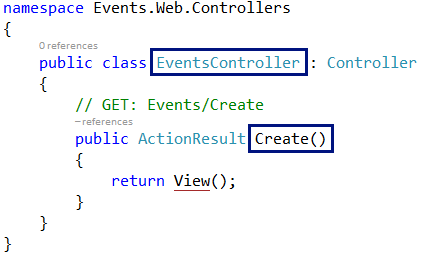
1. Create EventsController to handle the actions related to events: **create** / **edit** / **delete** / **list** events.



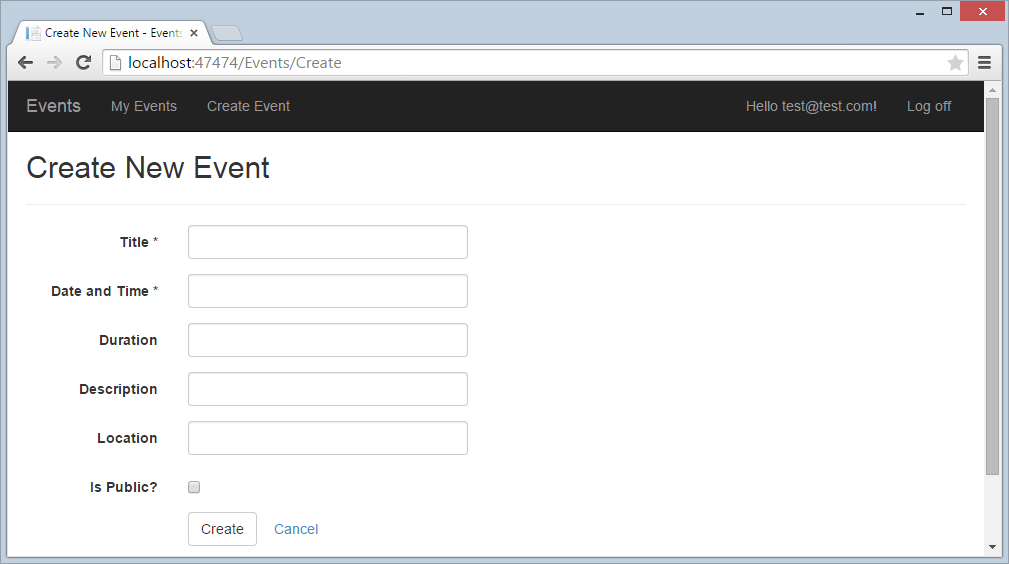




1. Add the **"Create" action** (HTTP GET) to display the "New Event" form:



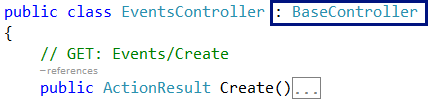
1. Build the project, run it and **test** the new functionality (**login** and click **[Create Event]**). The "New Event" form should be rendered in the Web browser:



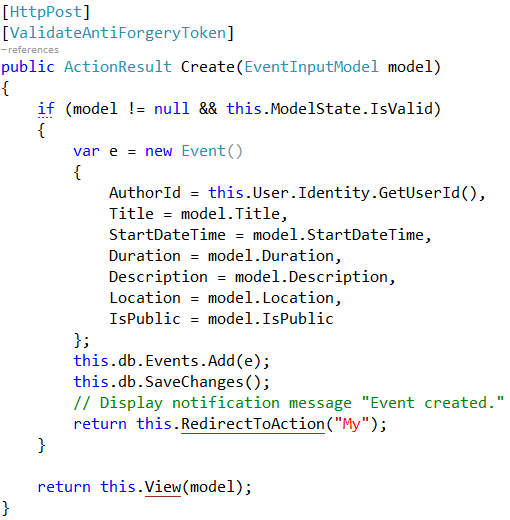
## Create New Event Logic

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.

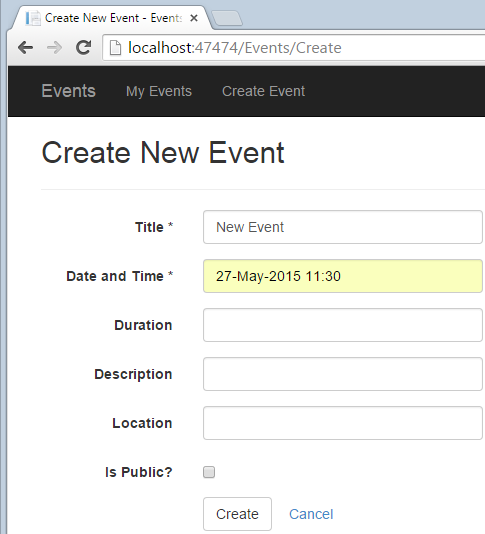
1. First, extend the BaseController to inherit the database context:



