MVC 5

For Beginners

The tactical guide book

Contents

1. Introduction 1
Introduction 1
Other titles by the author 1
MVC 5 - How to build a Membership Website 1
C# For Beginners: The tactical guidebook 2
Rights 2
About the author 2
What is ONE.NET 3
What is MVC 4
Why MVC? 6
Installation 7
Exercise 1-1: Your first MVC 5 application 8
Folder structure 12
Executing a solution 13
Two ways to send data to a view 15
ViewBag - a loosely typed model 17
Class - a strongly typed model 18
Exercise 1-2: Displaying values using the ViewBag 18
What's Next? 20
2. Models 21
What is a model? 21
Technologies used in this chapter 21
How to create a model 21
Class - A Strongly Typed Model 22

Model attributes 23
Exercise 2-1: Create a Model and add Attributes 25
Create the model 25
Create the controller 26
Create the Create view 27
Test the model 27
Too short name and empty email field 28
Validated name and faulty email address 28
Validated form fields 28
Create the Index view 28
What's Next? 29
3. Controllers 30
Introduction 30
Technologies used in this chapter 30
Routes 30
Route Mapping 31
Route Attributes 32
Exercise 3-1: Route Attributes 33
Creating the Details view 33
Test the new Details view 34
Add routing attributes 35
Actions and Parameters 36
Exercise 3-2: Action Parameters 38
Adding the Id property to the CustomerModel class 38
Adding the GetById action 39
Testing the GetById action 39

Action Accept Verbs 41
What's Next? 42
4. Razor Views 43
Introduction 43
Technologies used in this chapter 45
Razor Syntax 45
Cross-Site Scripting Attacks (XSS) 47
Exercise 4-1: Cross-Site Scripting 49
Adding the model 49
Adding the controller 49
Adding the Create view 50
Test the model 50
Code Expressions 51
Layout Views 52
HTML Helpers 54
DisplayNameFor 55
DisplayFor 56
Exercise 4-2: DisplayFor and DisplayNameFor 57
Adding the GetStaticBooks action method 57
Creating the StaticBooks view 57
ActionLink 58
Exercise 4-3: Add a menu item ActionLink 59
Adding the ActionLink 60
Form Helpers 61
BeginForm 62
AntiForgeryToken 62

HiddenFor 62
LabelFor 63
EditorFor 63
DropDownListFor 64
ValidationMessageFor 64
ValidationSummary 65
Exercise 4-4: Add a Create view and update the books list 66
Create the view 67
Update the books list 68
Add a new book with the Create view 68
Partial Views 69
How to create a partial view 70
How to call a partial view 71
Partial view without a view model 71
Exercise 4-5: Create and call a partial view 73
Change the Book model 73
Add the GetBestBook action method 73
Create the partial view 74
Create the partial view content 74
Add the @Html.Action call to the _Layout view 75
5. Introduction to Bootstrap 76
Introduction 76
Technologies used in this chapter 77
Scripts & CSS 78
Typography 80
Glyph Icons 80

Font Awesome 82
Grid Layout and Responsive Design 82
Combining multiple column definitions 89
Respond.js 90
Components 91
Buttons 91
Alert and Label 93
Tables 94
Basic table styling with Bootstrap 95
Condensed table 95
Hover effect 96
Striping effect 96
Colored rows 97
Navigation bar 98
The structure 99
6. CSS Styling 102
Introduction 102
Technologies used in this chapter 102
CSS Box Model 104
The Display and Visibility Properties 105
Property values 106
How to add a CSS style sheet 106
Adding styles to the style sheet 107
Style properties 108
Selectors 108
Simple selectors 108

Other selectors 110
More complex selectors 114
Pseudo classes 118
Cascading 119
Specificity 120
Inheritance 121
Developer Tools (F12) 122
Inspecting an element by clicking it in the web page 122
Inspecting an element by selecting it in the tree view 123
Inspecting the color of an element 123
Change style rules directly in the browser 123
Change HTML directly in the browser 123
7. JQuery/Ajax 124
Introduction 124
Technologies used in this chapter 125
$(document).ready() 125
Selectors 126
By Element Type 127
The .css function 128
The .html function 132
Looping with the .each function 133
By Id (#) 134
By Class Name 137
The .addClass function 138
The .removeClass function 139
The .hasClass function 140

The .toggleClass function 140
Class name selector 140
Multiple class name selectors 141
By Attribute 141
Adding/Removing Elements 143
Append 143
Prepend 144
Remove 144
Hide/Show 145
Events 146
.click() 146
.change() 147
.on() 148
Display data from a clicked table cell 149
Add a new table row 150
.hover() 151
Ajax 152
Exercise 7-1: Ajax and a partial view 153
The Month Model 154
Adding the partial view _MonthPartial 154
Alter the Index view 156
Adding the month list to the controller 157
Altering the Index action to render the model with the view 158
Adding the Ajax search functionality 159
Altering the Index action to handle asynchronous calls 162

1. Introduction

Introduction

This course is primarily aimed at developers who are new to MVC 5 and have prior experience with C#. The book presupposes that you have a solid base foundation in C# since it won't be explained in any detail. Even if you already have created a couple of MVC projects you might find the content in this book useful as a refresher. You might have worked in previous versions of Visual Studio and MVC and want a fast no-fluff way to get up to speed with MVC 5.

The examples in this book are presented using Visual Studio 2015 but they will work with 2013 Professional Update 4 as well. The free express version should do fine when you follow along and implement them yourself.

At the end of this chapter you will have an understanding of the design goals of a MVC application and have created your first MVC web application.

Other titles by the author

The author has written other books and produced video courses that you might find helpful.

MVC 5 - How to build a Membership Website

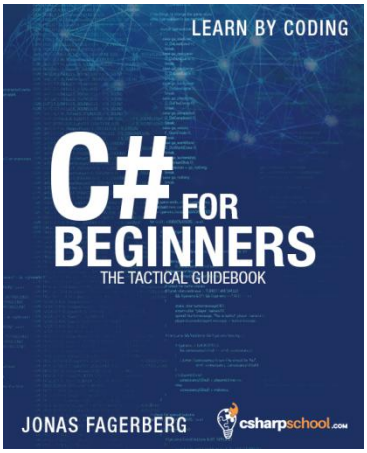
This is a comprihensive book describing how to create a membership web site; the *MVC 5 - How to build a Membership Website* video course is based on this book.

In this book you will learn how to build a membership website from scratch. You will create the database using Entity Framework code-first, scaffold an Administrator UI and build a front-end UI uisng HTML5, CSS3, Bootstrap, JavaScript, C# and MVC 5.
Prerequisites for this course are: a good knowledge of the C# language and basic knowledge of MVC 5, HTML5, CSS3, Bootstrap and JavaScript.

You can buy this book following this link:

C# For Beginners: The tactical guidebook

This book is for **YOU** if you are new to C# or want to brush up on your skills and like a **CHALLENGE**. This is not your run of the mill encyclopedic programming book; it is highly modularized, tactical and practical, meaning that you learn by reading theory and then implement targeted exercises building many applications.



You can buy **C# For Beginners: The tactical guidebook** at Amazon following this link: https://www.amazon.com/dp/B017OAFR8I

Rights

You can reach the author at: info@csharpschool.com.

About the author

Jonas started a company back in 1994 focusing on education in Microsoft Office and the Microsoft operating systems. While still studying at the university in 1995, he wrote his first book about Widows 95 as well as a number of course materials.

In the year 2000, after working as a Microsoft Office developer consultant for a couple of years, he wrote his second book about Visual Basic 6.0.

Between 2000 and 2004 he worked as a Microsoft instructor with two of the largest educational companies in Sweden. First teaching Visual Basic 6.0, and when Visual Basic.NET and C# were released he started teaching these languages as well as the .NET Framework. Teaching classes on all levels for beginner to advanced developers.

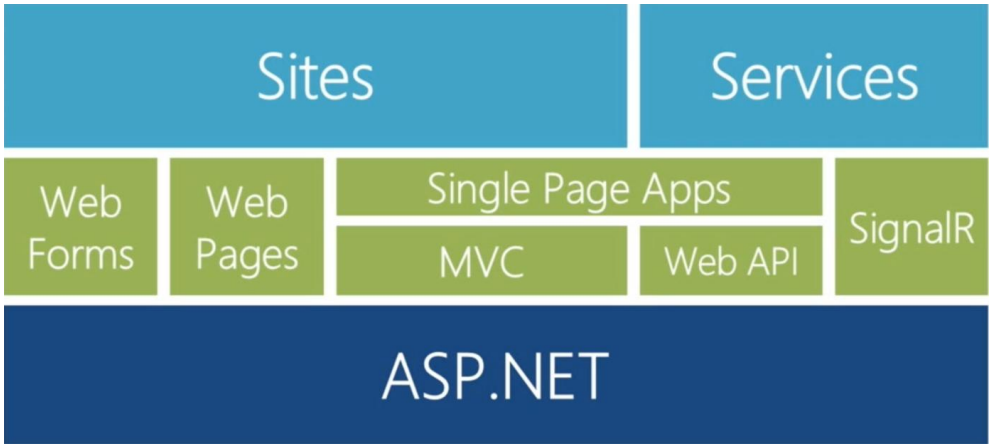
From the year 2005, Jonas shifted his career towards consulting once again, working hands on with the languages and framework he taught.

Jonas wrote his third book *C# programming* aimed at beginners to intermediate developers in 2013 and now in 2015 his fourth book *C# for beginners - The Tactical Guide* was published.

Jonas has alsso produced a 24h+ video course called *MVC 5 - How to build a Membership Website* showing in great detail how to build a membership website.

What is ONE.NET

ASP.NET is the common code executing as a foundation underneath everything else handling things like cashing, security and serving responses to the users.



The *Sites* box represent the content being delivered to browser pages that the users can interact with and the *Services* box represent data other than HTML which is sent to the browser, such as XML or JSon.

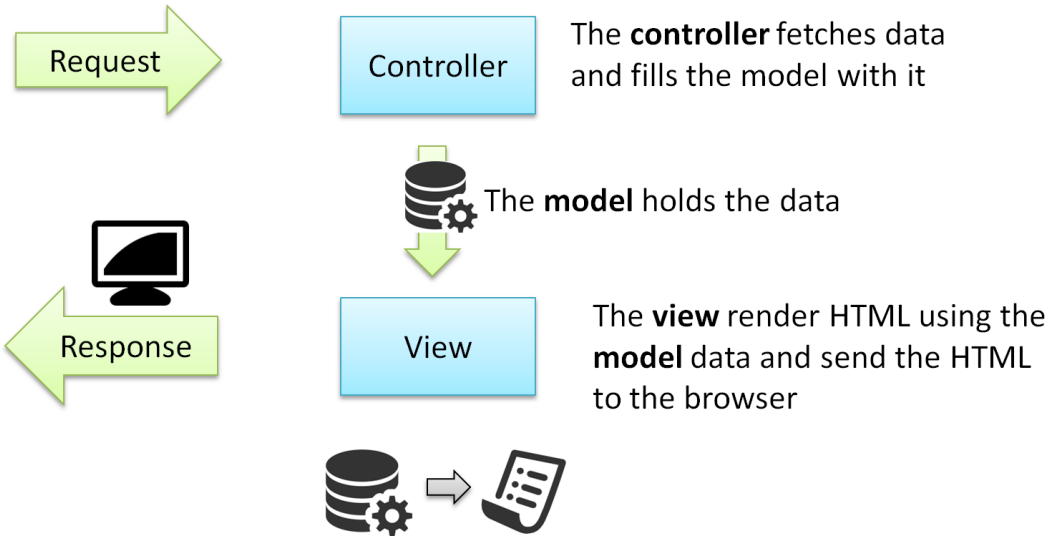
The green boxes in between the *ASP.NET* box and the *Sites* and *Services* boxes in the illustration are the glue that holds everything together. Here you find Web Forms, MVC, Web Pages, Web API and SignalR. No matter if you chose to use MVC, Web Forms or any of the other models they all run on the same foundation which is ASP.NET. This is referred to as *ONE.NET* or *ONE ASP.NET*.

The neat thing with *ONE.NET* is that the same foundation is used making it possible to utilize multiple models (MVC, Web Forms and others) in the same solution. You will even have the benefit of being able to utilize much of the same scaffolding such as security, Identity and Bootstrap across them all. You don't even have to include all of them from the start. In Visual Studio 2013 and beyond it is possible to add Web Forms, SignalR and Web API to the MVC solution when it is created or at a later time as needed. This mean that you can take your current skills and apply them throughout the *ONE.NET* solution.

What is MVC

MVC stands for *Model-View-Controller* and is a way to enforce separation of concerns when it comes to building a web application. The purpose of a *Model* is to transport data, the *View* is rendered as HTML to the user's browser and the *Controller* handles communication with the back-end data storage.

It cannot be stressed enough that the *Controller* has no knowledge of the *View* nor of how to display HTML, it is also clueless of what type of output is being generated by the *View*. The *Controller* uses a *Model* associated with the *View* to package the data and send it to the *View*, perhaps as JSon or XML. The *Controller* and *View* are completely separated from one another.



The image above describes the MVC pattern NOT your entire application. The *Controller* uses other classes to fulfill its tasks and should NOT contain all the application's C# code because that much code in a single *Controller* is unmanageable. An example of other classes that can be used are data layer classes which serves up and updates data in the back-end database.

Request: A user requests data, maybe by typing something into the browser, and the request gets routed to the *Controller*. This is the first big difference compared with for instance Web Forms (.aspx), PHP (.php) or ASP (.asp) which are file-based systems. In contrast to these types of systems where a URL (web page) often maps to a physical file, MVC will serve up the HTML on demand without basing it on a physical file stored in a file system. In reality the user is calling a method on a *Controller* class when using a MVC based application.

Controller: The tasks the *Controller* performs varies greatly. It could for instance be authenticating a user before fetching data, filling the model and sending it to the *View*, serving up an image or storing data in a data source. The common theme is that it handles data of some sort which then is packaged into a *Model* and sent to a *View*.

Compared with the "old school" way of accomplishing a task by calling an .aspx file through a URL to do something working with a method on a *Controller* is much more

lightweight and freeing because the *Controller's Action* method can make decisions based on logic.

Instead of having the headache of figuring out what the folder structure should look like and where the individual pages should be stored MVC only requires a good class structure.

Model: The *Model* is very easy to describe because it is simply a class which defines and holds the data being sent to and from the *View*. It is important to note that the *Model* shouldn't contain any logic only properties and attributes.

View: The *View's* task is to take the *Model* data and turn it into HTML with the help of C# and Razor syntax. A wiew is like a template which is filled in with the data sent to it. Later in the book you will see that it is possible to use C# code in a *View* but keep in mind that it should be used sparingly because it is not the *View's* task to handle logic, that is what the *Controller* is meant to do.

Response: When the *View* has been rendered the resulting HTML is served up to the calling browser(s).

Why MVC?

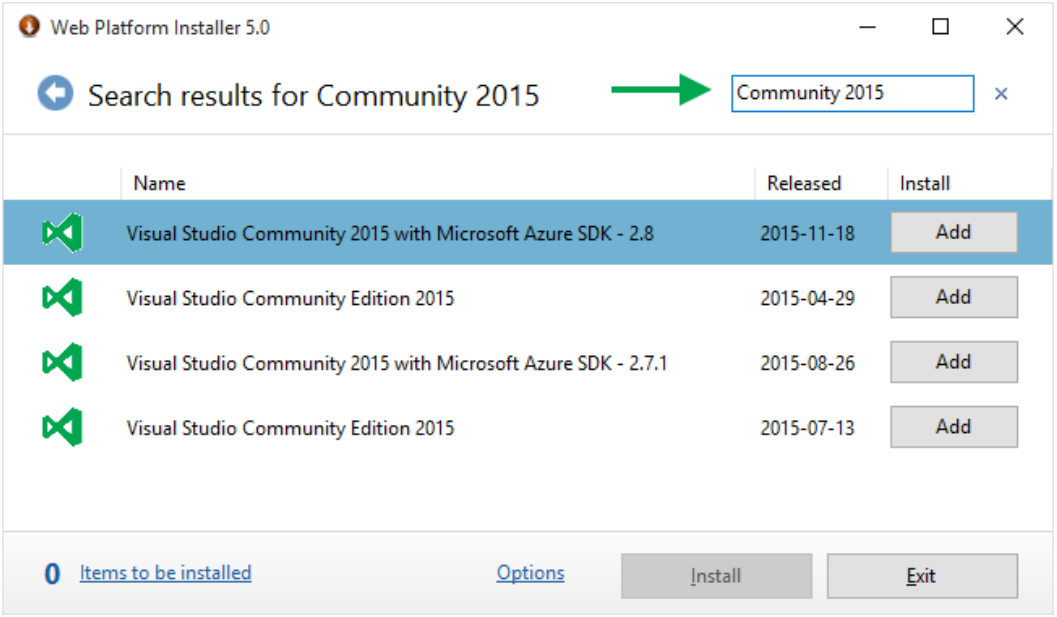
- MVC was designed to be testable with small units of code in isolation. This is a huge thing for the longevity of the code because specific unit tests can *automatically* pinpoint where the code break during updates or a changes.
- Browser differences are handled automatically for you.
- MVC gives you great application structure from beginning to end
  You no longer have to use huge cumbersome monolithic controls which often are hard to maintain. MVC lends itself to writing better code.
- MVC usually ends up having less code in the end because it is broken up into smaller units requiring less conditional logic and it often calls other services.
- With MVC there is initially a bit more to take in as opposed to learning the basics of Web Forms where you can just drag and drop. In the long run however MVC wins massively because there are so many potential problems that can occur in more advanced Web Forms solutions. One problem that frequently occur is that controls don't interact with one another as you would expect, leaving you with

the option to write your own server controls which is time consuming and can be a tricky task. Another Achilles heel for Web Forms can be handling membership security across controls and pages which is neatly handled in MVC.

- With Web Forms there are a lot of events which have to be orchestrated for each page and on top of that each page consists of multiple "pages" such as the master page, the actual .aspx page and all the intrinsic control pages. This can create a lot of headache and overhead.
- With Web Forms there is a lot of *ViewState* data being sent back and forth between the client and the server to keep track of the state of all the intrinsic controls. This can create massive overhead in traffic and slow down the time it takes to load the page.
- With Web Forms HTML id's and classes can be assigned very strange names and values which can be handled by an experienced Web Forms developer but can be a nightmare for beginners. This is designed in a more intuitive way and is easier to implement using MVC.

Installation

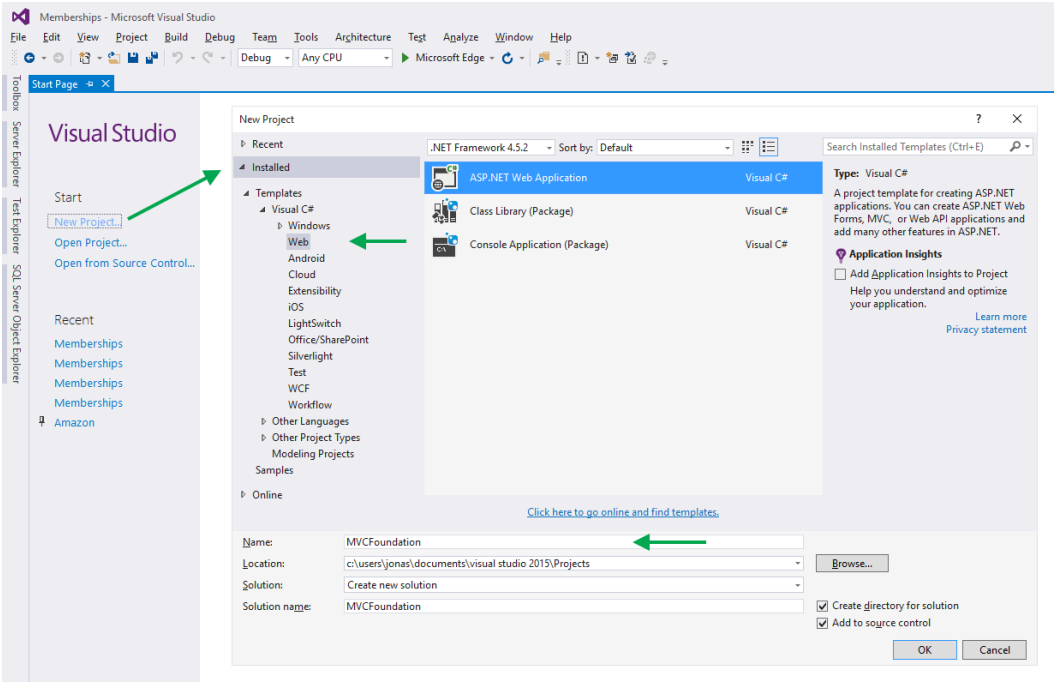
If you haven't installed Visual Studio 2015 you can do so by using the *Microsoft Web Platform Installer* which will help you install the latest components. Once it has been installed you can search for *Community 2015* and install the latest version of *Visual Studio Community 2015 with Microsoft Azure SDK* which contains everything you need when developing MVC applications. Core components which are installed are the *Visual Studio IDE*, *SQL Server LocalDB edition* for handling databases and *IIS Express* for handling HTTP requests.



Exercise 1-1: Your first MVC 5 application

In this exercise you will create an MVC application from scratch and look at what was automatically generated in the solution.