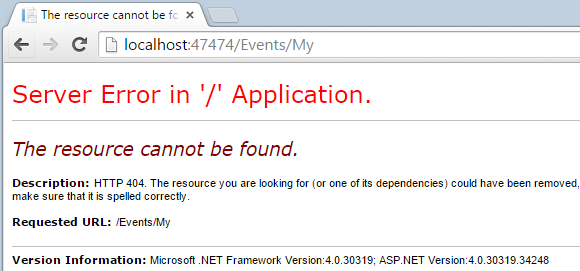
1. Next, write the action to handle POST \Events\Create:



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

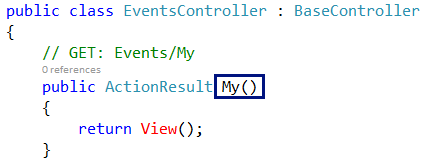


When the **form is submitted**, the **result** will be like this:

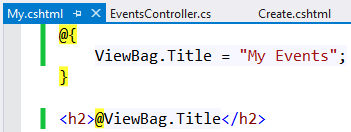


This is quite normal. The "My" action in the EventsController is not missing.

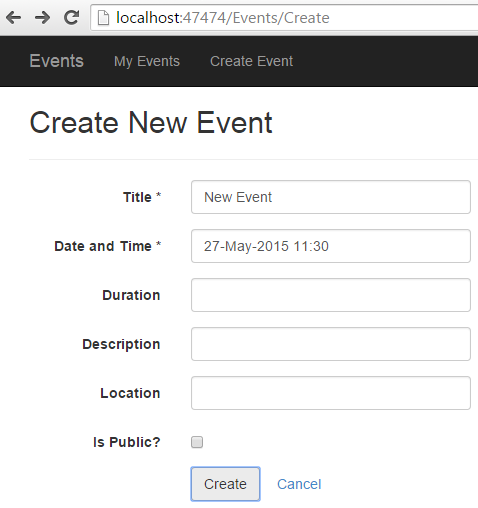
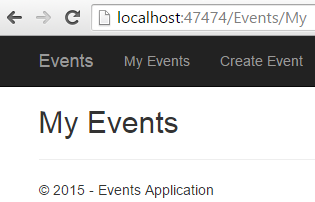
1. Let's define the action "My" and the view behind it in the EventsContoller:



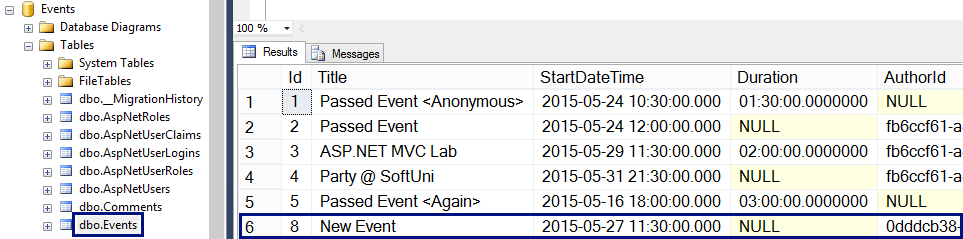
Create an empty view \Views\Events\My.cshtml (it will be implemented later):



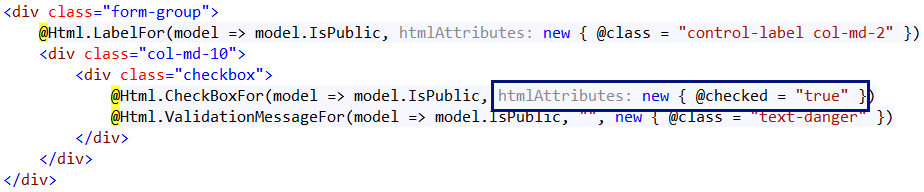
1. Now run and test the application again:

 🡪

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

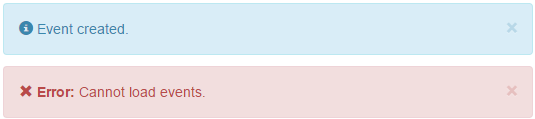


1. At the home page the new event is not listed. Why? The event is not public. Let's make the **"Is Public?" check box** in the "New Event" form **checked by default**:



## Notification System (Info / Error Messages)

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



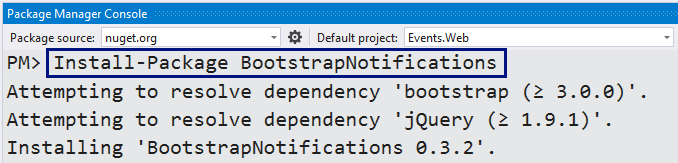
Typically, MVC controller actions that modify DB objects (e.g. EventsController.Create) work as follows:

* **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.