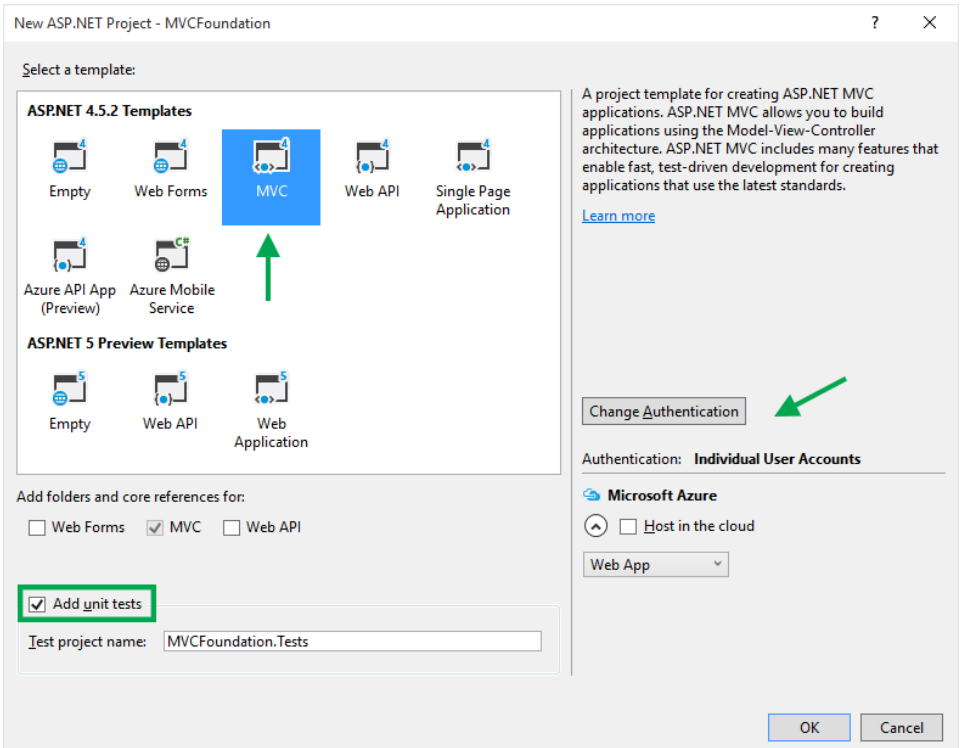
The easiest way to create a new project is to click the *New Project* link or select **File-New-Project** in the main menu. This opens the *New Project* dialog in which you choose what project type to create. One neat feature when creating projects is that the necessary libraries are pulled in using *NuGet* packages which makes them very easy to update.



1. Click the **New Project** link in Visual Studio.
2. Expand the **Visual C#** node and click on the **Web** node in the *New Project* dialog.
3. Make sure that the **ASP.NET Web Application** template is selected in the list.
4. Give the project a name; use *MVCFoundation* as project and solution name.
5. You can change the destination folder on your hard drive by clicking on the **Browse** button.
6. Click the **OK** button to create the project and its corresponding solution.
7. The *ASP.NET Project* dialog will appear where you choose the type of project you want to create. Select **MVC** in the template list (see image below).
   Note that this choice does not limit you to use only MVC. With *One.NET* the idea is that you can add other functionality later as needed using *NuGet* packages.
   You can even do it up front in this dialog by checking the appropriate checkboxes below the template list.
8. Although this isn't a course on Unit Testing I will show you how to add a separate test project by checking the *Add unit tests* checkbox. This will add a second

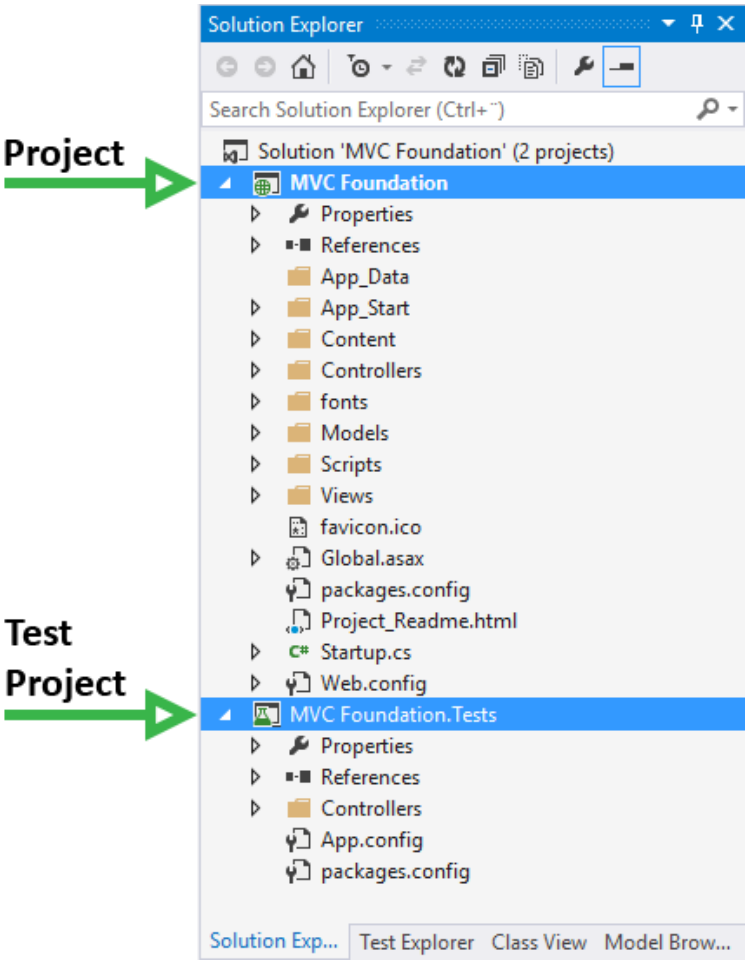
project to the solution to which you can add Unit tests for your application. Unit testing is a fantastic way of verifying the functionality of your application.

9. The default authentication model for MVC applications is *Individual User Accounts* which basically mean that you collect usernames, email addresses and in some scenarios password hashes in a database against which the application can authenticate user logins. There are other authentication models you can implement for other types of applications. Click on the **Change Authentication** button if you want to select another authentication model.
10. In this book you will not be using Azure cloud hosting to host your application. If you decide you want to host an application using Azure in the future you can enable it by checking the **Host in the cloud** checkbox and provide the necessary account information.



11. Click the **OK** button after you have selected the **MVC** template and checked the **Add unit tests** checkbox.

12. When the solution and projects have been created you can see them in the *Solution Explorer* (usually displayed at the right hand side of the Visual Studio IDE).
    If the *Solution Explorer* is hidden you can show it by selecting **View-Solution Explorer** in the main menu or pressing **Ctrl+W, S** on the keyboard.



13. Now that Visual Studio has finished creating the solution you can go ahead and run it by clicking the "Play" button in the toolbar or by pressing **F5** on the keyboard.
14. Explore the application by clicking on the links in the menu.
15. Close the browser to end the application.

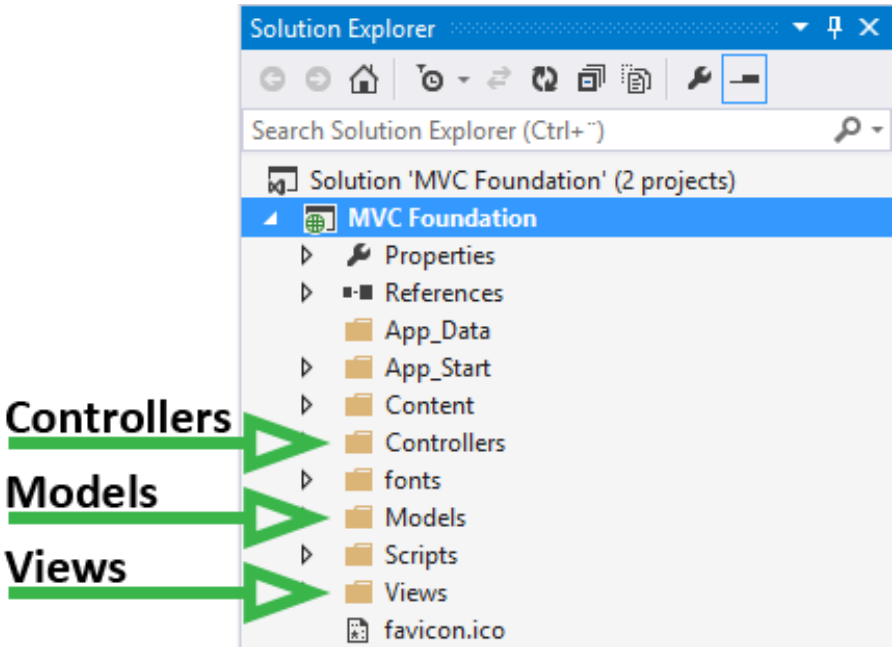
Folder structure

There is a naming convention for the folder structure in MVC projects which makes it easy to find the files you are looking for. You can be certain that all *controller* classes are located in the **Controllers** folder, all the *model* classes are in the **Models** folder and all the *views* are in the **Views** folder.

A *controller* class must have a ***Controller*** suffix. If the controller's name is *Home* then the class name has to be *Home**Controller***.

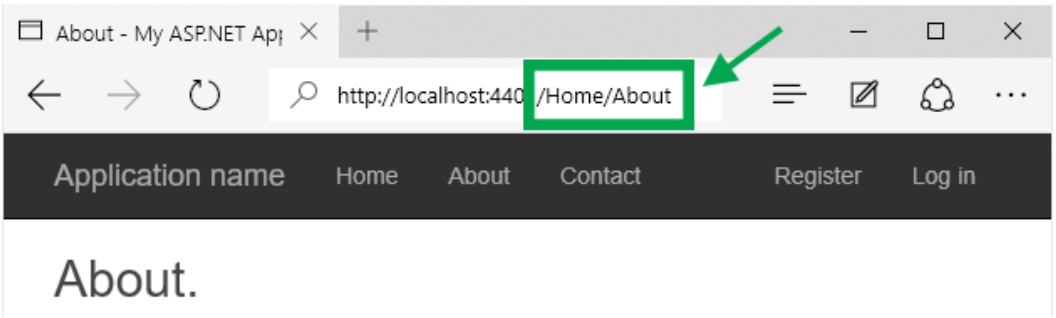
The views inside the **Views** folder adheres to a second naming convention stating that all views associated with a specific *controller* must be placed inside a folder with that *controller's* name (excluding the ***Controller*** suffix which only the *controller* class should have). Each view is associated with an *action* method in the view's corresponding *controller*.

These naming conventions hold true as long as you don't change the default conventions.



Executing a solution

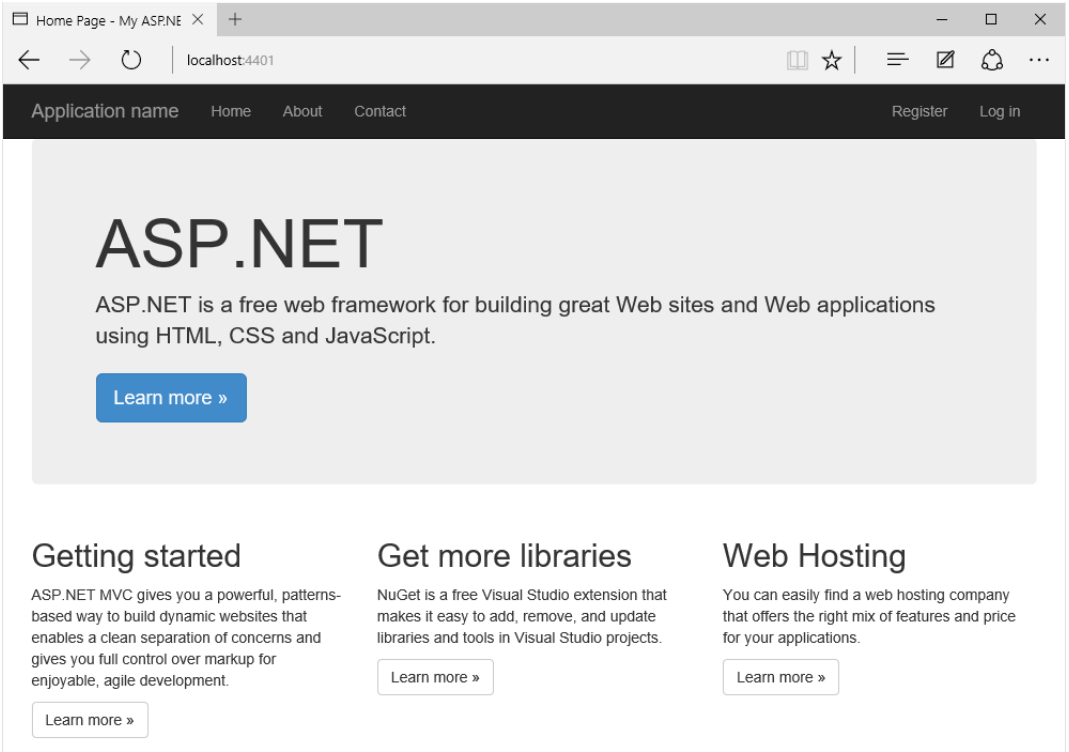
To execute a solution you press **F5** to start it with debugging or **Ctrl+F5** to start without debugging. Note that no file extension is present in the URL. This type of friendly URL can be used because no traditional web pages are involved in MVC projects. Web pages are rendered using HTML, Razor and C# code on the server side, and JavaScript and JQuery can be used to call the server from the client browser when the need arises.

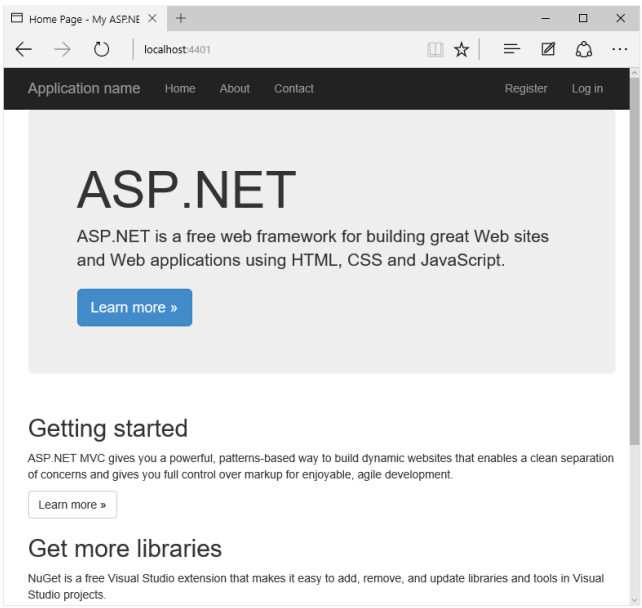


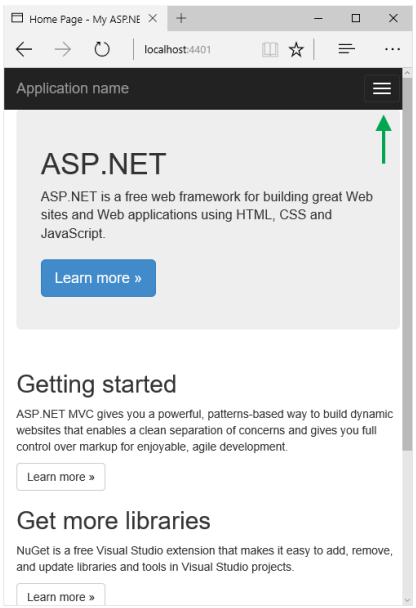
A really nice default web site is created for you complete with a working menu bar containing links that you can click on to navigate the site. Another neat thing is that responsive design is available right out of the box. This mean that resizing the browser or viewing the site on another type of device automatically will change appearance and scale the content to fit the current device screen. This has all been made possible in a straightforward and easy way by including Bootstrap as part of all new MVC projects by default. The Bootstrap grid layout makes it really easy to position the elements on the page in an esthetically pleasing way.

Avoid assigning specific pixel widths or heights to page elements because that can mess up the flow of the responsive design.

The images below shows the same site displayed resized for different device screen sizes. Note that the menu appearance changes when the site is viewed on small devices.







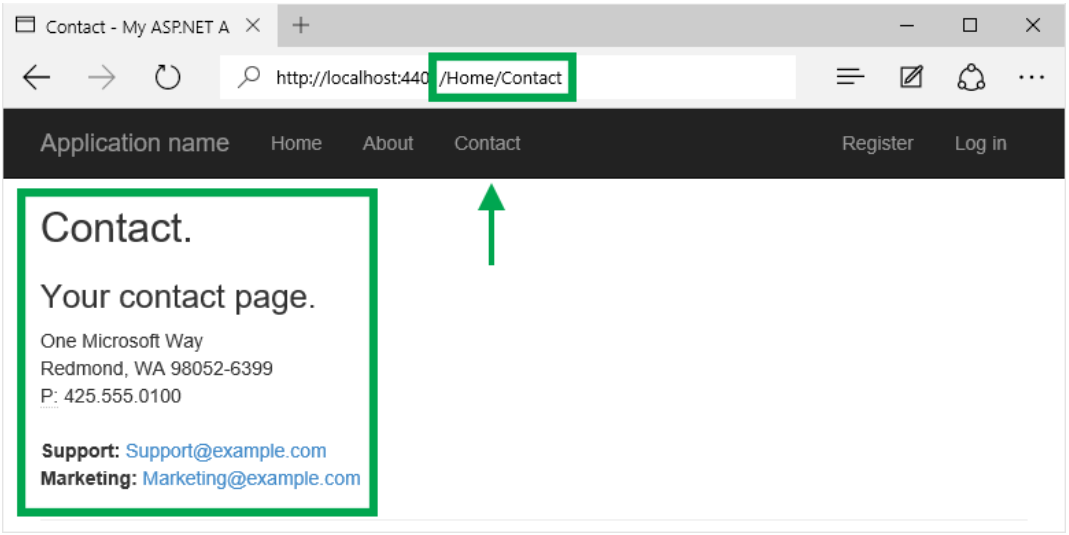
Looking at the HTML code that was rendered when you navigated to the **About** view you can see that it consist of very clean HTML 5 code. The *DOCTYPE* tag is an immediate giveaway that it is an HTML 5 page. The automatic addition of the character set tag (*charset*) is a good thing since it can prevent some strange and subtle cross-site scripting vulnerabilities. The *modernizr* JavaScript library helps make the content look nice in older non-HTML 5 browsers like Internet Explorer 6. You can view the rendered HTML by right clicking in the browser and select **View source** in the context menu; depending on the browser the source may be displayed in a separate window or in the **F12** debug tool.

Later you will have a closer look at how code like this is rendered and how you can modify it, but for now let's close the browser and the source window you opened and return to Visual Studio to have a closer look at the MVC project.

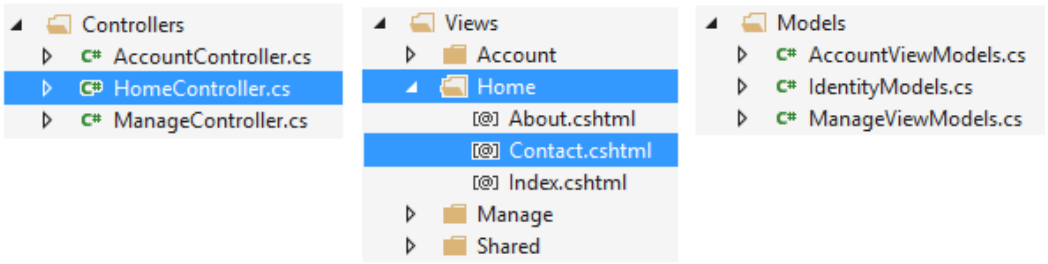
Two ways to send data to a view

Imagine if you will that you have been assigned the task to modify the data displayed in the *Contact* section of the application. When the application's home page is visible in the browser and you click the **Contact** link in the menu toolbar the *Contact* page should be

displayed, but how do MVC know what data to serve up and which Razor view to render?



As you can see in the image above the URL is */Home/Contact* which mean that the **Contact** action method was called on the **Home** controller. What does that mean? If you look at the project structure in Visual Studio there are three key folders (**Models**, **Views** and **Controllers**) as mentioned in the Folder structure section earlier in this chapter.

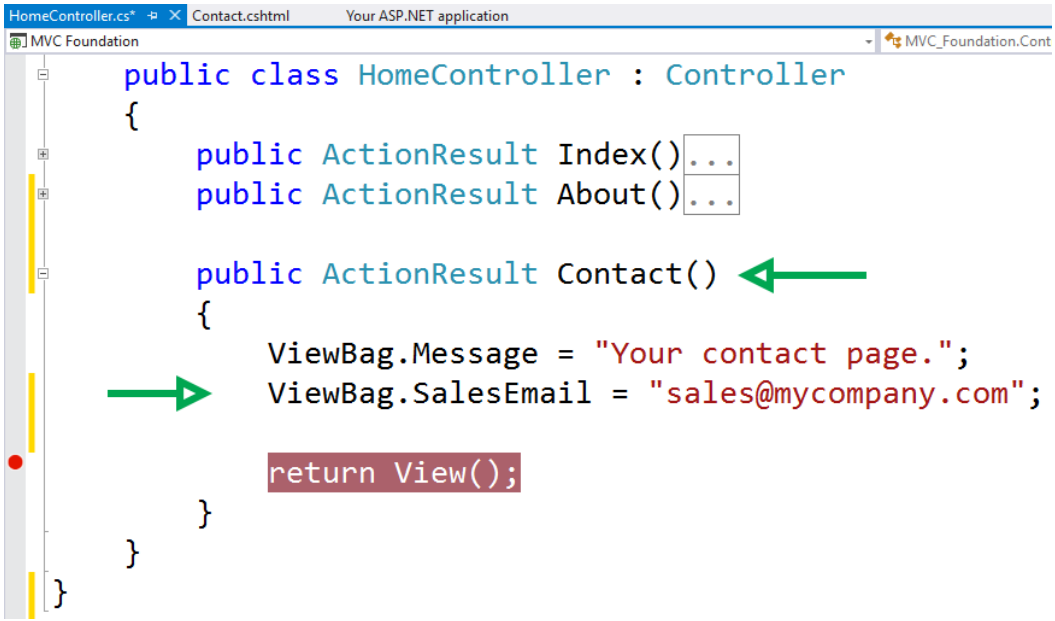


It turns out that the *Home* part of the URL correspond to a C# class called ***Home-Controller*** located in the **Controllers** folder. Inside that class there is an *action* method called **Contact**. If you place a breakpoint inside this action and debug the application you will end up inside this very action when you click the **Contact** link in the menu bar.

ViewBag - a loosely typed model

By default the **Contact** action method is empty apart from a **ViewBag** statement and a call to the **View** method. The first way to pass data to a view is to store it inside the **ViewBag**, which is exactly what the name implies, a "bag" in which you can place values needed in the view. You can pass any type of data in the **ViewBag** by simply adding a property name to it and assigning a value to that property. Let's say that you want to add an email address to your sales department to the *Contact* page (view). One way to achieve this is to store the email address in a property called **SalesEmail** in the **ViewBag**. Because the **ViewBag** is a *dynamic* object you can add any type of data which then can be rendered in the view.

```
ViewBag.SalesEmail = "sales@mycompany.com";
```

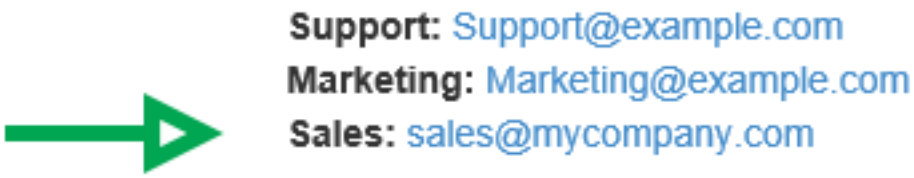


Now that you have added a value to the **ViewBag** it is time to change the Razor view to display the email address on the rendered page. To achieve this you first need to figure out where that view is located. It turns out that the MVC naming convention states that a controller action must have a corresponding razor view named exactly as the action; it should be located in a sub-folder to the **Views** folder with the same name as the controller (without the ***Controller*** suffix).