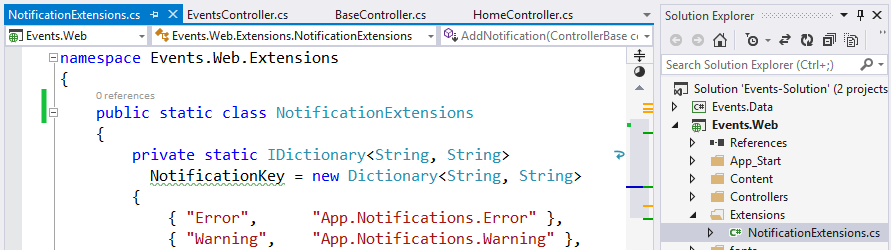
As the **"Post-Redirect-Get" pattern** says, in Web applications it is highly recommended to redirect to another page after executing an action that changes something at the server side: <http://en.wikipedia.org/wiki/Post/Redirect/Get>.

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:

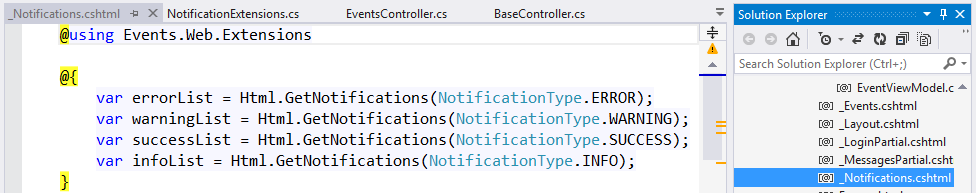
1. From the package management console in Visual Studio **install the NuGet package** BootstrapNotifications in the Events.Web project:



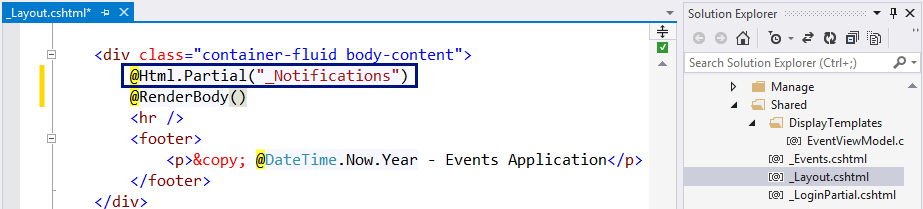
NuGet will add the class Extensions\NotificationExtensions.cs:



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



1. 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():



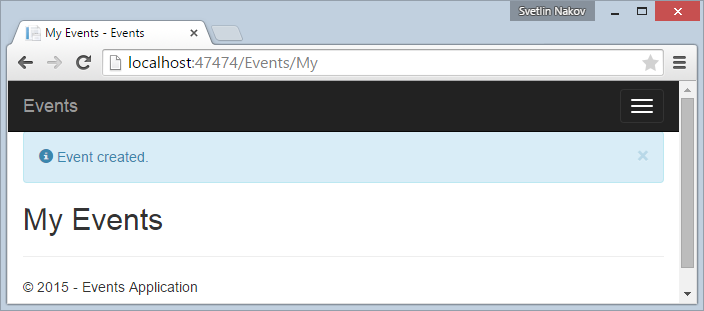
1. **Put notifications** in the controller actions after successful database change.

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.

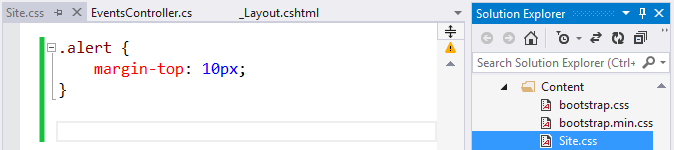
Add notification messages after db.SaveChanges() in the actions that change the database, followed by RedirectToAction(…):



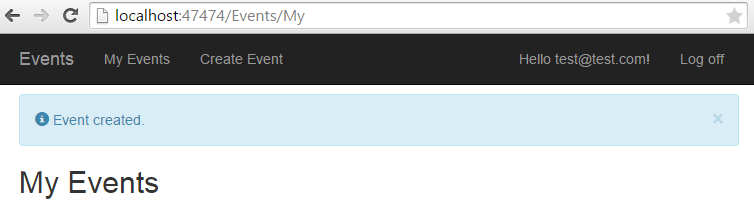
1. Run and **test the application**. Create an event to see the "**Event created.**" notification message:



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



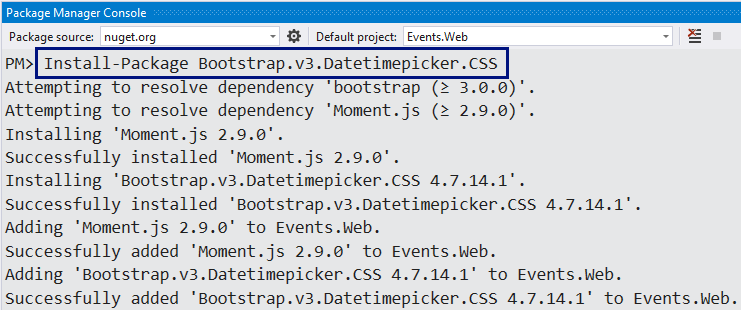
1. Save the changes, refresh the site with [Ctrl + F5] and create a new event to **test the changes**:



## 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 thedate and duration fields:

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

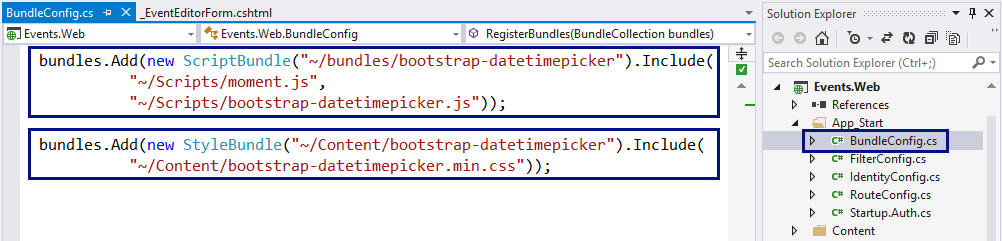


It will add the following files to your MVC project:

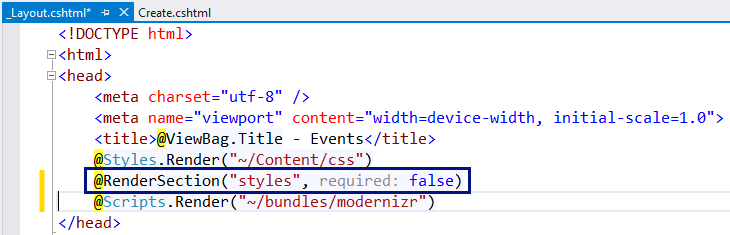
* \Scripts\bootstrap-datetimepicker.js
* \Scripts\moment.js
* \Content\bootstrap-datetimepicker.css

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

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

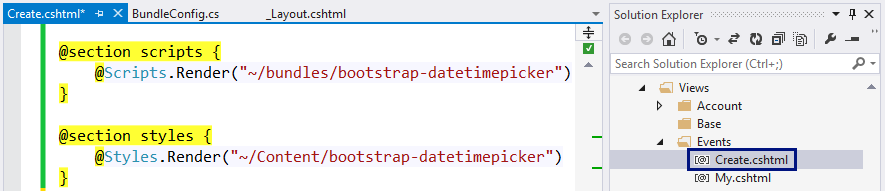


1. Create a **placeholder for the CSS bundles** in the \_Layout.cshtml:

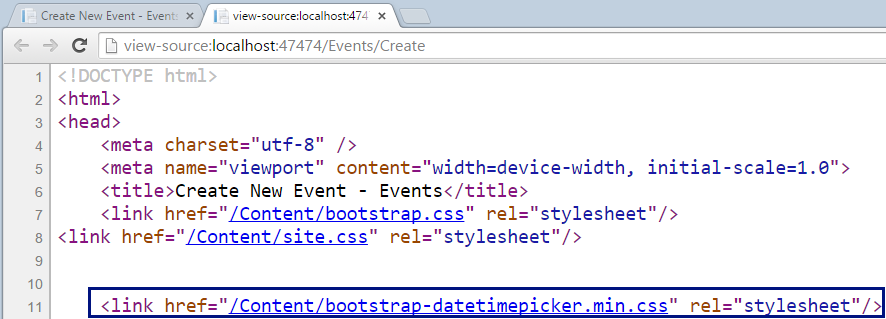


It will be used later to inject the date-time picker's CSS from the events editor form.