In this case you are calling the **Contact** action by clicking on the **Contact** link in the menu bar. This tells MVC to locate an action method called **Contact** in the **Home** controller. When the **Contact** action has finished working and is ready to send the data to the view it will look inside the **Views** folder for a sub-folder named **Home** and inside that folder locate the view named *Contact.cshtml* which is the Razor view template used to render this particular action.

To display the **SalesEmail** value you added to the **ViewBag** on the rendered page you need to open the Razor view named *Contact.cshtml* and modify it to display the email address. Note that you add the property twice inside the anchor tag; first you add it to the **href** to create a **mailto:** link, which will make it possible to activate the email when clicking on the link, then you add it as link text in the anchor tag.

```
<address>
    <strong>Support:</strong><a href="mailto:Support@example.com">
        Support@example.com</a><br />
    <strong>Marketing:</strong><a href="mailto:Marketing@example.com">
        Marketing@example.com</a><br />
    <strong>Sales:</strong><a href="mailto:@ViewBag.SalesEmail">
        @ViewBag.SalesEmail</a>
</address>
```



Class - a strongly typed model

The second way, and often the preferred way, to send data to a view is by using a strongly typed model class which defines the type of data that can be sent to and displayed in the view. We will talk more in-depth about models in the next chapter.

Exercise 1-2: Displaying values using the ViewBag

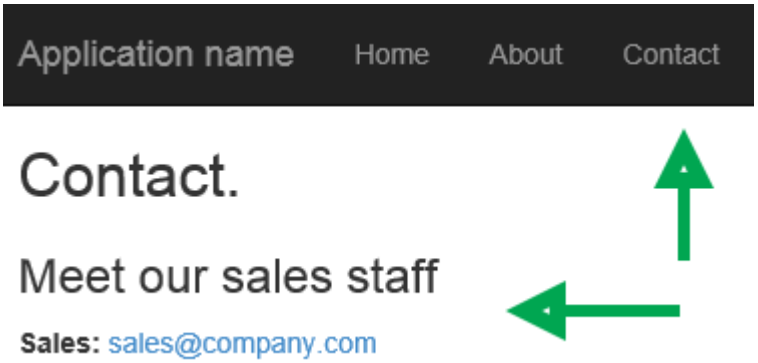
Now that you have successfully created the *MVCFoundation* solution and seen that it works right out of the box it's time to change some of the output. In this exercise you will

change the *Contacts* page to display a sales email and remove the physical address section.

1. Open the *MVCFoundation* solution you created in the previous exercise.
2. Expand the **Controllers** folder in the Solution Explorer and click on *HomeController.cs*. This will open the **HomeController** class.
3. Locate the action method called **Contact**. This method will be called when the user clicks on the **Contact** link in the application's menu bar.
4. Run the application and click on the **Contact** link in the application's menu bar and note the text "Your contact page" displayed in big bold letters. Next you will change this text by modifying the text in the **Contact** action method located in the **HomeController** class.
5. End the application by closing the browser or by clicking on the "Stop" button in Visual Studio.
6. Change the existing **ViewBag** text from "Your contact page" to "Meet our sales staff" and start the application again.
7. Click on the **Contact** link to verify that the change you made is reflected on the page.
8. Close the browser.
9. Next you will delete some content and display an email address on the **Contact** page by using the **ViewBag**. Add the following code to an empty line below the already existing **ViewBag** statement and above the **return** statement.
```
ViewBag.SalesEmail = "sales@company.com";
```
10. Now you will modify the view displaying the contact information. Expand the **Views** folder in the Solution Explorer and click on the *Contact.cshtml* view. This is the Razor view template which is used to render the HTML sent to the user's browser when the user navigates to the *Contact* page.
11. Delete the top **<address> … </address>** block containing the physical address.
12. Change the content of the remaining **<address> … </address>** block to the code below. Note that you are injecting your **SalesEmail** property from the **ViewBag** into an anchor tag (link) in two places. The first is in the **href** which is the link/URL the browser will follow when the user clicks the link. The second will display the email address as link text for the user to click on.
```
<address>
```

```
    <strong>Sales: </strong><a href="mailto:@ViewBag.SalesEmail">
        @ViewBag.SalesEmail</a><br />
</address>
```

13. Start the application and click on the *Contact* link in the application menu bar. Note the changes to the text on the page.



14. Close the application.

What's Next?

In this chapter you got the 10.000 feet overview of MVC, what the M-V-C stands for and are represented by. You also learned how to install Visual Studio and create your first solution. Set aside time to do the exercises NOW, don't let your brain tell you that you can do it tomorrow or next week or in a month. Plant your feet firmly on the ground and challenge your brain, tell it that you are determined to succeed at this new thing called MVC and you will do the exercises NOW. Both you and I know that "doing it later" means not doing it at all, so bet on yourself and your future and do the exercises NOW. When you have finished the exercises I want you to keep the momentum going and dive right into the wonderful and exiting world of *Models*.

2. Models

What is a model?

A model is *just* a class.

With that said, you often want to decorate your model properties with attributes to help the Razor view engine to better automate the creation of the HTML elements representing the property data types. Attributes can be useful for generating correct input controls and constraining and validating input data.

Technologies used in this chapter

- **C#** - To write the code in the model class.
- **Razor** - To add and use the model in the view.

How to create a model

Models should be placed in the **Models** folder if you follow the default convention, this is however a convention that is often broken in large applications because it can be cumbersome to have all models in the same folder. Another case might be that you are using an already existing web service or class library which contains the models you need.

To add a model class to the **Models** folder you:

1. Right click on the **Models** folder.
2. Select **Add-Class** in the context menu.
3. Give the model (class) an appropriate name
   One popular convention is to add the suffix *Model* to the end of the model class names to distinguish them from other classes.
4. When you click the **OK** button or hit **Enter** on the keyboard the model class is added to the **Models** folder and is opened in the Visual Studio IDE

Tip: *A shortcut for adding properties to a class is to use one of the built in code snippets. Simply write **prop** and hit **Tab** twice to add a property template to the class.*

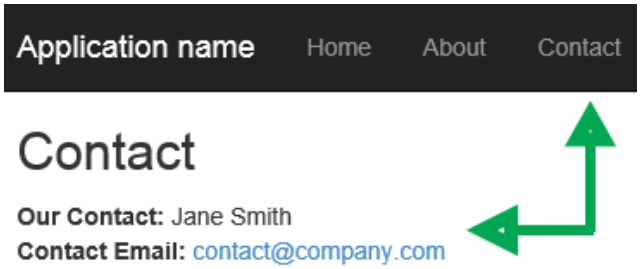
Class - A Strongly Typed Model

The preferred way to send data to a view is to use a strongly typed model class which defines what data can be sent to and displayed in the view from the server. This brings the third folder mentioned earlier into play, the **Models** folder.

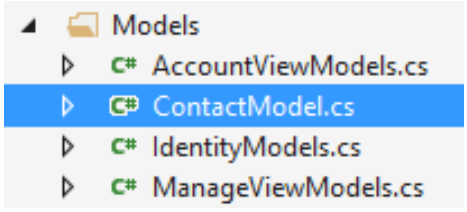
For smaller projects you can place all the **Model** classes inside this folder organizing them in sub-folders. For larger solutions you probably want to separate the models into one or more class library projects and reference those projects from the MVC project.

Model is a fancier way of saying class because that is essentially what a model is. Just as you create classes to define your objects in an application you use classes to define models for the views. One important distinction between object classes and model classes is that a model should only transport data and not contain any logic. Because as mentioned earlier it is the controller's task to perform the logic and provide the views with data through its action methods.

Let's say that you want to display a header with the text "Contact" followed by the name of the person to contact and a generic contact email where the customers can reach you.



To achieve this you could create a new model class called **ContactModel** in the project's **Models** folder and add two properties called **ContactEmail** and **ContactName** to it.



```
public class ContactModel
{
    public string ContactEmail { get; set; }
    public string ContactName { get; set; }
}
```

After creating the **ContactModel** class you will use it in the **Contact** action in the **Home** controller by creating an instance of it and passing it to the view.

```
public ActionResult Contact()
{
    var contact = new ContactModel
    {
        ContactEmail = "contact@company.com",
        ContactName = "Jane Smith"
    };

    return View(contact);
}
```

Finally you will alter the Razor view to enable it to receive an instance of the **Contact-Model** class as its model, you do this by adding a ***@model*** directive at the beginning of the view. You can then access the data in the passed-in object through this directive. Note however that you use the name *Model* with a capital letter "M" to access the data when adding it to HTML markup.

```
@model MembershipSite.Models.ContactModel

<h2>Contact</h2>

<address>
    <strong>Our contact: </strong>@Model.ContactName<br />
    <strong>Contact email: </strong>
        <a href="mailto:@Model.ContactEmail">@Model.ContactEmail</a>
</address>
```

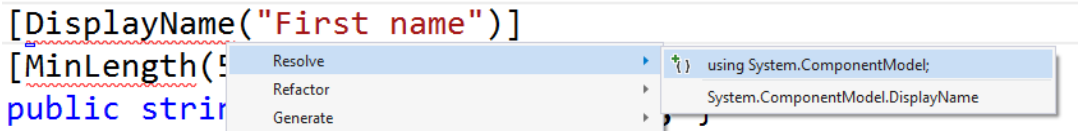
Model attributes

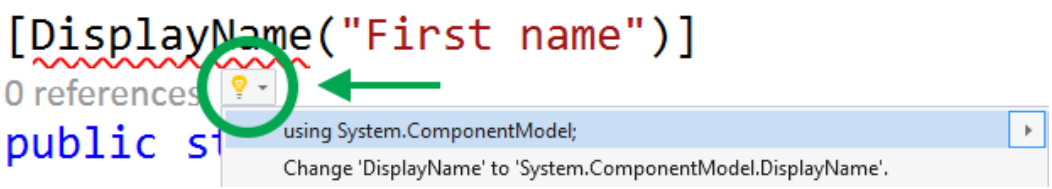
You can decorate properties with attributes to help the Razor view engine (and the Entity Framework for database tables) to better understand your intensions with the model properties. The added information from the model class' attributes can then be used when displaying controls in the view and when performing validation.

The following list describes different categories of frequently used attributes:

- DataTypeAttribute
  Specifies what type of data that will be entered into the control, it could be that it will contain a number or an email address. This is very helpful when the HTML 5 is rendered because if you use certain data type attributes on the properties the user can get a much nicer experience. For example if you state that a property should contain an email address the control can then be rendered as the new HTML 5 email control which automatically will display the @-sign and the .com buttons on the device keyboard. The data type can also help with data validation.
- DisplayName attribute
  Can be used to change the text that will be displayed in the control's label for the property. Let's say you have added a property called **FirstName** and want the control label in an input form to say "First name" (with a space between the two words) then you add a **DisplayName** attribute to the **FirstName** property. To use these attributes you have to add a **using** statement to the **ComponentModel** namespace at the beginning of the code file.
  ```
  using System.ComponentModel;
  ```
- ValidationAttribute
  - RequiredAttribute
    Specifies that this input field must have a value for the form to be posted back to the server. When creating a table model for Entity Framework this attribute indicates that **null** values are not allowed to be stored in the column.
  - StringLength, MinLength, MaxLength
    Specifies the number of characters that must be entered into the input control. To use these attributes you have to add a **using** statement to the **DataAnnotations** namespace at the beginning of the code file.
    ```
    using System.ComponentModel.DataAnnotations;
    ```
  - RegularExpressionAttribute
    Can be used to create a more advanced validation using regular expressions.

Tip: *A shortcut for adding a namespace is to write the name of the attribute or class name, right click on the name and select* ***Resolve-using…*** *in the context menu in Visual Studio 2013 or click on the "light bulb" menu button and select the same menu option in Visual Studio 2015.*





Exercise 2-1: Create a Model and add Attributes

In this example you will create a new model class in which you decorate the properties with attributes. For it to make any sense you have to jump ahead a little but I'm certain that you will understand it just fine.

The model you create will contain a **FirstName** and **EmailAddress** property where the **FirstName** property must contain at least 5 characters and its label should display the text "First Name" using attributes. The **EmailAddress** property is a required field and should be declared as an email field using the correct attribute.

Create the model

1. Open the *MVC Foundation* solution from the previous exercise if it isn't open already.
2. Right click on the **Models** folder.
3. Select **Add-Class** in the displayed context menu.
4. Name the model class **CustomerModel**.
5. Click the **OK** button or hit **Enter** on the keyboard.
6. The following model class is displayed in the IDE.
   The comment will not be present in your model class.

```
public class CustomerModel
{
    // Add properties and attributes here ...
}
```

7. The first property you will add is called **FirstName** with a **string** data type. You want to decorate the property in the following way:
    a. Add the **DisplayName** attribute to specify that the title (label) for this control should be displayed as "First name" with a space.
    b. Specify that the input must be at least 5 characters for it to be valid; use the **MinLength** attribute.

```
[DisplayName("First name")]
[MinLength(5)]
public string FirstName { get; set; }
```

8. The second property is called **EmailAddress** of type **string**.
    a. You want to explicitly specify that this property is an email address using the **EmailAddress** attribute.
    b. You also want to use the **Required** attribute to force the user to enter a value in the generated HTML 5 input field.

```
[EmailAddress]
[Required]
public string EmailAddress { get; set; }
```

The complete code for the **Model** class:

```
public class CustomerModel
{
    [DisplayName("First name")]
    [MinLength(5)]
    public string FirstName { get; set; }

    [EmailAddress]
    [Required]
    public string EmailAddress { get; set; }
}
```

Create the controller

1. Right click on the **Controllers** folder.
2. Select **Add-Controller** in the context menu.

3. Select *MVC 5 Controller - With read/write actions* in the dialog box and click the **Add** button.
4. Name the controller **CustomerController** in the dialog box and click the **Add** button.
   Note that the controller class called **CustomerController** is created in the **Controllers** folder and an empty folder named **Customer** is added to the **Views** folder.

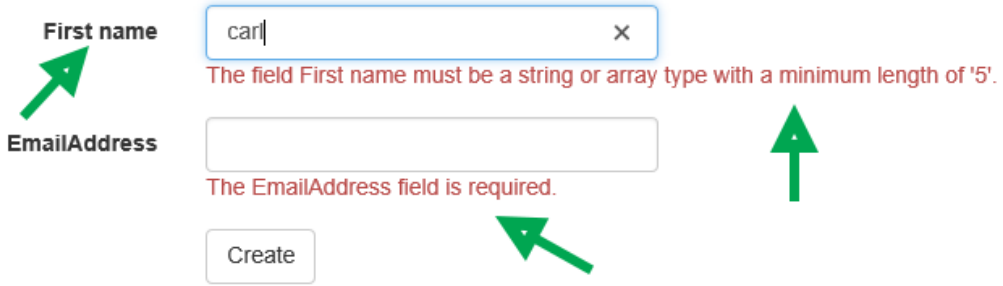
Create the Create view

1. Right click on the **Customer** folder inside the **Views** folder.
2. Select **Add-View** in the context menu.
3. Name the view **Create** in the **View name** field in the dialog box.
4. Select **Create** in the **Template** drop down.
5. Select **CustomerModel** in the **Model class** drop down.
6. Click the **Add** button to create the view.

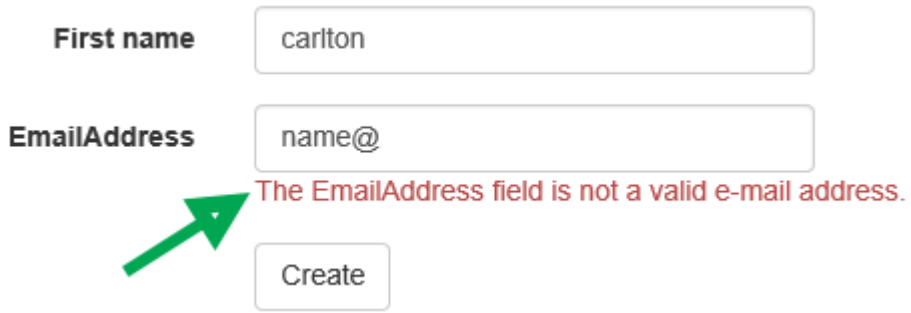
Test the model

1. Press **F5** on the keyboard.
2. Add *customer/create* to the URL if it is missing.
   http://localhost:xxxx**/customer/create**
3. Look at the label descriptions for the two text fields and note that the first label displays "First name" with a space and the second label displays "EmailAddress" without a space. The reason for this is that you only added the **DisplayName** attribute to the **FirstName** property.
4. Enter a name with less than 5 characters and see the validation act on the entered value.
5. Enter a faulty email address into the email field and a name containing more than 5 characters in the first name field.
6. Enter a correct email address in the email field.
7. Click the **Create** button. An error should be displayed. This error appears because no **Index** view is present in the **Customer** view folder and the **Create** view is trying to redirect to the **Index** view when it has finished working.
   ```
   @Html.ActionLink("Back to List", "Index")
   ```

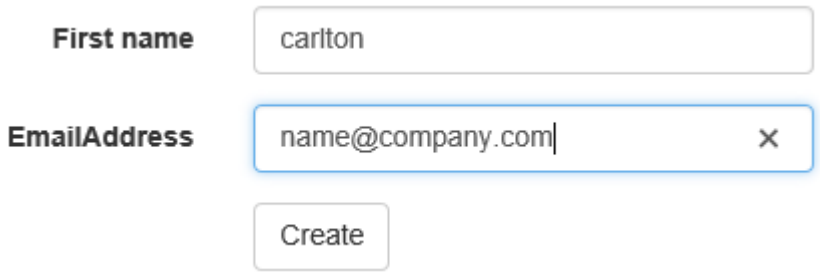
Too short name and empty email field



Validated name and faulty email address



Validated form fields



Create the Index view

1. Right click on the **Customer** folder in the **Views** folder.
2. Select **Add-View** in the context menu.
3. Name the view **Index** in the **View name** field in the dialog box.
4. Select **Empty** in the **Template** drop down.
5. Select **CustomerModel** in the **Model class** drop down.

6. Click the **Add** button to create the view.
7. Test the view like you did in the previous section.

What's Next?

In this chapter you learned that models really only are classes with attribute decorations on the properties and that they never should contain any logic. In the next chapter you will learn how to work with controllers which incidentally also are *just* classes. You will look at what routing is and how you can change routes if needed. I know it is easy to procrastinate and push things you know you should do immediately into the future; to get the most out of this training you should strive to do the exercises right after you have read a whole chapter or a section. Jump straight into the wonderful and exiting world of controllers when you have finished the exercises in this chapter, don't wait.

3. Controllers

Introduction

A controller is *just* a class.

With that said, in this chapter you will work with controllers and look at how to manage routes which handles the incoming requests to your controllers from the web. Then you will explore controller actions which are the public methods responding to the incoming HTTP requests.

Filters can be used to inject pre- and post- processing to an action. Parameters can be used to input data to an action and action results return data from an action.

It's important to know that any **public** method inside a controller class can be called from the browser. To avoid potential problems with this you should strive to only have action methods inside controller classes. If you need to call methods from within an action method those methods should be placed in another class, not the controller class itself. This class could be part of a business- or data layer or even a class containing extension methods for a specific purpose.

Technologies used in this chapter

- **C#** - To write code in the controller.
- **Actions** - Methods called over HTTP.
- **Routes** - Possible paths to an action method.
- **Accept verbs** - Defines how an action can be called (get, put, post).

Routes

The MVC routing engine is the same routing engine used by ASP.NET Web Forms and Web Services. How do ASP.NET know how to deliver a request like *http://localhost/home/contact* to the correct controller action? The answer is that the routing engine finds the correct controller and action by inspecting the URL. In the previous example the default routing engine would look at the first part of the URL which is *home* and route it

to the **Home** controller, then it would inspect the second part of the URL which is *contact* and find an action method matching that name in the **Home** controller.

Important: *Remember that you only have to specify your own routes if the default route is not covering your particular scenario.*

Important: *It is also important to know that the order in which the routes are added is significant because the routing engine in ASP.NET will use the first route matching the URL.*

Route Mapping

To handle routes a default route map has been added to the *RouteConfig.cs* file in the project **App_Start** folder. If your project need custom routes they can be added to this file.

The route map has three parts:

```
routes.MapRoute(
   name: "Default",                    // Route name
   url: "{controller}/{action}/{id}",  // URL and parameters
   defaults: new {                     // Parameter defaults
      controller = "Home",
      action = "Index",
      id = UrlParameter.Optional }
);
```

- Name

  A name that can be used to access the route.

- Url

  In MVC the route defines the controller and action to be called as well as other information such as an id or other data. The default route URL is defined as *{controller}/{action}/{id}* which would be mapped against for instance *{home}/{contact}* (*http://localhost/home/contact*) which would display the **Contact** page, or *{customer}/{edit}/{5}* (*http://localhost/customer/edit/5*) which would display the **Edit** page for the customer with id 5. Note that the **id** parameter in the default route is assigned **UrlParameter.Optional** which mean that it can be omitted if not needed.

- Defaults
  The values listed in this section are the parameter defaults that will be used if no value is provided in the URL.
  - Example: If you for instance navigate to *http://localhost/* in the browser it will automatically map against *http://localhost/Home/Index* because the specified default controller is **Home** and the default action is **Index**. The **id** can be omitted because it has been assigned **UrlParameter.Optional**.
  - Example: If you navigate to *http://localhost/Home* in the browser it will automatically map against *http://localhost/Home/Index* because you have explicitly specified a controller name and the default action is **Index**. The **id** can be omitted because it is set to **UrlParameter.Optional**.
  - Example: If you navigate to *http://localhost/Customer/Details/5* in the browser it will no longer map against *http://localhost/Home/Index* because you have explicitly specified the controller name as **Customer** and the action as **Details** which will receive the value *5* through its **id** action parameter. This action method would be defined as follows in the **Customer** controller:
    ```
    public ActionResult Details(int id) { }
    ```

Route Attributes

If you prefer to keep the routing closer to the source code you can add routing attributes to the controller class or its actions. To be able to use route arguments the method **MapMvcAttributeRoutes** has to be called after the call to the **IgnoreRoute** method and before the other routes in the *RouteConfig.cs* file.