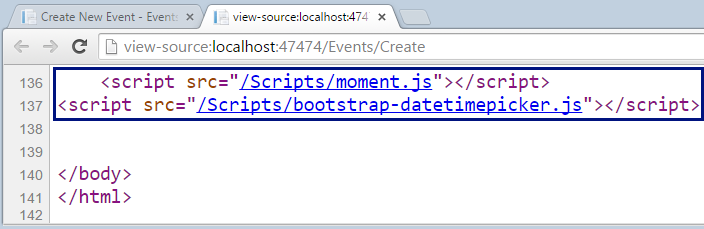
1. Add the date-time picker's scripts and CSS files in the "New Event" form, in \Views\Create.cshtml:



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



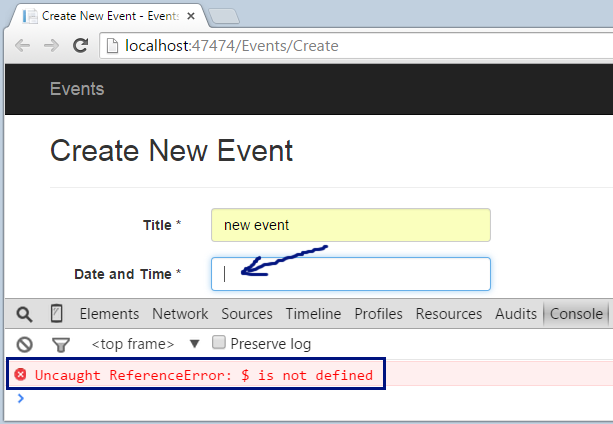
It will also inject the specified JavaScript from the bundles at the end of the page:



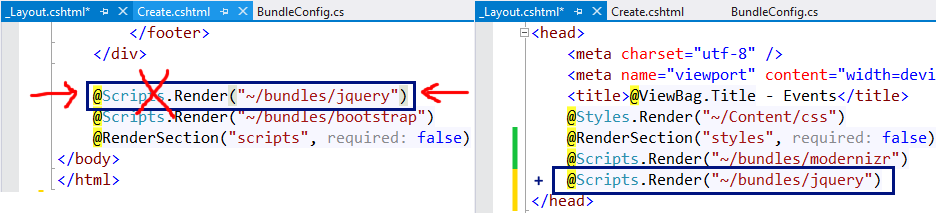
1. Now the site is ready to add the date-time picker in the "New Event" form. Let's try to attach the date-time picker for the field "StartDateTime" in \Views\Events\Create.cshtml:



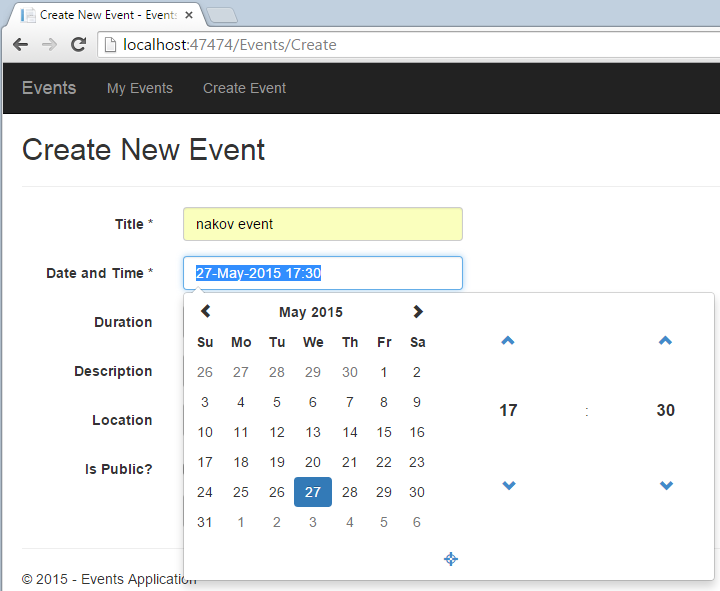
1. Now rebuild the project and run the code **to test the new functionality**. It does not work due to JavaScript error:



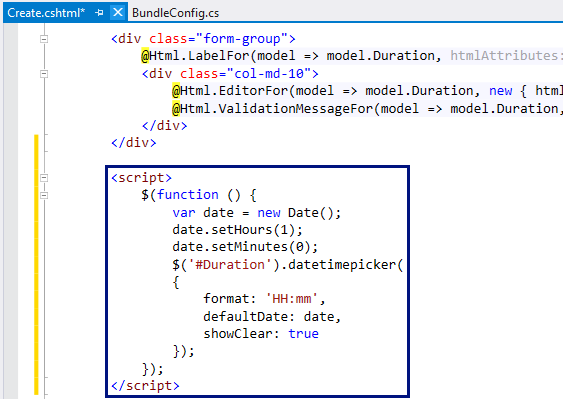
1. Seems like **jQuery is not loaded**. This is because the script for attaching the datetimepicker for the StartDateTime field uses jQuery, but the jQuery library loads later, at the end of the HTML document. This is easy to fix, just move the jQuery reference at the start of the HTML code in \_Layout.cshtml:



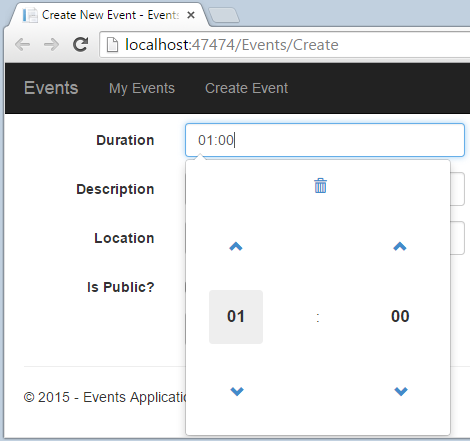
1. Now rebuild the project and run the code **to test again the new functionality**. It should now work correctly:



1. In similar way, add a time picker for the "Duration" field:



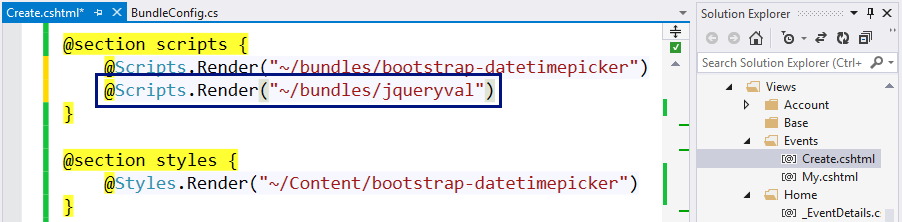
1. **Test** the new duration picker by starting the Web application:



## 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:

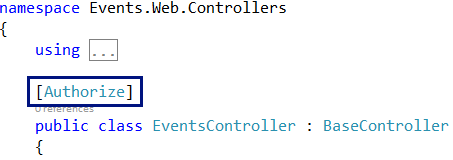


**Test the "New Event" form** to ensure the validation is now client side.

## List My Events

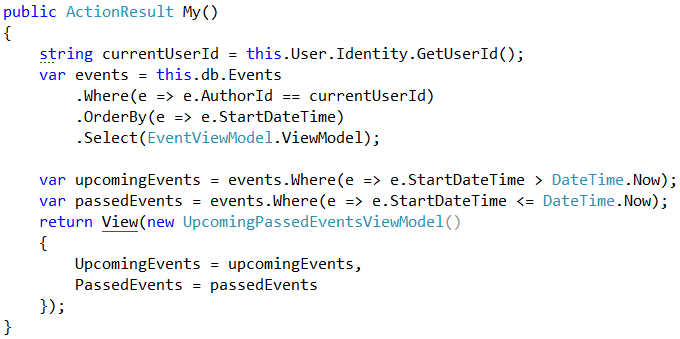
Now let's list current user's events at its personal events page: \Events\My.

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

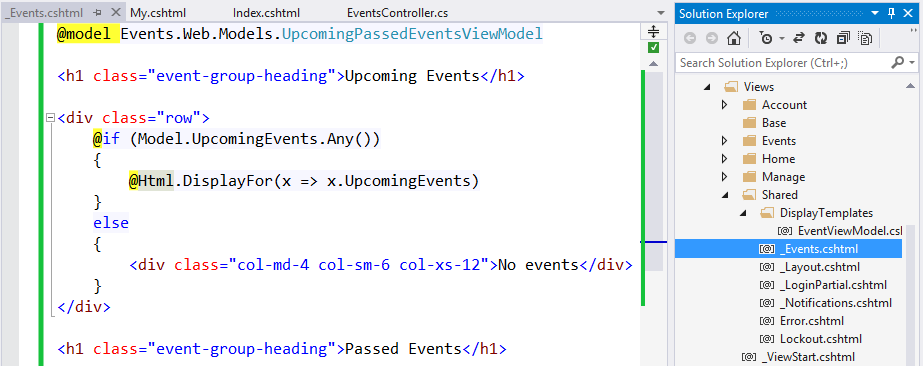


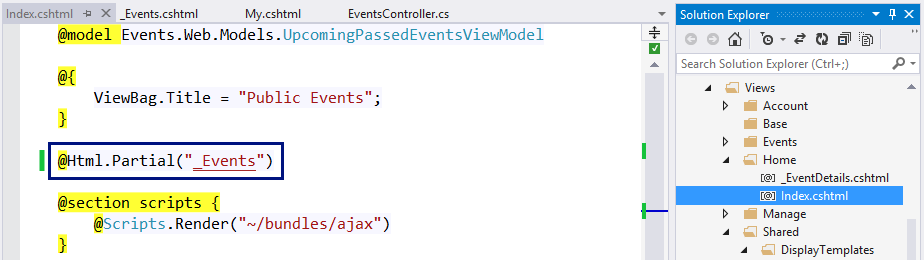
The [Authorize] attribute will redirect the anonymous users to the login form. It says that all controller actions of the EventsController can be accessed by logged-in users only.