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapMvcAttributeRoutes();

        // The default route and other routes are defined here ...
    }
}
```

Let's say that the controller name is **CustomersController** and you want to change the controller route from *http://localhost/Customers* to *http://localhost/Customer* without renaming the controller class. To achieve this you could apply the **RoutePrefix** attribute to the class to have the name propagated throughout the controller for all actions.

```
[RoutePrefix("Customer")]
public class CustomersController : Controller
{
}
```

If you want to change a route for a specific action you can add a **Route** attribute to that action. Let's say you want to make it possible to display the detail information about a specific customer passing in a customer id. The default scaffolded action would be called **Details** with an **int** parameter named **id** and the URL would be *http://localhost/Customer/Details/5* (where the number is the customer id). It is possible to change the routing for that call to *http://localhost/Customer/5* to make the URL a bit shorter, to achieve this you apply the **Route** attribute to the **Details** action method. The parameter name and data type specified within the curly braces in the **Route** attribute will be strongly typed to the specified data type. In this example only integer values would match the route, for any other type of value such as a string the route would not be met and the route engine would skip this action and search for another matching route.

```
[Route("Customer/{id:int}")]
public ActionResult Details(int id) { }
```

Exercise 3-1: Route Attributes

In this exercise you will add custom **Route** attributes to the **Customer** controller's **Details** action. The first route attribute will make it possible to access customer details through a URL referencing only the **Customer** controller and an id (*http://localhost/Customer/5*). The second route attribute will add the possibility to display customer details through a URL referencing the **Customer** controller using an alternate name for the action method (*http://localhost/Customer**/Detail**/5*).

Creating the Details view

First you need to add the **Details** view to the **Customer** folder located in the **Views** folder.

1. Right click on the **Views-Customer** folder.
2. Select **Add-View** in the context menu.
3. Name the view **Details** in the **View name** field.
4. Select **Details** in the **Template** drop down.
5. Select **CustomerModel** in the **Model class** drop down.
6. Click the **Add** button to create the view.

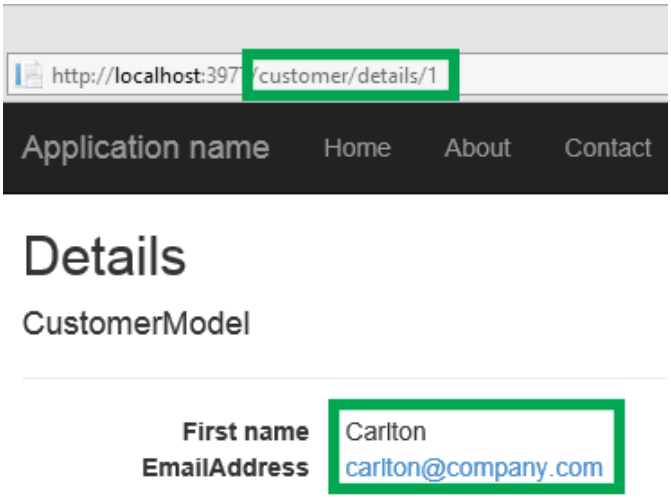
Test the new Details view

Run the application before making any changes so that you know what the default behavior is for navigating to the view.

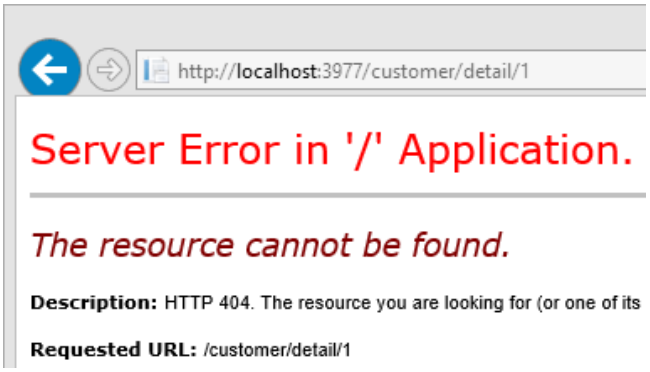
1. Open the **Customer** controller and find the **Details** action method and add a fake customer, you use a fake customer since you don't have a database to fetch data from. Render the customer with the view.

```
public ActionResult Details(int id)
{
    var customer = new CustomerModel
    {
        EmailAddress = "carlton@company.com",
        FirstName = "Carlton"
    };

    return View(customer);
}
```

2. Press **F5** on the keyboard to start the application
3. Navigate to the following URL (*port* is the localhost port number):
   http://localhost:port/Customer/Details/1
   This should display the **Details** page with the fake customer data.



4. Now change the URL to *http://localhost:port/Customer/Detail/1* removing the *"s"* from the **Details** action name. This should display an error page with a *404 - "The resource cannot be found"* error because thre is no action with that name.
5. Now change the URL to *http://localhost:port/Customer/1* removing the *Details* action name completely. This should also display an error page with a *404 - "The resource cannot be found"* error because the default action is **Home** not **Details**.



Add routing attributes

Now let's add the routing attributes to the **Details** action in the **Customer** controller.

1. Close the browser and stop the application.
2. Open the **Customer** controller and locate the **Details** action method.

3. Add the following two **Route** attributes immediately above the **Details** action method.

```
[Route("Customer/{id:int}")]
[Route("Customer/Detail/{id:int}")]
public ActionResult Details(int id)
```

4. Open the *RoutingConfig.cs* file in the **App_Start** folder and add the following method call immediately below the **IgnoreRoute** method call.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapMvcAttributeRoutes();
    // The default route and other routes are defined here ...
}
```

5. Press **F5** on the keyboard to start the application.
6. When the browser has opened enter the following URL in the browse field:
   http://localhost:port/Customer/Details/1
   This should display a 404 error message.
7. Now try the previous two URLs *http://localhost:port/Customer/Detail/1* and *http://localhost:port/Customer/1*
   There URLs should display the **Details** page with the fake customer data.

Actions and Parameters

Now that you have seen how to add routes and have added your own route, it is time to dig into routes and action parameters. Let's say that you want the user to be able to search for a customer by name. To achieve this you could add a custom route to the *RoutingConfig.cs* file. The user should be able to enter a URL on the following form: */customer/**name*** where ***name*** is substituted for an actual name.

Because the name specified could be Carl, Joanna, Lisa or any other name the default route will not work for this scenario unless you want to add an action method for every name in existence.

Add the following custom route above the default route in the *RoutingConfig.cs*.

```
routes.MapRoute("Customer", "customer/{name}",
   new { controller = "Customer", action = "Search", name = "" });
```

You might wonder how a route like the one previously described can work without a controller and action specified in the defined path. It works because of the default values added in the route's anonymous object.

If you run the application now and navigate to the URL *http://localhost:port/Customer/Carl* you will get a 404 error because no action in the **Customer** controller handles that request. By adding the following **Search** action to the controller class the URL will reach a page displaying the string "Search reached".

```
public ActionResult Search()
{
    // Will display the provided text in the browser, no view needed.
    return Content("Search reached");
}
```

If you run the application now and provide the same URL as before it will work and the text "Search reached" will be displayed in the browser. This works because of the default value provided for the *name* part of the URL defined by the custom route. The default value in the route for the **name** parameter is an empty string (*name = ""*).

If you don't want to provide a default value for the *name* part of the URL in the custom route then you can specify it as being an optional value which mean that it can be left out completely or provided as needed by the user. Just change the *name = ""* in the route to the following:

```
name = UrlParameter.Optional
```

It is however likely that you actually want the user to be able to send in a name to the action method; to enable that you have to change the action method to take a **name** parameter. You most likely want to do something with that passed-in value, let's display it in the browser to confirm that the action was reached.

It is also possible to give a default value for the **name** parameter in the action's parameter list by simply assigning a string value to it inside the action method's parenthesis.

Important: *The routing engine will do everything it can to populate a parameter. It will look in routing data from the URL, in the query string and in posted form values. It will first try to match the parameter by name and if that fails by order and data type.*

Before you use the value provided by the **name** parameter inside the action you could use the **HtmlEncode** method on its value to prevent any cross-site scripting attacks from happening when using the received value. MVC will automatically encode text that is rendered to the browser but you might consider encoding the parameter value if you use it in the controller's action.

Exercise 3-2: Action Parameters

In this exercise you will add a new action called **GetById** to the **Customer** controller class. This action will take an **int** parameter called **id** and based on the passed in value will return the appropriate customer from a list of customers. Then you will redirect from the **GetById** action to the already existing **Details** view to display the customer information. You also have to add a property called **Id** of type **int** to the **CustomerModel** class so that the customer can be fetched by its id.

Because we haven't talked about Entity Framework yet and therefore don't have a functioning database you will work with temporary data stored within the action itself. I must stress that this is not how you typically would store and fetch data in a real world scenario but to make it as simple and time saving as possible it will have to do for this particular exercise.

Adding the Id property to the CustomerModel class

You need to add the **Id** property to be able to store unique ids for the customers. Later you will use this property to fetch a customer based on the value stored in it.

1. Open the **CustomerModel** class located in the **Models** folder in the Solution Explorer.
2. Add a property called **Id** of type **int** to the class. The order of the properties in the class is irrelevant. Remember that you can write ***prop*** and hit the **Tab** key twice to add a property.
3. Save the class.

Adding the GetById action

1. Open the **CustomerController** class from the Solution Explorer.
2. Add a **public** action method called **GetById** which takes a parameter called **id** of type **int** and return an **ActionResult**.
```
public ActionResult GetById(int id) { }
```
3. Next add a variable called **customers** inside the **GetById** action which holds a list of **CustomerModel** objects. Add the following three **CustomerModel** objects to the list Carl (id: 1, carl@company.com), Lisa (id: 2, lisa@company.com) and Joanna (id: 3, joanna@company.com).
```
var customers = new List<CustomerModel>
{
    new CustomerModel{ Id = 1, EmailAddress = "carl@company.com",
        FirstName = "Carl"},
    new CustomerModel{ Id = 2, EmailAddress = "lisa@company.com",
        FirstName = "Lisa"},
    new CustomerModel{ Id = 3,
        EmailAddress = "joanna@company.com",
        FirstName = "Joanna"}
};
```
4. Query the list and fetch the customer matching the id passed in to the action. Assign the result to a variable called **customer**. You can use LINQ and Lambda to fetch the customer from the list if you like.
```
var customer = customers.FirstOrDefault(c => c.Id.Equals(id));
```
5. The last thing you need to do is to render the appropriate view. Since there is no view called **GetById** you can reuse the **Details** view and display the customer information in it. Pass in the name of the view (**Details**) and the **customer** object you fetched from the list to the **View** method when returning from the action. If you look in the **Details** view you can see that its model type is **CustomerModel** which is exactly what you fetched from the list.
```
return View("Details", customer);
```
6. Save the changes.

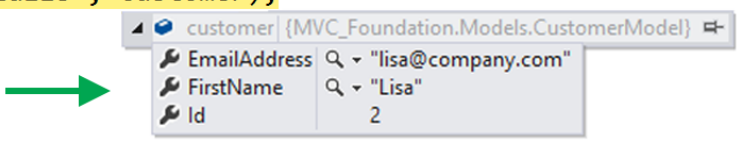
Testing the GetById action

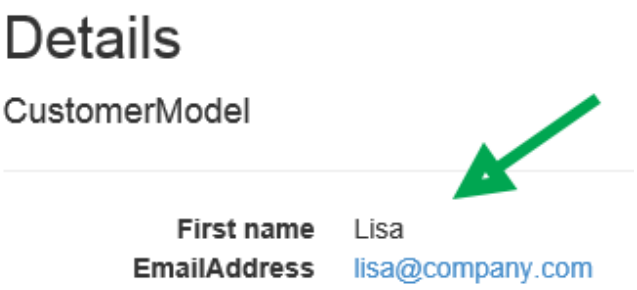
1. Press **F5** on the keyboard to start the application.
2. Call the action from the URL field in the browser.

http://localhost:port/customer/getbyid/2

3. The **Details** view should be displayed with the customer data for the customer matching the id passed to the **GetById** action.

```
public ActionResult GetById(int id)
{                                          id 2
    var customers = new List<CustomerModel>
    {
        new CustomerModel{ Id = 1,
            EmailAddress = "carl@company.com",
            FirstName = "Carl"},
        new CustomerModel{ Id = 2,
            EmailAddress = "lisa@company.com",
            FirstName = "Lisa"},
        new CustomerModel{ Id = 3,
            EmailAddress = "joanna@company.com",
            FirstName = "Joanna"}
    };

    var customer = customers.FirstOrDefault(c => c.Id.Equals(id));

    return View("Details", customer);
}
```





Action Accept Verbs

When fetching data from the server the call end up in an action method which is decorated with the **HttpGet** verb. Because this is the default verb it can be omitted from the action definition.

To be able to send data from the browser to the server, for instance posting a form with data filled in by the user, then an **HttpPost** accept verb must be added to the action handling the request.

Inside the **Customer** controller where you added read and write actions earlier the **Create**, **Edit** and **Delete** actions appear twice; one will serve up the view to the browser and the other will send data back to the server. The action without the **HttpPost** verb will be used to render the view and the action with the **httpPost** verb will be used when the user clicks the view's **submit** button to send data to the server.

```
// This action uses HttpGet
public ActionResult Create()
{
    return View();
}

[HttpPost]
public ActionResult Create(FormCollection collection)
{
    try
    {
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The **RedirectToAction** returned after the post action has finished successfully will send the call to the **Index** action in the current controller redirecting the browser to that view.

If an exception occurs in the **HttpPost** version of the **Create** action the current view will be returned to the browser keeping the user on the same web page (view) from which the post originated.

What's Next?

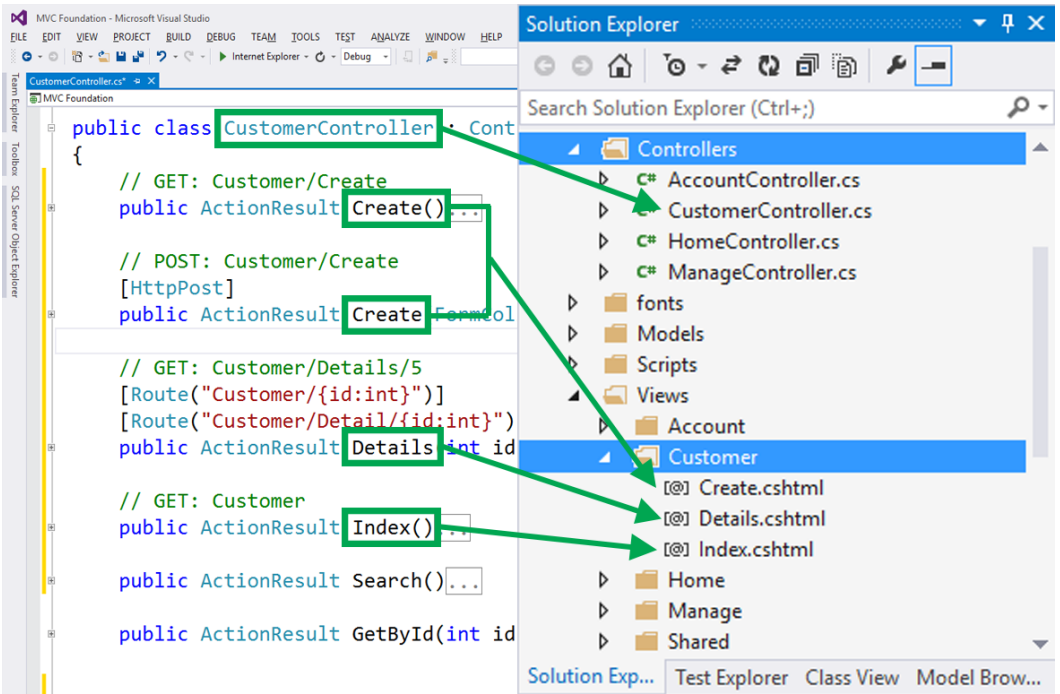
In this chapter you took a deep dive into controllers and routes. Routes are not something that you will dabble with on a regular basis but it's vital to know how they work in order to fully understand MVC. In the next chapter you will learn how to work with Razor views creating the HTML template in which C# code is mixed with Razor syntax and HTML to bring astounding web content to life. Don't procrastinate and skip the exercises in this chapter, do them and then jump straight into the wonderful world of Razor views.

4. Razor Views

Introduction

Views are built using Razor syntax, C#, HTML 5 and CSS which is rendered to the user's browser on demand. When the user types in a URL in the browser the route engine will pick it apart and figure out which controller and action to call.

Views adheres to a naming convention which states that all views associated with a specific controller must be placed inside a folder with that controller's name. Each view should have an action method with the same name as the view in the corresponding controller.



Apart from the different controllers view folders there is also one named **Shared** which contain views that are available anywhere in the application. Views placed in a subfolder inside the **Views** folder are only available from the corresponding controller and other

views inside that subfolder. If you want to reuse a view or a partial view then you place it inside the **Shared** folder.

The order in which the view engine will look for views are:

1. Inside a folder with the controller's name in the **Views** folder.
2. Inside the **Shared** folder in the **Views** folder.
3. If it can't find the view in either folder it will throw an error.

By default many actions return the view it originated from.

```
public ActionResult Index()
{
    return View();
}
```

You sometimes want to change this default behavior and redirect to another view. You might for instance create a view for editing item values and reuse that view when creating new items. To achieve this you simply specify the name of the view inside the parenthesis when returning the view. To send a model to the view you pass the object as a second parameter.

```
return View("EditCreateView");
```

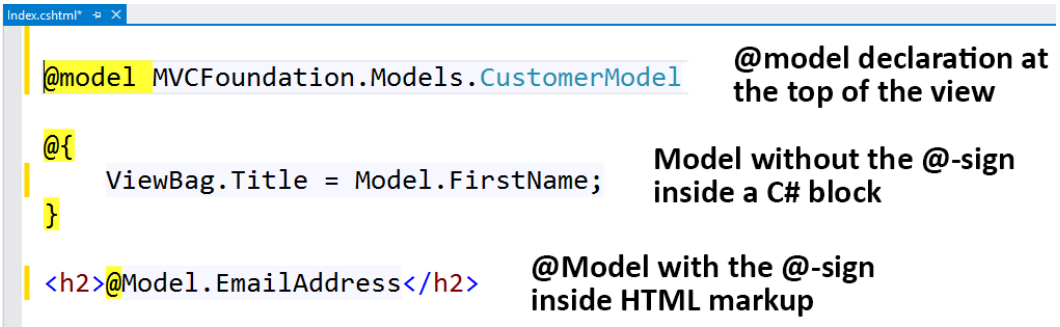
or

```
var model = new CustomerModel { EmailAddress = "a@b.com",
    FirstName = "alice" };

return View("EditCreateView", model);
```

Although it is possible to use un-typed views sending data using only the **ViewBag** it's not best practice and generally not how you want to design your MVC application. There are many benefits to strongly typed views using specific classes as models as you will learn in this chapter.

To bind a model to a view you have to add the **@model** directive referencing the full namespace and model class at the top of the view's *.cshtml* file. To access the data passed in through the model you use the **Model** or **@Model** objects which provide

IntelliSense when writing the code, which one you use depends on if you are inside a C# block or in HTML markup.



Note that the **@model** directive is written in all lowercase letters and the **Model/ @Model** objects begin with an uppercase M.

Technologies used in this chapter

- **HTML 5** - HTML elements are used when creating views.
- **Razor** - Makes it possible to use C# in HTML views.
- **Views** - HTML markup, Razor syntax and C# rendered into HTML when displaying the web page.
- **Layout view** - A special view loaded for all views.
- **@Html extension methods** - Enables reuse of HTML, often with a single line of code.
- **Partial views** - Can be used to render a portion of the page as a separate view. This enables reuse and the possibility to update portions of a page without reloading the whole page.
- **Form components** - Different components used when building an input form such as textboxes, checkboxes and buttons.
- **Form validation** - Used to validate user input in components without having to do a roundtrip to the server.

Razor Syntax

The **@**-sign plays a significant role in Razor syntax since it mostly denotes server-side code; code which will be executed on the server such as C# code. The Razor engine is

very good at distinguishing when to use the **@**-sign as literal text and when it signifies server-side code. There are however a few edge cases you should be aware of.

- @MyTwitterHandle
  You would expect this to be interpreted as literal text but the Razor engine stumbles here and think it is C# code being executed. To avoid the dreaded "yellow screen of death" showing an error message you must tell the Razor engine that it should be interpreted as text by using two @-signs.
  @@MyTwitterHandle
- carlton@company.com
  Email addresses are however handled without error because whenever Razor encounters *someone at somewhere dot something* it will interpret it as being an email address and display it accordingly.

For the most part you don't have to worry about "knowing" how to write Razor syntax because it comes naturally. As long as you write HTML where you want and need and use the **@**-sign where you want server-side C# code it will just work most of the time. If you like to dig in to the Razor syntax in more detail you can visit www.asp.net.

Looking at the code below you can see that Razor do a very good job of determining where HTML and C# begins and ends. We will go through the most common Html helper methods later in this chapter but for now I want you to note all the places where the **@**-sign is present in the HTML markup. Also note that a C# code-block can encapsulate HTML like in the **foreach** loop listing the customer data.

```
<table class="table">
    <tr>
        <th>@Html.DisplayNameFor(m => m.ISBN)</th>
        <th>@Html.DisplayNameFor(m => m.Title)</th>
    </tr>

    @foreach (var item in Model) {
    <tr>
        <td>@Html.DisplayFor(modelItem => item.ISBN)</td>
        <td>@Html.DisplayFor(modelItem => item.Title)</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id = item.Id }) |
            @Html.ActionLink("Details", "Details", new { id = item.Id }) |
```

```
            @Html.ActionLink("Delete", "Delete", new { id = item.Id })
        </td>
    </tr>
    }
</table>
```

The model class used with the view:

```
public class Book
{
    public int Id { get; set; }
    public string ISBN { get; set; }
    public string Title { get; set; }
}
```

The following list of books is used in the upcoming examples:

```
static List<Book> books = new List<Book>
{
    new Book{ Id = 1, ISBN = "123456789", Title = "My Fantasy Bok"},
    new Book{ Id = 2, ISBN = "234567890", Title = "My Sci-Fi Bok"},
    new Book{ Id = 3, ISBN = "345678901", Title = "My Romance Novel"}
};
```

Important: *It's important to know that even though server-side code is involved in many places only one round-trips to the server is made when the view is rendered. It's beyond the scope of this book to go into detail on how this is handled behind the scenes but the short version is that caching and other nifty implementations on the server side handles it.*

Cross-Site Scripting Attacks (XSS)

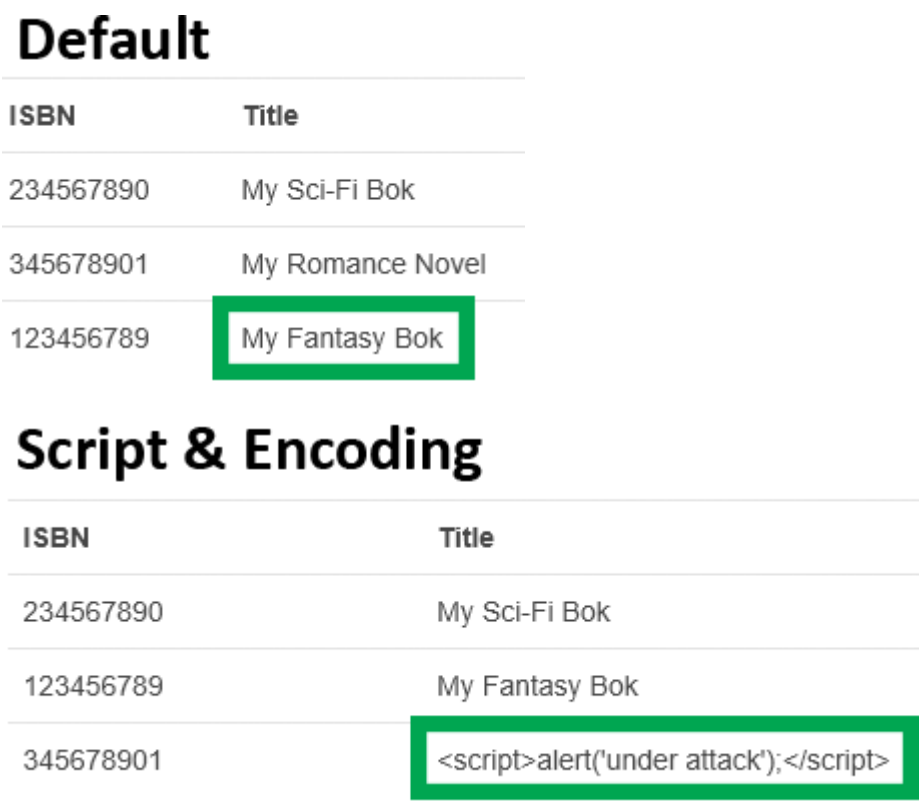
Cross-site scripting attacks (XSS) is something you have to be very mindful of because it is the #1 browser vulnerability on the web. XSS can be very harmful to your data and system, it can steal cookies from users, execute malicious scripts or display fake login dialog boxes. To safeguard against XSS you must HTML encode all data sent to your system from a user. Fortunately this is done automatically by Razor in a MVC application.

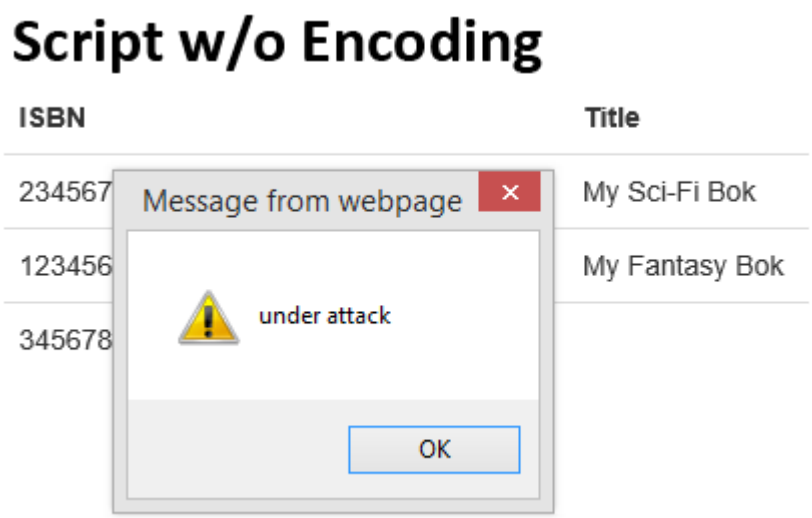
If you for some reason want to get around the automatic HTML encoding Razor provides you can use an HTML helper called **Raw** to display the string un-encoded.

Razor syntax example: `@Html.Raw(item.Title)`

If the view has the content described above and is sent a list of books the book list should appear in the browser. But if a hacker somehow managed to insert a malicious script during an XSS attack you could be in a lot of trouble; if the text sent to your database during a post back wasn't HTML encoded by default or if you explicitly forego the encoding with the **Raw** helper the web page could be hijacked. Let's see what happens in three scenarios where you leave the default setting without a script, use the default setting with a script and what would happen if the script was executed.

Changing the title of the third book from "My Romance Novel" to "**<script>alert('Under Attack');</script>**" would be annoying for the site visitors but not really harmful. Imagine what could happen if a malicious script was stored and executed.





Exercise 4-1: Cross-Site Scripting

In this exercise you will test XSS first hand by implementing a new view, controller and a **Book** model class containing **Id**, **Title** and **ISBN** number properties. Start by adding the model and then create a new controller called **BooksController** with read/write actions. Then create a view called **Index** which displays a list of books when the **Index** action is called from the browser.

Adding the model

1. Right click on the **Models** folder.
2. Select **Add-Class** in the context menu.
3. Name the class **Book**.
4. Click the **OK** button or hit **Enter** on the keyboard.
5. Add the following properties to the model class:

```
public int Id { get; set; }
public string ISBN { get; set; }
public string Title { get; set; }
```

6. Save the model.

Adding the controller

1. Right click on the **Controllers** folder.
2. Select **Add-Controller** in the context menu.

3. Select *MVC 5 Controller - With read/write actions* in the dialog box and click the **Add** button.
4. Name the controller **BooksController** in the dialog box and click the **Add** button.
5. Create a list of **Book** objects in the controller class above the **Index** action method (you can use the book list described earlier in this chapter).
6. Use LINQ inside the **Index** action to fetch the list of books in descending order on the title and pass the list to the view.

The controller's code:

```
public class BooksController : Controller
{
    static List<Book> books = new List<Book>
    {
        new Book{ Id = 1, ISBN = "123456789", Title = "My Fantasy Bok"},
        new Book{ Id = 2, ISBN = "234567890", Title = "My Sci-Fi Bok"},
        new Book{ Id = 3, ISBN = "345678901", Title = "My Romance Novel"}
    };

    // GET: Books
    public ActionResult Index()
    {
        var model = books.OrderByDescending(b => b.Title);
        return View(model);
    }

    // Other Action methods ...
}
```

Adding the Create view

1. Right click on the **Index** action name or inside the **Index** action.
2. Select **Add-View** in the context menu.
3. Name the view **Index** in the **View name** field.
4. Select **List** in the **Template** drop down.
5. Select **Book** in the **Model class** drop down.
6. Click the **Add** button to create the view.

Test the model

1. Press **F5** on the keyboard.

2. Add *books/Index* to the URL if it is missing:
   http://localhost:xxxx/**books/index**
3. The books should be listed in the view sorted on the **Title** property.
4. Now change one of the titles to ***<script>alert('You have been hacked!');</script>***
5. Run the application again. The title you changed should now display the script as text. The reason the script is not executed is that all rendered text is HTML encoded by default.
6. Now open the **Index** view you just created (you can find it in the **Views**-**Books** folder).
7. Comment out the following code:
   ```
   @Html.DisplayFor(modelItem => item.Title)
   ```
8. Add the following code on an empty line below it:
   ```
   @Html.Raw(item.Title)
   ```
9. Run the application again. A dialog box with the text "You have been hacked" should appear when you navigate to the **Index** view.
10. Close the dialog box and the browser and stop the application.
11. Restore the book title to its original value.
12. Comment out the **Raw** helper in the **Index** view.
13. Remove the commented out **DisplayFor** helper in the **Index** view.
14. Save the solution.

Code Expressions

Implicit and explicit code expressions are available in Razor. When using implicit code expressions Razor will figure out how to render the code. With the following code expression Razor will try to figure out what was intended and would interpret it as a property value and literal text. Razor would not divide the property value with the numeric value.

```
@item.Value / 5
```

If the **Value** property contain **10** the output would be the literal text *10 / 5* not *2* which is the expected result of the division.

If your intension was to actually display the result of the division then you would have to explicitly tell Razor to do so by adding parenthesis around the expression.

```
@(item.Value / 5)
```

Layout Views

As you might have noticed by now there are things displayed in the browser which are not part of the views you have created; that HTML markup is part of a layout view called *_Layout* located in the **Views**-**Shared** folder. The layout view contains elements that are common to all pages on the site such as the menu bar at the top, the header- and possibly a footer section.

Let's look at the layout view in small chunks. The first element is the **DOCTYPE** tag which denotes that HTML 5 is being used. Then the opening **<html>** tag begins the HTML page; this tag has a matching closing tag at the end of the layout page.

```
<!DOCTYPE html>
<html>
```

Next is the **<head>** section which contain settings that will be loaded when the page is displayed. Here you find metadata definitions for the character set used on the page and how the content will be rendered with the viewport data. Then the browser tab title is defined.

The *Styles.Render* method will render the CSS style sheet content from the bundled CSS style sheets. Instead of sending each CSS file one at a time they are bundled together and sent as one file to the client. The same goes for the script files which also are bundled before they are sent to the client. The scripts are un-bundled by the client when the *Scripts.Render* method is reached.

You can see the bundles and their associated files by opening the *BundleConfig.cs* file in the **App_Start** folder.

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0">
```

```
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
```

Next is the **<body>** section which contain all code that will be displayed between the header and footer sections. The first thing loaded in the **<body>** section is the **nav-bar** (menu bar). All the **nav-bar** classes added to the ** ** elements are Bootstrap classes defining the style of the menu and menu items. A more detailed description of Bootstrap and its classes will be covered in an upcoming chapter.

If you look at the menu you can see that its menu items are represented by a styled un-ordered list, a list of bullet points containing anchor tags displayed in a fancier way. The **ActionLink** helper method will produce anchor tag links, more on that later in this chapter.

There is also an HTML helper method called **Partial** which will render another view as a part of the current view. You will get acquainted with and do an exercise on that later in this chapter.

```
<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("Application name", "Index", "Home",
                    new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
```

```
        </div>
    </div>
```

The menu is followed by the container section in which the views you create will be loaded using the **RenderBody** method, this method call must always be present in the layout view. The container also has a **<footer>** section which is displayed at the bottom of each page.

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
</div>
```

More script bundles are rendered at the end of the **<body>** section. The **RenderSection** method call will check the views for defined **@section** blocks and render them. The most common sections are JavaScript which can come in handy if a JavaScript declared in a view don't load properly. If you want to render JavaScript from a view you place the JavaScript code inside a section called **scripts** in that view.

If the required parameter is set to **true** in a **RenderSection** method call then that section must be present in all views. If it is set to **false** the section can be provided in any or none of the views as you see fit.

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

HTML Helpers

HTML Helpers use reflection to figure out how to represent a property as an HTML control. The performance hit is negligible in contrast to the gains in easy and fast coding as well as UI benefits. Another thing that speaks in favor of using HTML helpers is that changes over time will not break your code like your own "hand written" HTML would if you for instance changed or added properties.

There is one thing I want to mention before we look at the different HTML helpers one at a time and that is that Lambda expressions are used in most of them. Looking at the code below **model** is used in the Lambda statement, but which **model** is it referring to and why is it there? In this case **model** actually refers to the **model** with a capital letter M (**Model/@Model**) which might be confusing. The reason the Lambda statement is using the **model** object is that it needs to know about a specific property in order to get to the attributes defined on the property.

```
@Html.DisplayFor(model => model.FirstName)
```

To avoid confusion between the **@model** directive and the **model** used in the Lambda statement you can simply use another name for **model** in the lambda statement. In the example below **model** has been changed to **m**.

```
@Html.DisplayFor(m => m.FirstName)
```

If you tried to use the model object **Model/@Model** in the Lambda expression you would end up with a totally different result because then the value of the property would be used instead of the property name.

```
@Html.DisplayFor(Model.FirstName)
```

DisplayNameFor

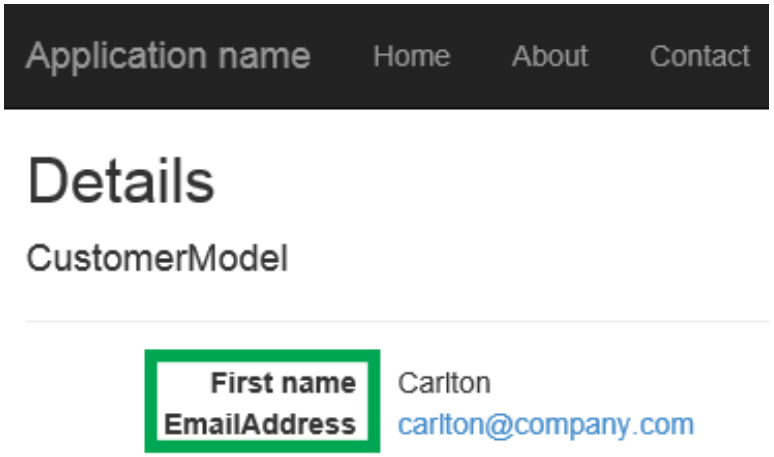
This helper displays the name of the property or the value given in the **DisplayName** attribute in the model class. In the example code below the text "First name" would be displayed for the **FirstName** property because it is decorated with the **DisplayName** attribute and "EmailAddress" would be displayed for the **EmailAddress** property because it don't have the **DisplayName** attribute specified.

Razor syntax example:

```
@Html.DisplayNameFor(m => m.FirstName)
@Html.DisplayNameFor(m => m.EmailAddress)
```

The model class:

```
[DisplayName("First name")]
public string FirstName { get; set; }
public string EmailAddress { get; set; }
```

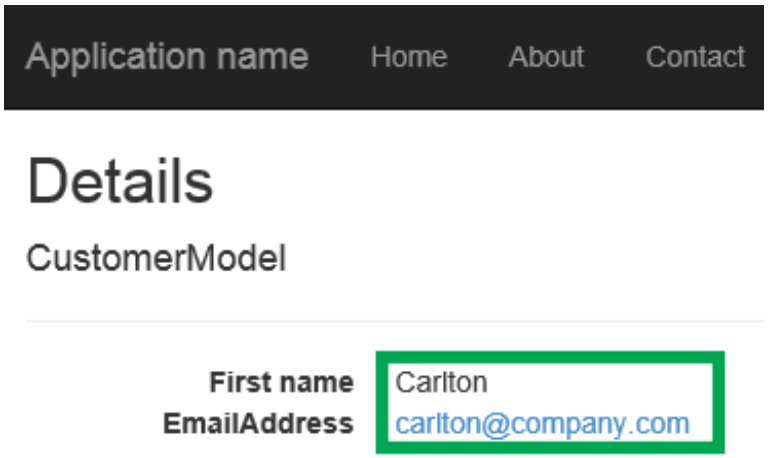


DisplayFor

This helper displays a property value. Using the code from the previous section the name "Carlton" will be displayed for the **FirstName** property and "carlton@company.com" for the **EmailAddress** property.

Razor syntax example:

```
@Html.DisplayFor(m => m.FirstName)
@Html.DisplayFor(m => m.EmailAddress)
```



Exercise 4-2: DisplayFor and DisplayNameFor

In this exercise you will create a view from scratch which displays a static list of books using the **DisplayFor** and **DisplayNameFor** helpers. You will not use the scaffolding to create this view instead you will create an empty view and build it from the ground up.

Adding the GetStaticBooks action method

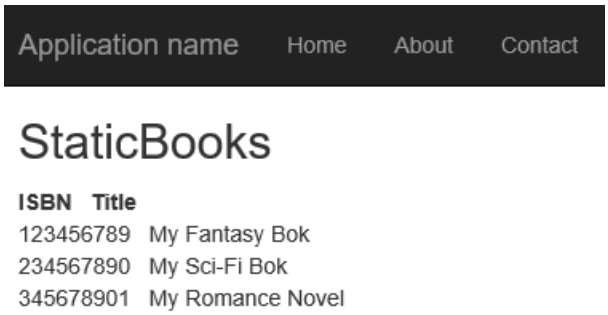
1. Open the **BookController** if it's not already open.
2. Add a method called **GetStaticBooks** with a return type of **ActionResult** which return the book list as is.

```
public ActionResult GetStaticBooks()
{
    return View("StaticBooks", books);
}
```

3. Save the controller class

Creating the StaticBooks view

You want to display the default heading for the view and a list of all the books with the model's property names as headings. Create a view named **StaticBooks** in the **Books** folder which is rendered from the **GetStaticBooks** action method. Instead of displaying the book list using a table you will use ** ** elements to display the data.



1. Right click on the **Views**-**Books** folder and select **Add**-**View**.
2. Name the view **StaticBooks**. Leave all other settings with their default values except the checkbox **Create as a partial view** which should be unchecked.
3. Click the **Add** button to create the view.
    a. A view containing a C# block with a **ViewBag** assignment and a heading should appear.

4. Go to the top of the view and add the **IEnumerable<Book>** @model directive. The **Book** model is located in the **Models** folder. Note that the complete path to the **Book** class must be used. **IEnumerable** is used to when handling a list of items.

```
@model IEnumerable<MVC_Foundation.Models.Book>
```

5. Go to the end of the view below the heading.
6. Add a ** ** element containing a **** element. Place the call to the first **DisplayNameFor** method referencing the **ISBN** property inside the **** element. Add a space (** **) and then the second **DisplayNameFor** referencing the **Title** property. It is important to add a space between the closing parenthesis and ** **.

```
<div>
    <strong>
        @Html.DisplayNameFor(m => m.First().ISBN)  
        @Html.DisplayNameFor(m => m.First().Title)
    </strong>
</div>
```

7. Add a **foreach** loop with start and end curly braces.
8. Add a ** ** element to the loop and place the call to the first **DisplayFor** referencing the **ISBN** property inside the ** ** element. Add a space (** **) and then the second **DisplayFor** method referencing the **Title** property.

```
@foreach (var item in Model)
{
    <div>
        @Html.DisplayFor(m => item.ISBN)  
        @Html.DisplayFor(m => item.Title)
    </div>
}
```

9. Save the view.
10. Run the application and enter the following URL to display the book list: *http://localhost:**xxxx**/books/getstaticbooks*

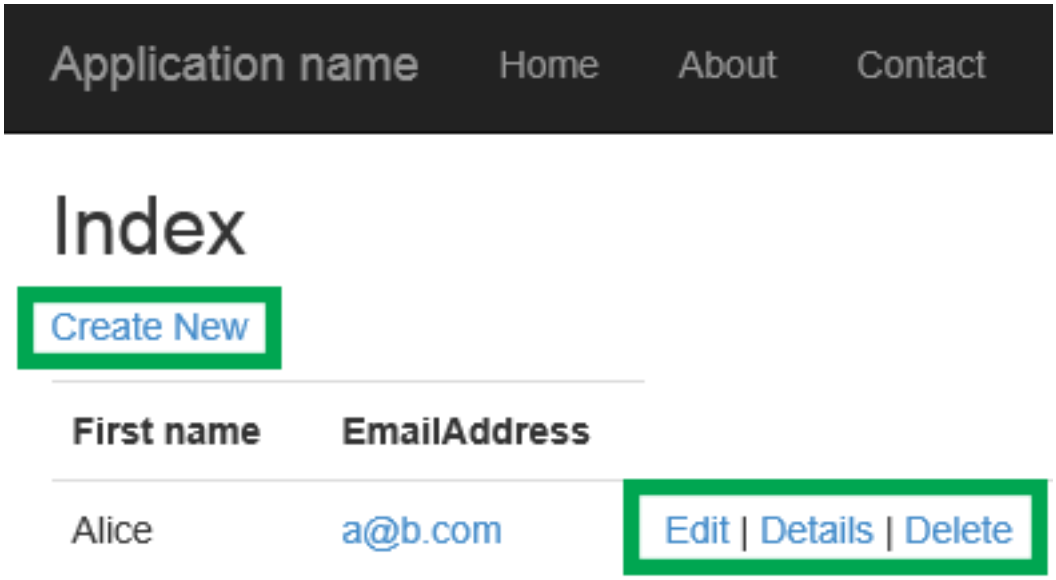
ActionLink

This helper is used to create links to other views in the application like the **Edit**, **Details** or **Delete** views. The first parameter is the link text, the second the controller action to

call when clicked and the third is any values you need to send to the action method. One value you might want to include is the item id, in this case a customer id.

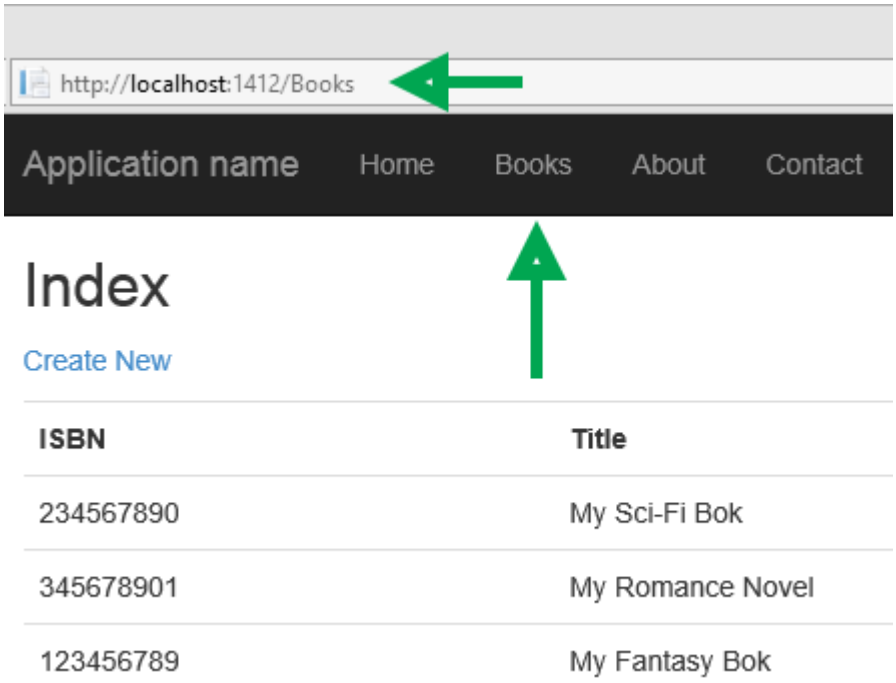
ActionLink will look at the routes defined in the application when determining which action to call.

Razor syntax example: `@Html.ActionLink("Edit", "Edit", new { id = Model.CustomerId })`



Exercise 4-3: Add a menu item ActionLink

In this exercise you will add a link to the **Index** action in the **BooksController**. To achieve this you will add an **ActionLink** to the **_Layout** view.



Adding the ActionLink

1. Click on the **_Layout** view in the **Shared** folder
2. Copy the first **ActionLink** in the menu section and paste it into the list of menu items.
3. Change the parameters of the **ActionLink**:
    a. The first parameter is the text to be displayed in the menu.
    b. The second parameter is the action method the menu item should call when clicked.
    c. The third parameter is the controller where the action is located. Note that you leave out the controller suffix.

```
<li>@Html.ActionLink("Books", "Index", "Books")</li>
```

4. Run the application and click the **Books** link in the menu to display the book list in the **Index** view of the **Books** controller.

Form Helpers

The following form helpers are often used when creating input forms to update or add data.

Example input form:

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>CustomerModel</h4>
        <hr />
        @Html.ValidationSummary(false, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.FirstName, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.FirstName,
                    new { htmlAttributes = new { @class = "form-control" }})
                @Html.ValidationMessageFor(model => model.FirstName, "",
                    new { @class = "text-danger" })
            </div>
        </div>

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

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create"
                    class="btn btn-default" />
            </div>
        </div>
    </div>
```

```
}
```

The model class below will be used throughout the chapter when describing controls.

```
public class DisplayModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsDirty { get; set; }
    public DateTime CurrentDate { get; set; }
    public ICollection<SelectListItem> Customers { get; set; }
}
```

BeginForm

This helper creates an open **<form>** tag; the form sends the response to the view's action method when the **submit** button is clicked. The helper is used in conjunction with a **using** block to group together the form's code.

```
@using (Html.BeginForm())
{
    @* HTML/Razor/C# defining the form *@
}
```

AntiForgeryToken

An anti forgery token is used to guarantee that a form post originated from your form and not from another page. It is a unique value which is evaluated on roundtrips to the server side controller actions.

Razor syntax example: `@Html.AntiForgeryToken()`

HiddenFor

This helper creates a hidden placeholder for a value. If the model contains an id then that id could be represented as a hidden value in the view because there is no reason for the user to see or be able to modify a unique id. This helper is often used when displaying **Edit** forms to the user.

Razor syntax example: `@Html.HiddenFor(m => m.Id)`

LabelFor

This is the equivalent of **DisplayNameFor** when creating an input form. It will create a label to display the property name or the value in the **DisplayName** attribute for the property in the model class.

The first parameter is the property for which the name is displayed and the second parameter is an un-typed object for HTML attributes. In this case some Bootstrap CSS classes have been applied to make the output look nicer.

Razor syntax example:

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

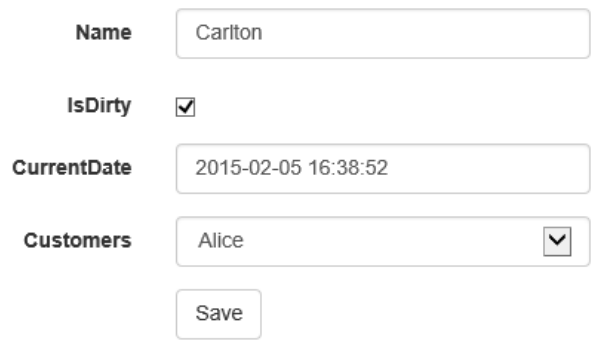
EditorFor

This helper is very versatile and can be used to create controls for many different data types. Among the data types it handles are **string**, **bool** and different numeric data types. If the HTML renderer can't figure out how to display a value it defaults to a text field (input field of type text). The parameter represent the model property to be rendered.

As you can see in the image below the date is displayed as regular text. If you want to have a date picker of some sort you have to implement it on your own or download code for it, there are plenty to choose from.

The drop down is available through the **DropDownFor** HTML helper.

Razor syntax example: `@Html.EditorFor(m => m.IsDirty)`



DropDownListFor

This helper displays a list of text and value pairs using the **SelectListItem** class when generating the items for the list. The first parameter is the model property containing the selected value, the second parameter is the list and the third is an optional un-typed list with HTML attributes. In this case a Bootstrap CSS class has been applied to make the output look nicer.

Model code:

```
public ICollection<SelectListItem> Customers { get; set; }
```

Razor syntax example:

```
@Html.DropDownListFor(m => m.CustomerId, Model.Customers, new { @class =
"form-control" } )
```

The image shows the difference in appearance between using or omitting the **form-control** Bootstrap class in the anonymous object.

ValidationMessageFor

This helper will display a validation message under the control if the validation fails. The first parameter is the property for which to display the validation message, the second parameter is a string for a hard-coded validation message (pass an empty string for automatic validation messages based on the model attributes and data types) and the third parameter is an optional un-typed list with HTML attributes. In this case the Bootstrap CSS class **text-danger** has been applied to give the text a red color.

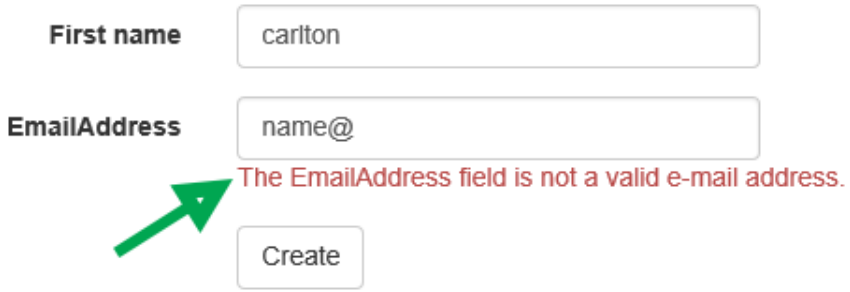
Automatic validation can be used with model properties decorated with attributes such as **Reqired**, **EmailAddress**, **MinLength**, **MaxLength** and many others.

Razor syntax example:

```
@Html.ValidationMessageFor(m => m.Name, "",new{ @class = "text-danger"})
```

Model code example:

```
[EmailAddress]
[Required]
public string EmailAddress { get; set; }
```

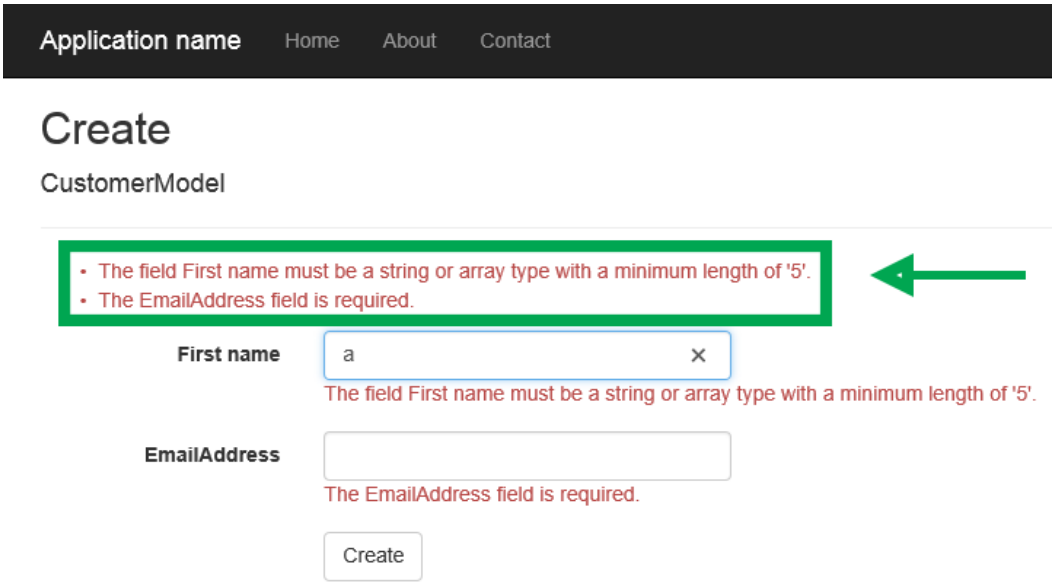


ValidationSummary

This helper will display a validation summary for all validation errors that occur when validating an input form. The first parameter determines if property validation errors should be displayed (the default value is **true**), the second parameter is a hardcoded validation message, the third parameter is an optional un-typed list with HTML attributes. In this case the Bootstrap CSS class **text-danger** has been applied to give the text a red color.

Razor syntax example:

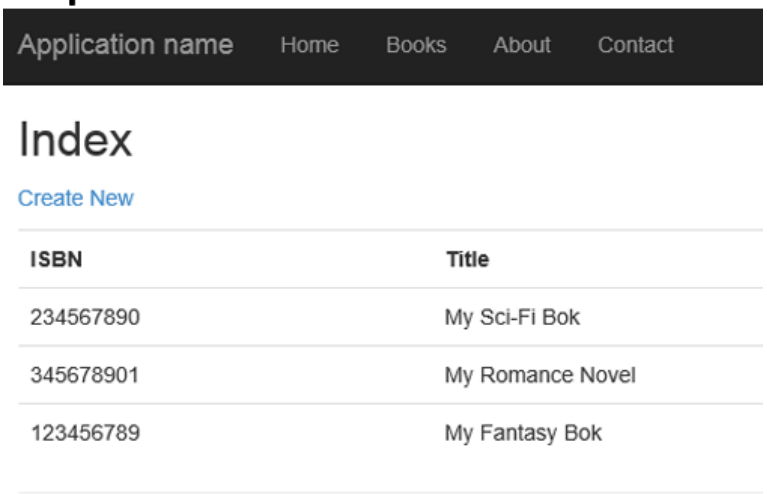
```
@Html.ValidationSummary(false, "", new { @class = "text-danger" })
```



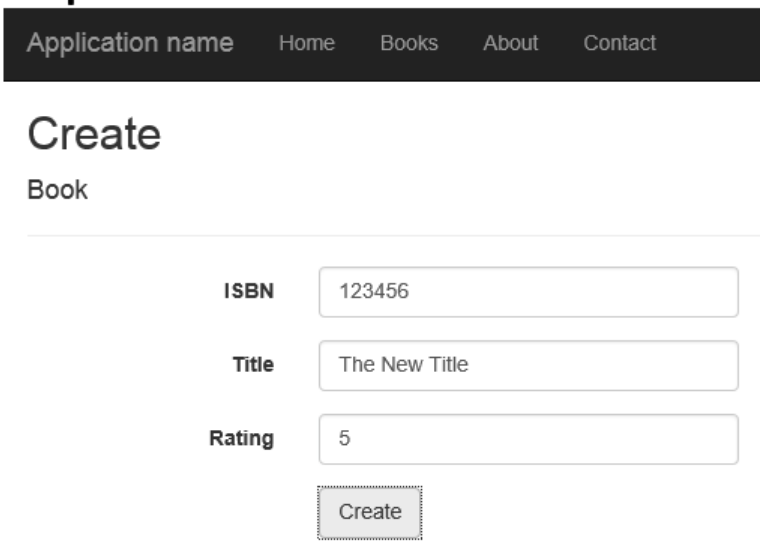
Exercise 4-4: Add a Create view and update the books list

In this exercise you will add a **Create** view for the **Create** action method in the **Books-Controller**. The action will check that it is possible to update the model before saving the new book to the booklist.

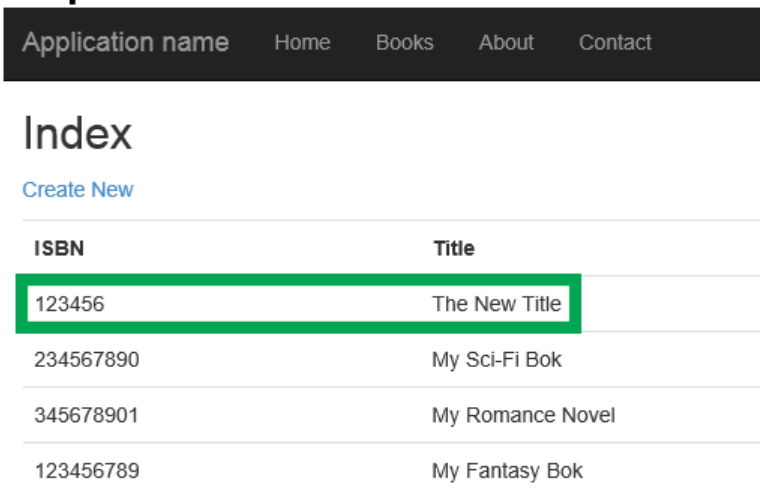
Step 1



Step 2



Step 3



Create the view

1. Open the **BooksController** class.
2. Right click on the **Create** action method and select **Add View**.
3. Name the view **Create**.

4. Select **Create** in the **Template** drop-down.
5. Select the **Book** model in the **Model class** drop-down.
6. Remember to uncheck the **Create as a partial view** checkbox.
7. Click the **Add** button to create the view.

Update the books list

1. Open the **BooksController** class.
2. Make the book list static by adding the **static** keyword to its declaration. A static variable will keep its value between posts because only one instance is ever created for the class. *Note that this is not according to best practices and should be avoided in a real world scenario.*
3. Expand the **Create** action method.
4. Change the **FormCollection collection** parameter to **Book book**:

```
public ActionResult Create(Book book)
```

5. Add an **if**-statement to the **try**-block in the action. The **if**-statement should call the **TryUpdateModel** method to see if it is possible to update the model. Update the **books** list if it succeeds and return the view without changes if it fails. Make sure that the **RedirectToAction** call is inside the **if**-block placed after the new book has been added to the **books** list.

```
if (TryUpdateModel(books))
{
    books.Add(book);
    return RedirectToAction("Index");
}
return View();
```

Add a new book with the Create view

1. Run the application and click the **Books** link in the menu.
2. Click the **Create New** link above the book list to add a new book with the **Create** view you just added.
3. Fill out the form and click the **Create** button.
4. When the book has been successfully added a redirect should take you to the **Index** view where the newly added book will appear in the list.
5. Close the application.

Partial Views

A partial view is a special view which can be reused in multiple views and/or to clean up the code inside a view. Let's say that you want to replace the following **foreach** loop with a call to a partial view containing the same code and make it possible to reuse the table rows it emits from other views.

```
@foreach (var item in Model) {
    <tr>
        <td>@Html.DisplayFor(modelItem => item.ISBN)</td>
        <td>@Html.DisplayFor(modelItem => item.Title)</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Details", "Details", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}
```

First you select and cut out the code inside the **foreach** loop leaving only an empty loop. Then you create a partial view making sure that it has the same type as the model in the loop and paste the cut-out code into the partial view. When naming a partial view it is common practice to begin the name with an underscore character (_), for instance *_Book*. Many developers like to be more explicit about the naming and add *Partial* to the end of the name *_BookPartial*.

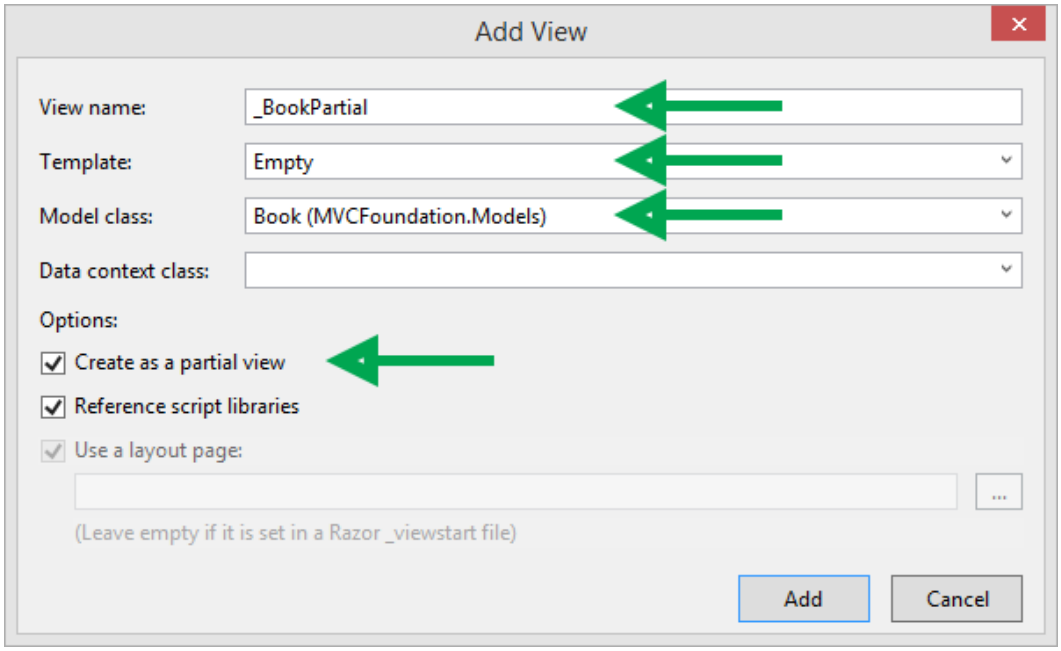
The partial view has to be rendered inside the empty **foreach** loop. Call the **Partial** HTML helper to render the partial view. In this example the **Partial** method require two parameters the name of the partial view and the **item** object from the loop. You do not specify the file extension for the partial view in the **Partial** method call.

You can only pass existing data to a partial view. This mean that you can pass in the whole view model or a subset of the view model to the partial view; in this example a subset of the view model is passed to the partial view.

```
@foreach (var item in Model)
{
    @Html.Partial("_BookPartial", item)
}
```

How to create a partial view

1. Right click on the view folder where you want to create the partial view and select **Add-View**.
   *If you crate it in a specific controller's views folder the partial view will only be reachable from views within that folder. If you want to use the partial view from any view in the application then it has to be created it in the **Shared** views folder.*
2. Specify the partial view name beginning with an underscore character in the **View name** field.
3. If you have code to paste in to the partial view then select **Empty** in the **Template** drop down list else use one of the pre-defined templates to scaffold out the partial view.
4. Select the class you want the partial view to have as its model in the **Model class** drop down list.
5. Check the ***Create as partial view*** check box.
6. Click the **Add** button to create the partial view.
7. Paste in the markup you want the partial view to contain or write new markup.
8. Save the partial view.



How to call a partial view

1. Add a call to the **@Html.Partial** helper method.
2. Pass in the name of the partial view as the first parameter.
3. Pass in the object you want the partial view to use as the second parameter. This is an optional parameter because you do not always want to pass in an object to a partial view.

```
@Html.Partial("_BookPartial", item)
```

Partial view without a view model

What if you want to render something that isn't part of the view model or if a model don't exist which is the case with the **_Layout** view. Because the **_Layout** view is used with all views it does not have a model, you can use the **@Html.Action** helper method to render a partial view from the **_Layout** view or a view that don't have a model. The **@Html.Action** method calls an action on the specified controller as a sub-request, you can then create a separate model in the action method and the partial view will be rendered using that model.

Important: *It is important to mention that the **@Html.Action** method is executed like a sub-request on the server and won't generate another roundtrip to the server.*

Keep in mind that a **public** method created in a controller can be called directly from the browser. To prevent this from happening you can decorate action methods with the **ChildActionOnly** attribute. When that attribute has been added the method can only be called from helper methods like **@Html.Action**, calling it in any other way from the browser will generate an error.

Let's say that you want to display the first book in the book list in the footer of every page. I know it don't make any sense to display the first book on every page but this example is meant to show how you can call an action method to render a partial view. What that partial view contain will differ from case to case. We will revisit the **@Html.Action** in the next chapter when Bootstrap is explored.

The first thing you need to do is to create the **GetFirstBook** action method in the **Books** controller and select and return the first book in the book list. When returning from the

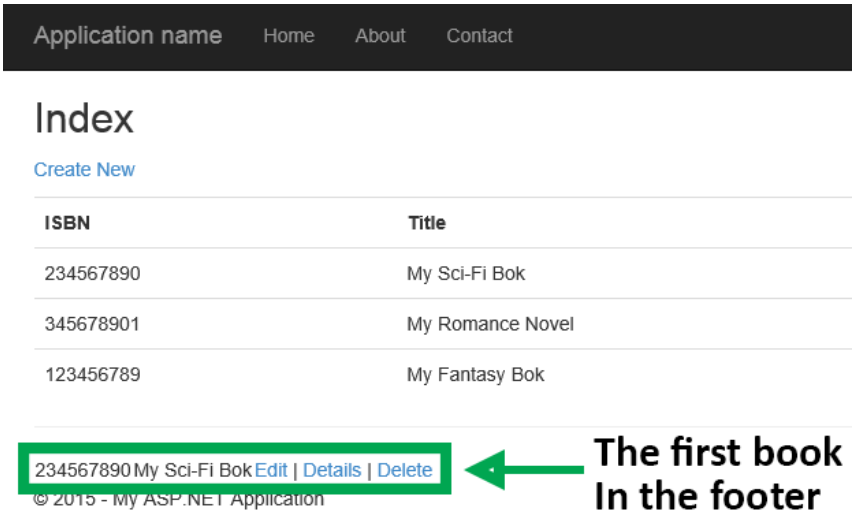
action you need to specify that it is a partial view being rendered by calling the **Partial-View** method with the name of the partial view and the book object.

```
public ActionResult GetFirstBook()
{
    var book = (from b in books
                orderby b.Title descending
                select b).FirstOrDefault();

    return PartialView("_BookPartial", book);
}
```

Next you have to call the action method from the **<footer>** section of the **_Layout** view. Note that the **_BookPartial** partial view emit table rows which mean that you have to place the **@Html.Action** call inside a **<table>** element.

The first parameter of the **@Html.Action** method is the name of the action method you want to call and the second is the name of the controller in which the action method resides.

```
<footer>
    <table>
        @Html.Action("GetFirstBook", "Books")
    </table>
    <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
</footer>
```



Exercise 4-5: Create and call a partial view

In this exercise you will create a partial view called **_BookDetailPartial** and render it from the **_Layout** view by calling the ***@Html.Action*** method which in turn will call an action method returning the partial view to the **_Layout** view displaying the information on every page.