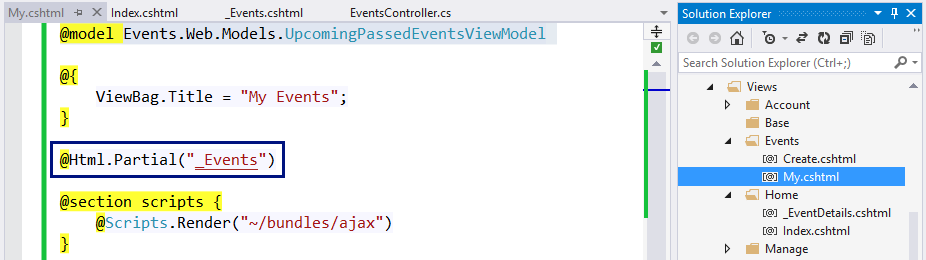
1. Next, add the HTTP GET controller action "My" in EventsController that will display current user's events:



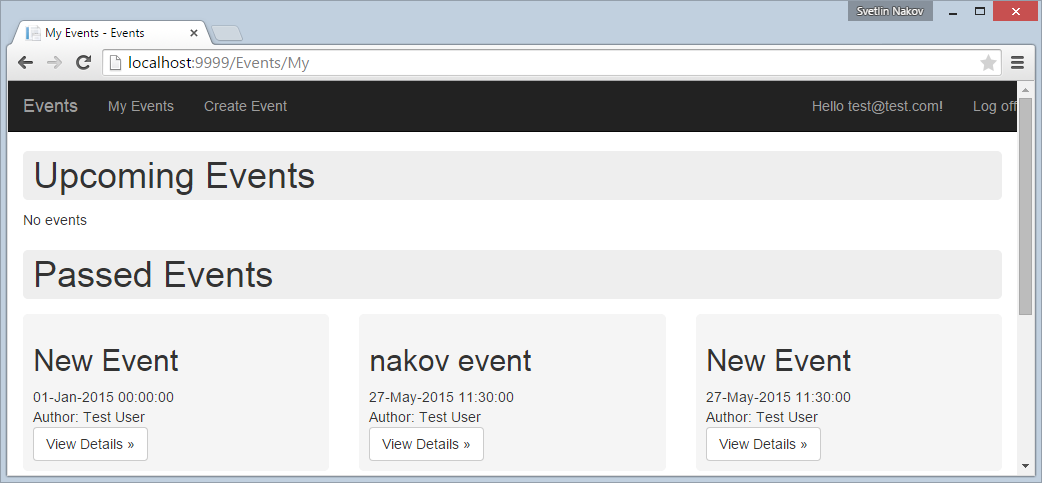
1. Finally, create the view My.cshtml behind the above action. It is essentially the same like the Index.cshtml view of the HomeController, right? Duplicating code is very bad practice, so let's **reuse the code**. First, extract a partial view \Views\Shared\\_Events.cshtml, then reference it from \Views\Events\My.cshtml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