You will begin by adding a **Rating** property to the **Book** model class to be able to determine which book is the highest ranked. The values will be used to find the top ranked book and return it as a partial view from the **GetBestBook** action method in the **BooksController**.

Change the Book model

1. Click on the **Book** class in the **Models** folder.
2. Add a property of type **int** called **Rating**.

```
public int Rating { get; set; }
```

3. Save the model.
4. Click on the **BooksController** class in the **Controllers** folder.
5. Modify the **Books** list to add values for the rating property.

Add the GetBestBook action method

1. Open the **BooksController** if it's not already open.

2. Add a method called **GetBestBook** which return an **ActionResult**.

```
public ActionResult GetBestBook() { }
```

3. Fetch the book with the highest rating and store the book in a variable called **book**.

```
var book =  from b in books
            orderby b.Rating descending
            select b;
```

4. Return the book by calling the **PartialView** method and pass in the name of the partial view and the book you fetched from the list.

```
return PartialView("_BookDetailPartial", book.FirstOrDefault());
```

5. Save the controller class.

Create the partial view

1. Right click on the **Books** folder in the **Views** folder and select **Add-View**.
2. Name the view **_BookDetailPartial**.
3. Select **Empty** template.
4. Select the **Book** model class.
5. Remember to check the **Create as a partial view** checkbox.
6. Click the **Add** button to create the partial view.
7. A partial view with one line of code defining the model class will be created.

```
@model MVC_Foundation.Models.Book
```

Create the partial view content

Here you will get a sneak peek at the Bootstrap grid layout giving you a taste of what's to come in the next chapter. Don't worry too much about the classes you are adding to the ** ** elements you will look at them in detail in the next chapter, for now just add them and watch the result.

1. Add a ** ** decorated with the **row** class. Add another ** ** inside the first ** ** decorated with the **col-md-12** class.
2. Add an **<h4>** element inside the innermost ** ** to make the text a little larger. Fetch the title from the model using the **@Model** object inside the **<h4>**element.

```
<div class="row">
   <div class="col-md-12"><h4>@Model.Title</h4></div>
```

```
</div>
```

3. Add another row under the previous row in the same way you did in the previous two bullets then add two column ** ** elements with the classes **col-md-2** and of **col-md-10** respectively.
4. Write *ISBN:* and add the **ISBN** from the **@Model** object inside the first column ** **.
5. Write *Rating:* and add the **Rating** from the **@Model** object inside the second column ** **.

```
<div class="row">
    <div class="col-md-2">ISBN: @Model.ISBN</div>
    <div class="col-md-10">Rating: @Model.Rating</div>
</div>
```

6. Save the partial view.

Add the @Html.Action call to the _Layout view

To display the highest ranked book on every page you will add it to the **<footer>** element in the **_Layout** view.

1. Open the **_Layout** view located in the **Shared** folder.
2. Add the call to the **@Html.Action** method inside the **<footer>** element passing in the name of the action method and the name of the controller in which the action method resides.

```
<footer>
    @Html.Action("GetBestBook", "Books")
</footer>
```

3. Save the view.
4. Run the application.
5. The highest ranked book should be displayed at the bottom of every page as you navigate between them using the menu.
6. Close the application.

5. Introduction to Bootstrap

Introduction

Bootstrap is a free open-source frontend toolkit originally developed by Twitter for designing and building web applications using HTML, CSS and JavaScript. It has become very popular with web developers and designers because of its flexibility and ease of use. By using Bootstrap you can have a well designed and fabulous looking web site in no time at all because it contains the foundation you need for buttons, labels, badges, tables, layout, typography, forms, widgets (like picture carousels) and much more.

Another benefit is that the web pages will look great on any device because Bootstrap is designed with mobile first in mind. The layout is determined by the resolution of the device displaying the web pages. This saves you the time having to create a design for mobile devices once the design for the main web site is finished.

Being built on open standards such as HTML, CSS and JavaScript it can be used with any platform, server technology and editor.

A prerequisite for using Bootstrap is that you have basic skills in HTML, CSS and Java-Script; don't worry you don't have to be an expert on those topics when you begin using Bootstrap you will learn the necessary skills as you progress through this book.

Bootstrap is a framework containing well factored single responsibility CSS classes which mean that each class has a single purpose. Instead of creating a traditional large and bulky CSS class which styles an entire component, such as a table, Bootstrap gives you many small classes all with a specific purpose, from which you can pick-and-choose when styling your components.

Apart from CSS, Bootstrap also comes with a JavaScript library for the more complex components and widgets such as picture carousels. This library is only required if you use components and widgets that depend on JavaScript.

When creating an MVC Web Application in Visual Studio 2015 a default Bootstrap theme is already included in the project providing you with styled web pages from the get go. If

you don't like the default theme you can easily download another for free at sites like *www.bootswatch.com* or *www.bootstrapzero.com*. There is also a plethora of other free and paid sites where you can find really nice themes. It can be a real time saver to have a finished theme to begin with and tweak it a little to make it perfect for your web site.

To get started you only need to learn a few core Bootstrap classes and once you have mastered those the rest are easy to learn.

Bootstrap provides CSS classes for layout and to style controls, JavaScript functions for widget functionality and over 250 glyphs optimized in a few font files. A glyph is a symbol described as a scalable font which can be incorporated into text and components displayed on the web page.

In a MVC project the files are located in three different folders. The JavaScript files are located in the **Scripts** folder, the CSS style sheets are located in the **Content** folder and the glyphs are located in the **fonts** folder. As you can see in the image below the JavaScript and CSS files have two versions ending with *.js* or *.css* and *.min.js* or *.min.css*. The *.min* files are **minified** versions of the original files made smaller by removing unnecessary characters, spaces and comments. Using minified versions will make the page load faster and create a better user experience. You don't have to do anything for the minified versions to be used in Visual Studio projects because they are bundled up and minified automatically.

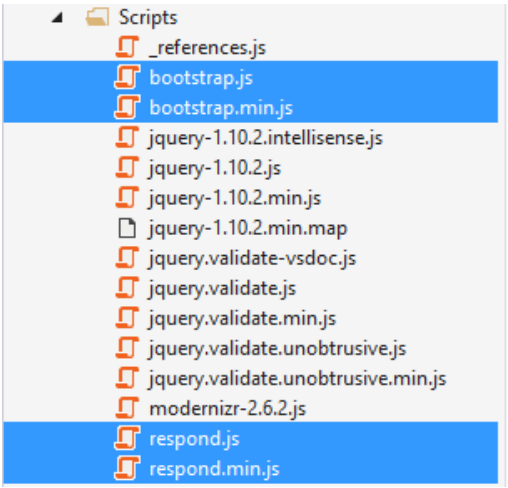
Important: *It is also important to note that the Bootstrap JavaScript files are dependent on the JQuery library which is installed by default in MVC projects.*

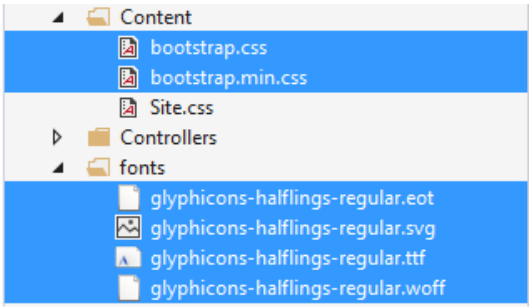
The *bootstrap.js* file handles JavaScript for widgets and components which rely on code to work properly. Although the *respond.js* file isn't part of Bootstrap it is worth mentioning here because it's a *polyfill* that provide alternate solutions for browsers that don't support specific features.

Technologies used in this chapter

- **HTML 5** - HTML elements are used when illustrating how Bootstrap works.
- **Bootstrap** - Used to style HTML elements and the web site.

Scripts & CSS





The CSS classes implemented by Bootstrap have meaningful descriptive names and do not contain any implementation detail. An example of a very specific (and perhaps not so well thought out) CSS class name for displaying an error text is **red-text**. The name does not say in what context it should be used or why it is named that way, is it because it denotes an error or simply that you enjoy the color red?

By using a name like that you have essentially created a class which only can be used to display text with a red foreground color, if you wanted to change the color (for error messages) at a later time you would have to change the name of the class and all places in the code where the name is used throughout the entire application.

Instead Bootstrap provides classes for displaying text for different scenarios or contexts such as **success** (green), **warning** (orange), **danger** (red) and **info** (light blue). Note that the class names don't contain any specifics about the default color associated with them. This strategy is deliberate because it makes it very easy to override the behavior of a class without changing its name. If you want to change the text color for warning messages in your application you simple override that name and change the color in a CSS file which instantly changes the color in the entire application.

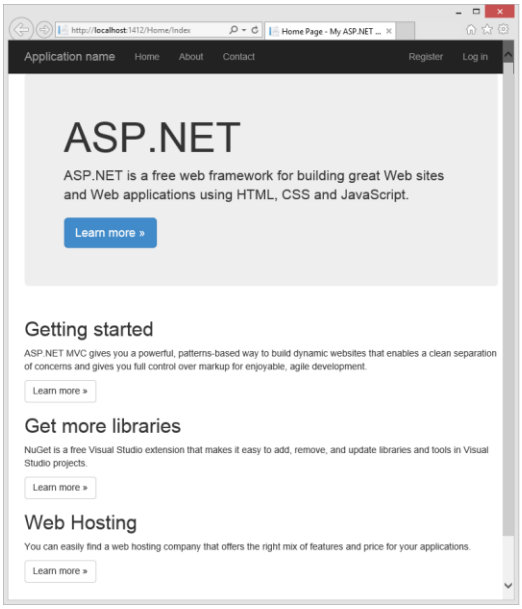
```
.red-text {
    color:red;
}
```

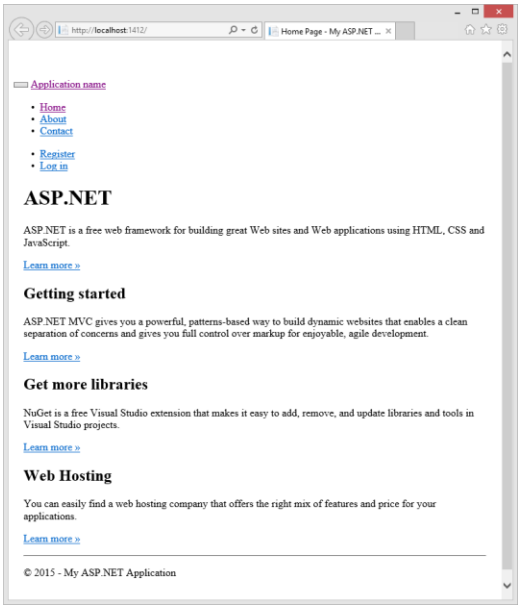
```
.danger {
    color:red;
}
```

```
<span class="red-text">This is a warning!</span>

<span class="danger">This is a warning!</span>
```

Another example of a poorly chosen CSS class name is **left-side** for a content area which probably will appear to the left side on the page, if we are to believe the class name. But that is not a given, the CSS could be changed to display the area on the right side of the page but still have the old name and thus creating confusion. A better name would be **side-bar** which don't specify exactly where the area will appear, only that it is to be displayed to one of the sides of the page, left or right.

To give you an example of what Bootstrap really do with the layout of the web pages out-of-the-box I have commented out the *bootstrap.css* reference in the *BundleConfig.cs* file where it is defined as part of a CSS bundle which is loaded when the web application starts. Below you can see the page displayed with and without Bootstrap present. Without Bootstrap the page get a retro feel taking you back to the early days of internet.





Typography

Looking at the image above you can clearly see that the page not only is better looking with Bootstrap, but the font family has changed as well to a cleaner and more readable sans-serif. If you aren't happy with the default Bootstrap font you can always override it in the *site.css* file and specify a new font family. If you want to change it for every rendered element in the **body** segment of the page you could add a different font family to the **body** selector overriding the font family defined by Bootstrap.

```
body {
    font-family:'Franklin Gothic Medium', 'Arial Narrow', Arial,
    sans-serifArial;
}
```

Glyph Icons

Instead of using images for icons, which can add to the page load time, you can opt to use glyph icons which are defined as scalable fonts. There are over 250 glyphs available in Bootstrap alone but there are other providers as well.

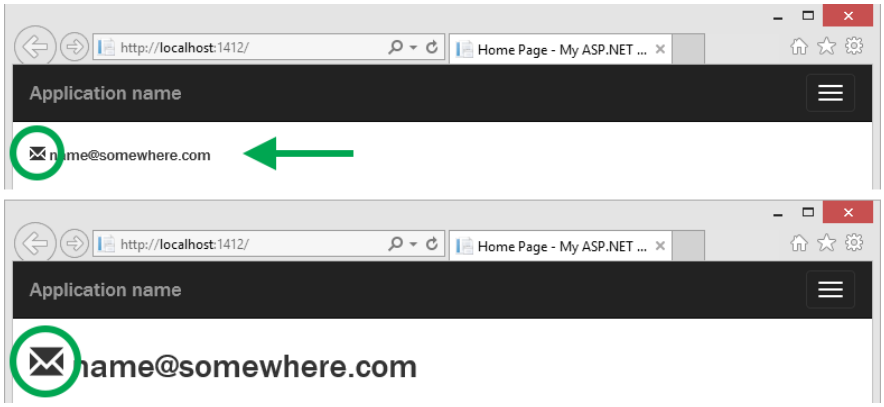
You can easily add a glyph to a control or text to make it stand out more and describe the purpose of the control with an illustration. You can see all the available glyphs, CSS classes and widgets provided by Bootstrap at *http://getbootstrap.com/components/*; the site also contain basic examples on how to use them.

Let's say you want to display an envelope icon to the left of an email address. Instead of uploading an image of an envelope with the solution you can add a glyph to the paragraph.

The one rule you must adhere to when using glyphs is that they have to be added as classes to an empty **** element. When using most Bootstrap classes you have to specify a main class describing the control or its main usage (in this case **glyphicon**) as well as a defining class (in this case the **glyphicon-envelope** class). The **glyphicon** class specifies that a glyph is being added and the **glyphicon-envelope** class which glyph to display.

```
<p>
    <span class="glyphicon glyphicon-envelope" aria-hidden="true"></span>
    name@somewhere.com
</p>
```

As you can see in the image below that the glyph scales really well when changing font size.



Font Awesome

There are other providers of glyphs that you might want to take a look at. One is *Font Awesome* which is widely used in the industry, it can be downloaded from *GitHub* at *http://fontawesome.github.io/Font-Awesome/.*

Important: *It can be tricky to get Font Awesome to work with applications uploaded to Azure.*

Grid Layout and Responsive Design

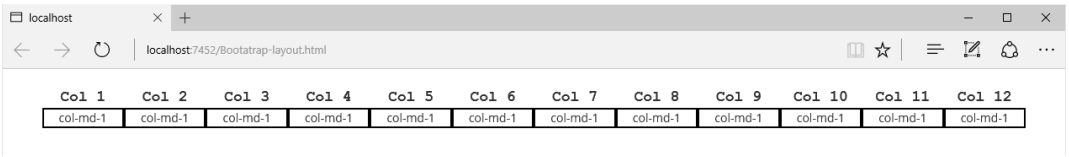
Grid layouts are not a new invention for web pages they have been around since the Sumerians first started taking down inventory on clay tablets. One of the features of using a grid layout is that it gives structure. It helps organize the data in rows and columns and to enforce margins. It is also easier for the user to take in the information since our eyes have been conditioned to move from left to right or right to left when reading information on a page.

Grids are used to create order, we love them so much that we often place grids inside of grids. This type of layout lends itself for browsers since every component on a web page is made up of rectangular elements. No matter how you style an element the browser will still considered it a four sided element.

The Bootstrap grid layout helps you by providing a container which centers the content on the page and structures the content in rows and columns. You can have as many rows as you need and up to 12 columns per row. It is also possible to nest a grid inside of a cell; the nested grid can then have up to 12 columns taking up the full space of the cell it is nested within.

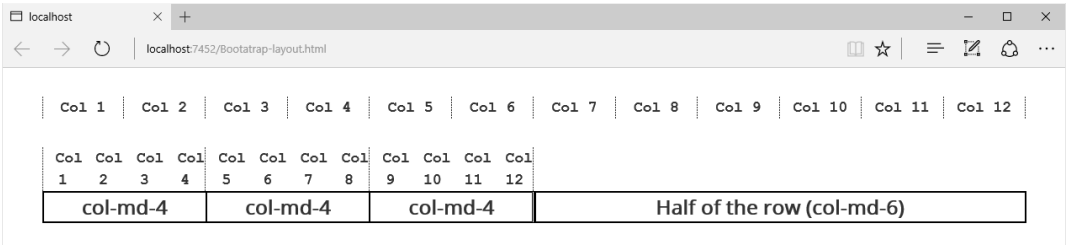
If not all columns of a row are used an empty area representing the remaining unused columns will be displayed.

The content flows within each column moving a cell's overflowing content to a new row within that element.



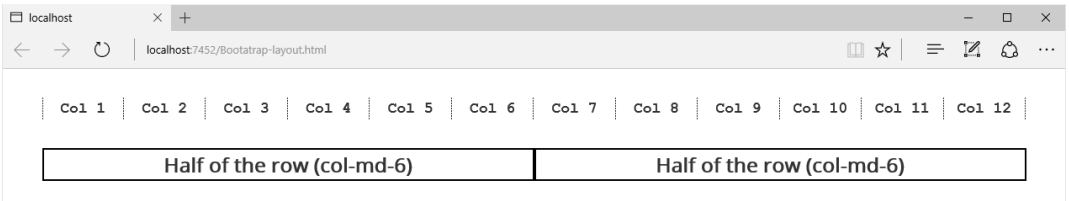

```
<div class="row">
    <div class="col-md-1">Col 1</div>
    <div class="col-md-1">Col 2</div>
    @*... 10 more columns*@
</div>
```

The second row in the grid is divided into two equally sized columns each 6 units wide where the left column is divided into 3 equally sized columns each 4 units wide. This is accomplished by nesting a second row divided into 3 columns inside the outer column effectively dividing the available 50% of the outer row into 3 columns taking up 33% each. Remember that the total number of outer columns per row must not exceed 12. The outer row consists of two columns each 6 units wide where the first column is divided into 3 columns each 4 units wide which adds up to 12 columns.



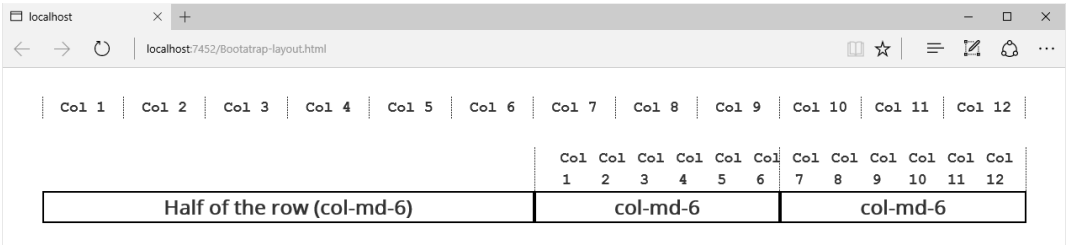

```
<div class="row" >
    <div class="col-md-6">
        <div class="row">
            <div class="col-md-4">4 units</div>
            <div class="col-md-4">4 units</div>
            <div class="col-md-4">4 units</div>
        </div>
    </div>
    <div class="col-md-6">Half of the row (6 units)</div>
</div>
```

The third row in the grid is divided into two columns each 6 units wide.




```
<div class="row">
    <div class="col-md-6">Half of the row (6 units) </div>
    <div class="col-md-6">Half of the row (6 units) </div>
</div>
```

The last row in the grid is divided into two equally sized columns each 6 units wide where the second column is divided into two columns each 6 units wide.




```
<div class="row">
    <div class="col-md-6">Half of the row (6 units)</div>
    <div class="col-md-6">
        <div class="row">
            <div class="col-md-6">6 units</div>
            <div class="col-md-6">6 units</div>
        </div>
    </div>
</div>
```

The **row** class is used when creating new rows in the grid and by doing so the content below that row will be pushed down the page. Each row can have up to 12 columns which you create by using the **col-** classes, there are several different **col-** classes targeting different device resolutions (see table below).

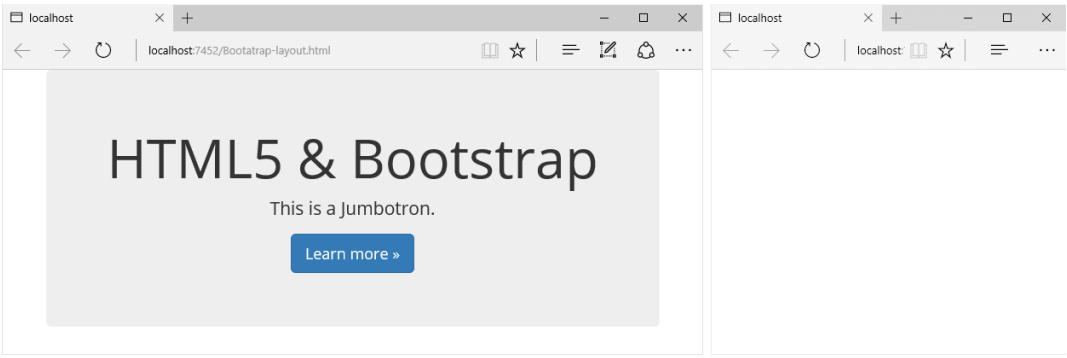
This mean that you can create different page layouts depending on the screen resolution you are targeting. You might want to display content in one way for a full screen desktop computer and in a completely different way on a smaller device. By targeting the device

resolution using the **col-** classes you can achieve different layouts for different devices. If you only specify a certain device resolution using the **col-** classes then the unspecified resolutions will be displayed using the default Bootstrap layout for those resolutions. For instance when the resolution falls below 992px the content begin to stack vertically which might not be the layout you want. You can change this behavior either by specifying that it should use the same layout for small, medium and large device types or you can provide a unique layout for a specific scenario.

You can also hide content for certain resolutions. Let's say that you only want to display a picture banner on devices with a resolution greater 992px and not on smaller devices such as mobile phones. To achieve this you can either use the **hidden-xs** and **hidden-sm** classes to hide the content on those types of devices or use **visible-lg** and **visible-md** to only show the content on medium and large sized devices.

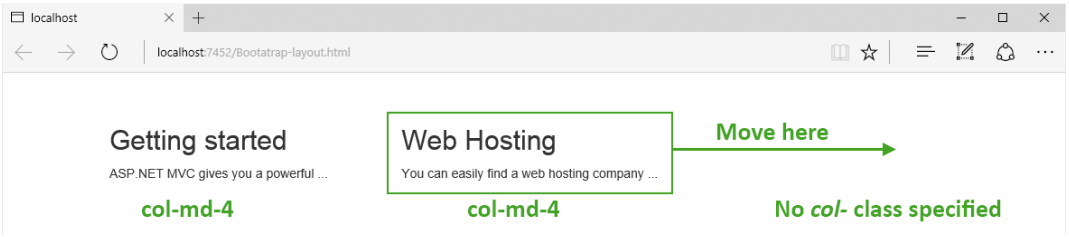
In the example below the **Jumbotron** (the gray area) is hidden on extra small devices using the **hidden-xs** Bootstrap class which mean that it will be hidden when the resolution is below 768px. You could achieve the same result by adding the **visible-sm**, **visible-md** and **visible-lg** classes to the **Jumbotron**.

```
<div class="jumbotron hidden-xs">
    <h1>ASP.NET</h1>
    <p class="lead">ASP.NET is a free web framework ... </p>
    <p><a href="http://asp.net" class="btn btn-primary btn-lg">
        Learn more &raquo;</a></p>
</div>
```



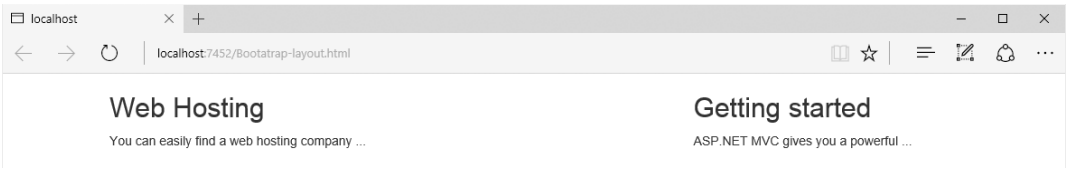
Column class	Resolution	Description
col-lg-*columns*	≥ 1200px	Displays the content on large devices
col-md-*columns*	≥ 992px	Displays the content on medium and large devices
col-sm-*columns*	≥ 768px	Displays the content on small to large devices
col-xs-*columns*	< 768px	Displays the content on extra small to large devices
hidden-*size* (lg, md, sm, xs)		Hides the content for the specified resolution and displays it for other resolutions
visible-*size* (lg, md, sm, xs)		Displays the content for the specified resolution and hides it for other resolutions

You can move content in relation to where it originally was placed in the HTML markup by using the **col-*xx*-offset**, **col-*xx*-push** and **col-*xx*-pull** classes on an element. Let's say you have two grid columns each taking up 4 units of the 12 available. The two pieces of content are flush against the left side of the grid (see image) and now you want to move the right most of the two columns to be flush against the right side of the grid, to achieve this you can use one of the **col-*xx*-offset** classes. Remember not to exceed the maximum 12 column limit.




```
<div class="row">
    <div class="col-md-4">
        <h2>Getting started</h2>
        <p>ASP.NET MVC gives you a powerful ...</p>
    </div>
    <div class="col-md-4 col-md-offset-4">
        <h2>Web Hosting</h2>
        <p>You can easily find a web hosting company ...</p>
    </div>
</div>
```

If you have the same scenario as in the previous example with two columns flush to the left side of the grid and the HTML markup is defined in a specific order that you don't want to change. Your dilemma is that the two columns should appear in the reverse order and displayed flush against the left and right sides of the grid (the same end result as in the previous example but the columns should be reversed). To achieve this you can use the **col-*xx*-push** and **col-*xx*-pull** classes.



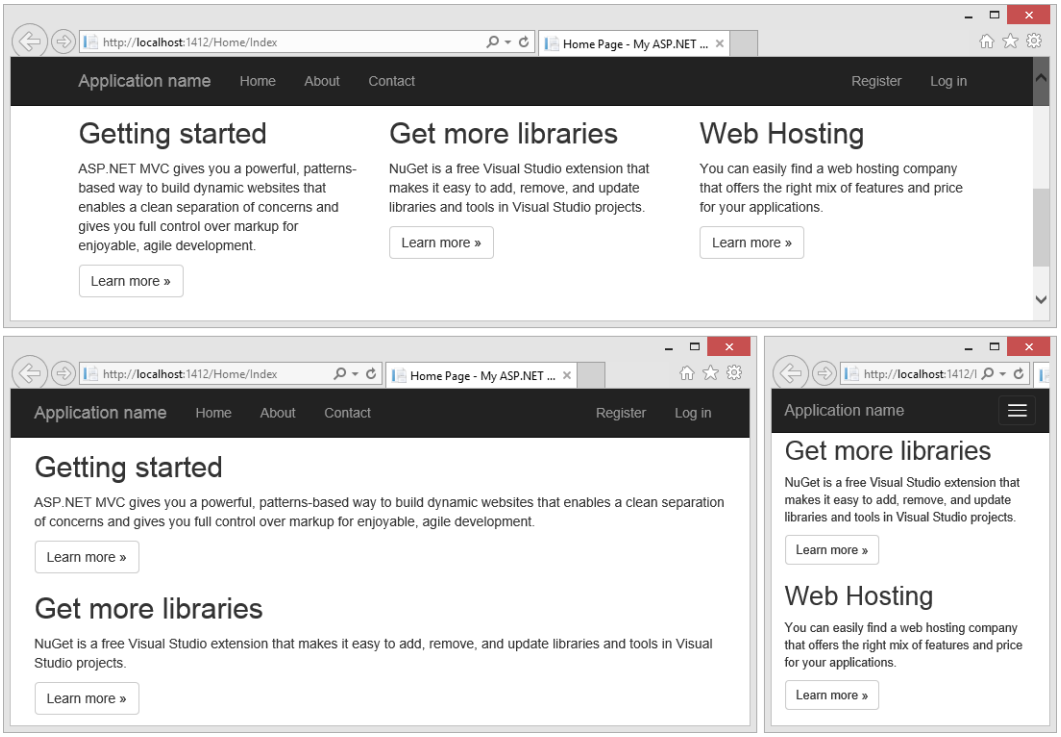

```
<div class="row">
    <div class="col-md-4 col-md-push-8">
        <h2>Getting started</h2>
        <p>ASP.NET MVC gives you a powerful ... </p>
    </div>
    <div class="col-md-4 col-md-pull-4">
        <h2>Web Hosting</h2>
        <p>You can easily find a web hosting company ...</p>
     </div>
</div>
```

Class	Description
col-*size*-offset-*columns* (size: lg, md, sm, xs)	Offsets (moves) the content in relation to where it originally was placed in the grid.
col-*size*-pull-*columns*	Pulls the column the specified number of columns to the left.
col-*size*-push-*columns*	Pushes the column the specified number of columns to the right.

Bootstrap provides responsive design optimizing the output for different devices, on small devices the content will be displayed vertically and on larger horizontally. Once the device drops below a certain resolution the content will automatically be stacked vertically unless you change the layout. The only thing you have to do is to use the correct Bootstrap classes and Bootstrap will style the page asking the device for its capabilities using media queries.

```
@media (min-width: 768px) {
    @*Styles for small devices*@
}
```

When the browser width is reduced and it falls below 992px the content stacks vertically instead of horizontally. If the width is less than 768px the menu changes to the "hamburger menu", the button with three lines which opens a drop down menu.

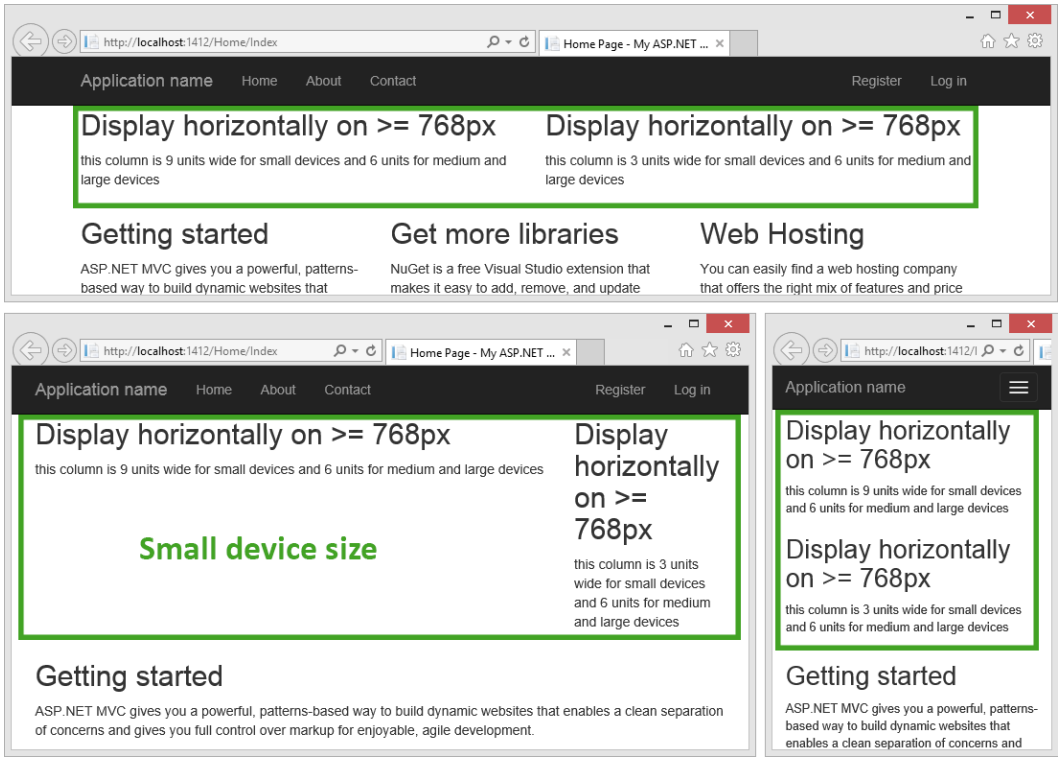


Combining multiple column definitions

It is possible to add more than one Bootstrap column class to an element effectively changing the design when the browser width changes. Let's say that you want to display horizontal content on small devices as well but the design should be different from the design on medium and large devices. This can be accomplished by adding a column definition for small devices overriding the default stacking design with your own horizontal design.

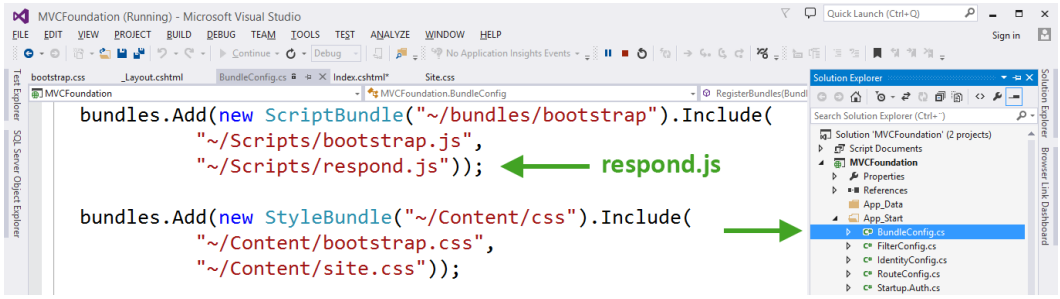
The following example shows how you can change the default column design for small devices to display two columns with 9 and 3 units respectively. The same content will be displayed on medium and large devices using two columns occupying 6 units each.

```
<div class="row">
    <div class="col-sm-9 col-md-6">
        <h2>Display horizontally on >= 768px</h2>
        <p>this column is 9 units wide for small devices and 6 units for
           medium and large devices</p>
    </div>
    <div class="col-sm-3 col-md-6">
        <h2>Display horizontally on >= 768px</h2>
        <p>this column is 3 units wide for small devices and 6 units for
           medium and large devices</p>
    </div>
</div>
```



Respond.js

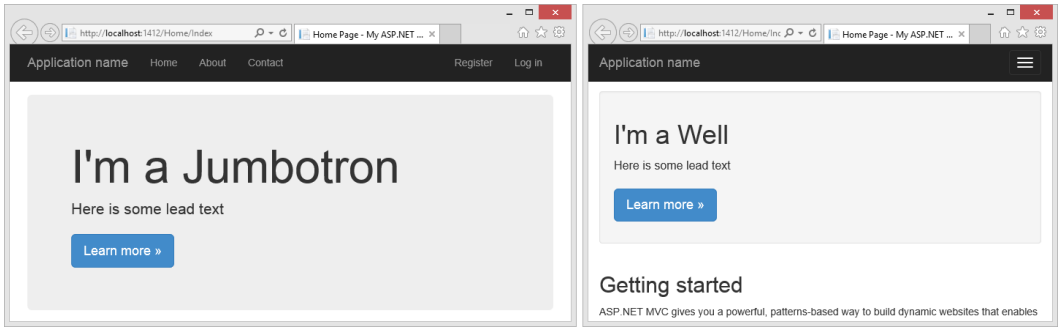
There are older browsers that don't support media queries such as IE6-IE8. To get responsive design to work with media queries in these browsers a JavaScript *polyfill* file called *respond.js* can be used which is bundled with the *bootstrap.js* file in the *BundleConfig.cs* file; it will fill in the missing pieces making it possible for older browsers to use media queries.



Components

There are many classes in Bootstrap which can be applied to HTML elements to style them and make them more appealing to the user. You have already seen two components at work on the default page created in every MVC project in Visual Studio, the navigation bar at the top of the screen (more about that component later in this chapter) and the **Jumbotron** which is the gray area with the large text at the top of the page. The latter component is created by adding the **jumbotron** class to a ** ** element and add some content to that ** **. The **well** class is similar to the **jumbotron** but the text will not be as pronounced.

```
<div class="jumbotron">
    <h1>I'm a Jumbotron</h1>
    <p class="lead">Here is some lead text</p>
    <p><a href="http://asp.net" class="btn btn-primary btn-lg">Learn more
&raquo;</a></p>
</div>
<div class="well">
    <h1>I'm a Well</h1>
    <p class="lead">Here is some lead text</p>
    <p><a href="http://asp.net" class="btn btn-primary btn-lg">Learn more
&raquo;</a></p>
</div>
```



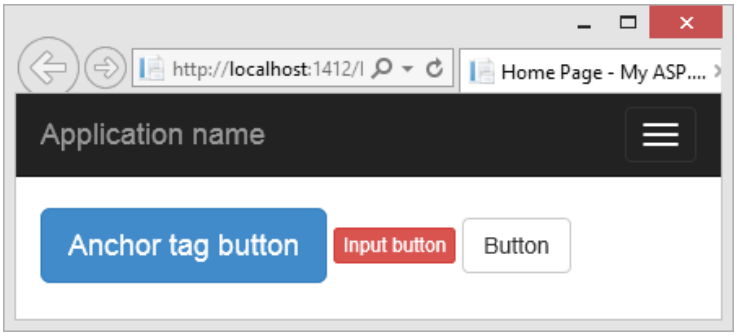
Buttons

Buttons are easy to style, all you need to do is to add the **btn** class and one of the button sizing classes **btn-xs**, **btn-sm**, **btn-md** or **btn-lg** to an element. To give a button color you can use one of the predefined "color" classes **btn-default**, **btn-success**, **btn-primary**, **btn-info**, **btn-warning** or **btn-danger**. Note that no color is specified in the class names, this is

deliberate to make it easy to redefine the color the classes represent. In some design scenarios the primary color might not be the default blue but instead a color that fits the design theme of that web site. Instead of creating new CSS classes to define the colors the designer can simply override the **btn-primary** class and change the color associated with that class in the entire theme. The previous image contain buttons defined with the **btn-primary** class.

The **btn** classes can be used with three HTML elements, the anchor tag **<a>**, the **<input>** element and the **<button>** element.

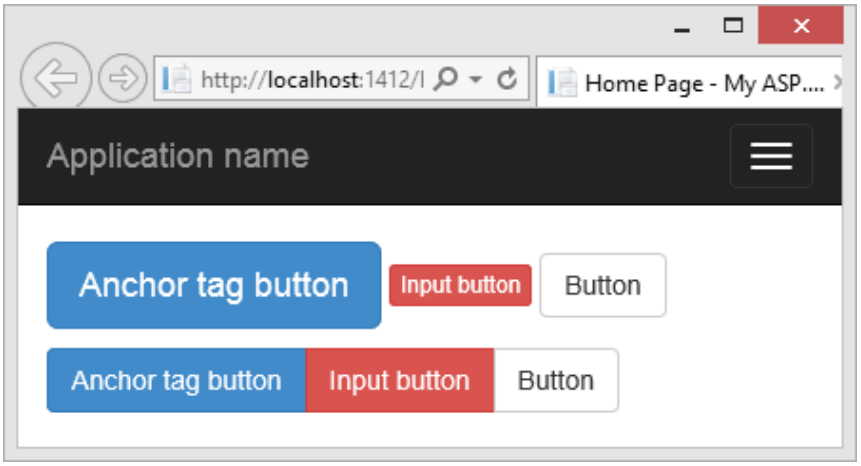
```
<a href="#" class="btn btn-primary btn-lg">Anchor tag button</a>
<input type="button" class="btn btn-danger btn-xs" value="Input button">
<button class="btn btn-default btn-md">Button</button>
```



If you want to display buttons as a group making them the same size instead of displaying them as separate buttons with space in between you can add a ** ** element decorated with the **btn-group** class around the buttons.

As you can see in the image below the button group keep the buttons together and adds rounded corners to the outermost buttons.

```
<div class="btn-group">
    <a href="#" class="btn btn-primary btn-md">Anchor tag button</a>
    <input type="button" class="btn btn-danger btn-md" value="Input
button">
    <button class="btn btn-default btn-md">Button</button>
</div>
```



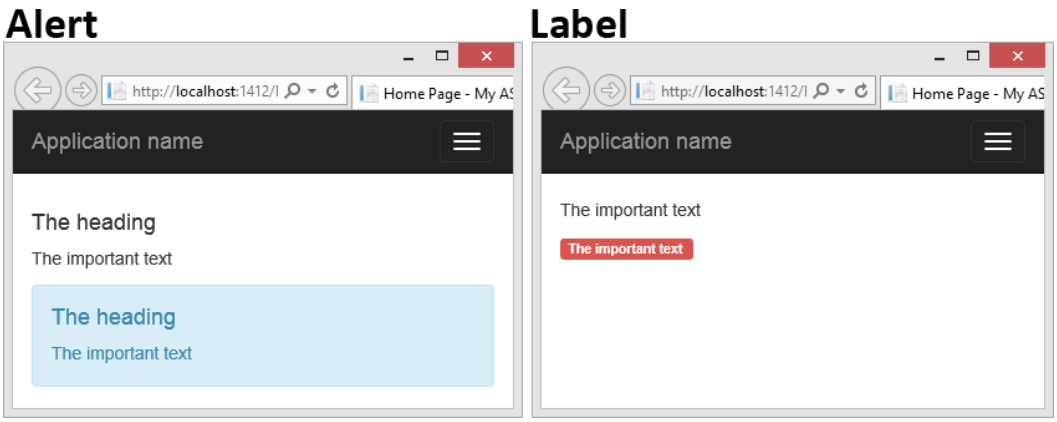
Alert and Label

If you have content that you want to stand out you can either add the **alert** or the **label** class to the surrounding element. As you can see in the image below the **alert** can handle multiple lines of text and even components such as buttons whereas the **label** is a single line of text.

The color is determined by the **alert-info**, **alert-danger**, **alert-success** or **alert-warning** classes for **alert** content or **label-info**, **label-danger**, **label-success**, **label-warning**, **label-primary** or **label-default** for **label** content.

```
@*Alert*@
<div>
    <h4>The heading</h4>
    <p>The important text</p>
</div>

<div class="alert alert-info">
    <h4>The heading</h4>
    <p>The important text</p>
</div>

@*Label*@
<p>The important text</p>
```

```
<p class="label label-danger">
    The important text
</p>
```

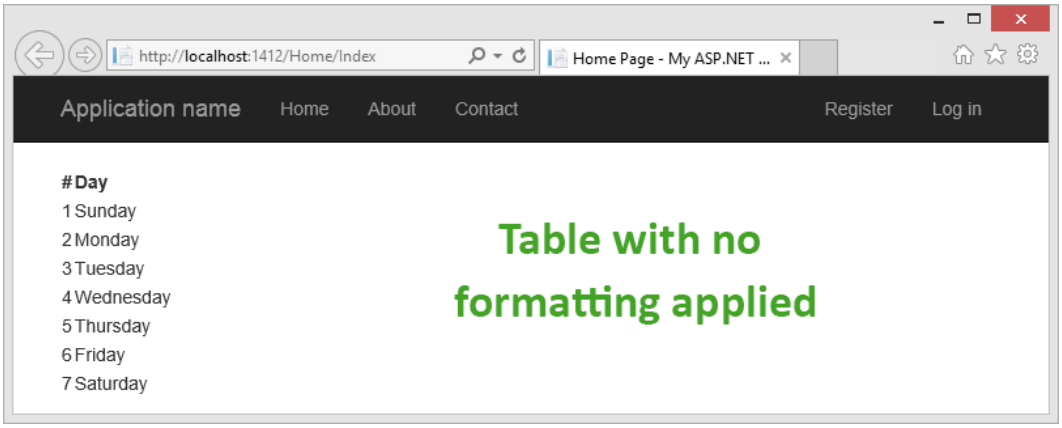


Tables

Tables can be a great way of displaying tabular data but its default styling leaves a lot to be desired. Luckily you can use simple Bootstrap classes to achieve wonders with your tables. In this section you will see some basic table styling. Later you will use this knowledge when you create views for database tables.

The table below is a list of the weekdays where no Bootstrap classes have been added.

```
<table class="">
    @*Header*@
    <tr><th>#</th><th>Day</th></tr>

    @*Rows*@
    <tr><td>1</td><td>Sunday</td></tr>
    <tr><td>2</td><td>Monday</td></tr>
    <tr><td>3</td><td>Tuesday</td></tr>
    <tr><td>4</td><td>Wednesday</td></tr>
    <tr><td>5</td><td>Thursday</td></tr>
    <tr><td>6</td><td>Friday</td></tr>
    <tr><td>7</td><