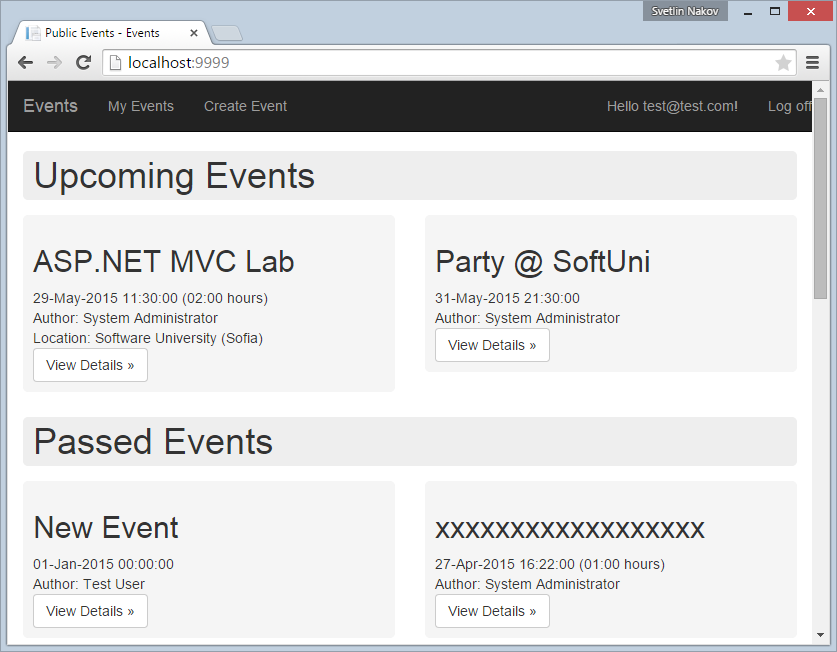




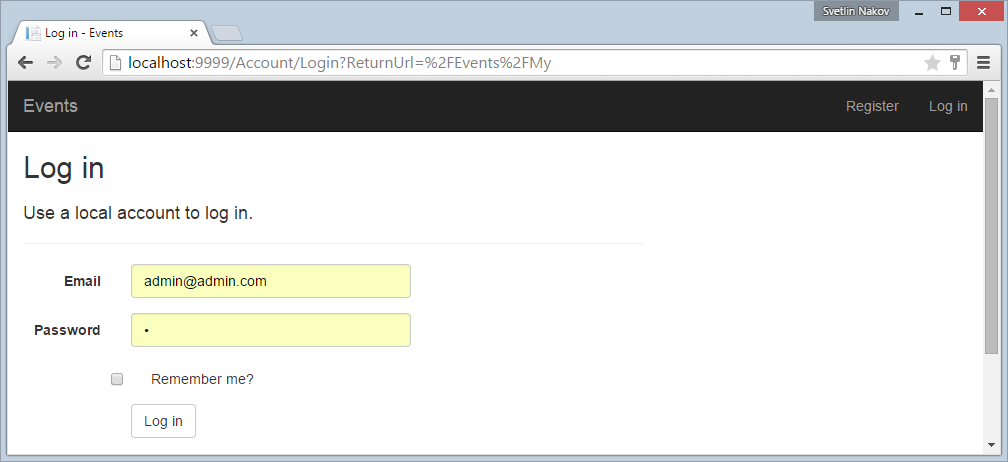


1. Test the new functionality "**My Events**", as well as the old functionality "**Home Page**":





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



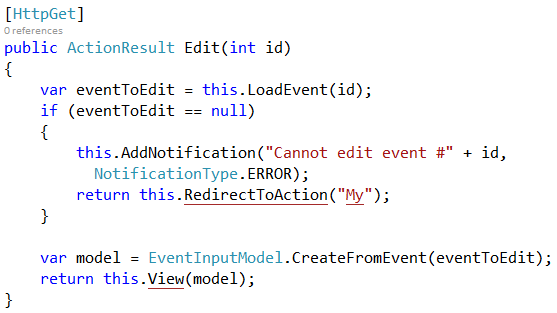
## Edit Existing Event Form

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.

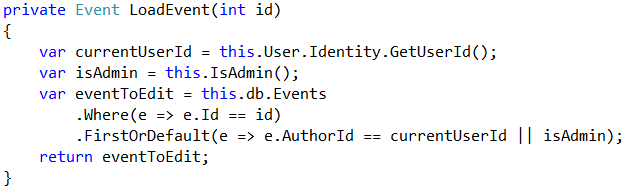
## Edit Existing Event Logic

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**":



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



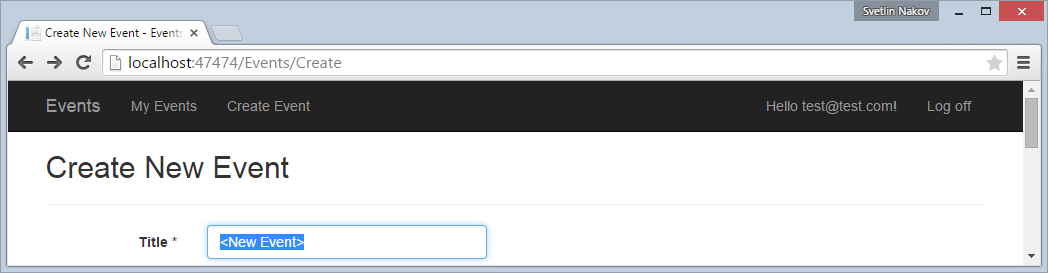
1. Write a HTTP **POST action "Edit"** to save the changes after submitting the event editing form. It should first **check the event ID** and show an error of case of non-existing event or missing permissions. Then it **checks for validation errors**. In case of validation errors, the same form is rendered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



1. Run and test the new functionality "Edit Event" by opening \Events\Edit\3.

## 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:

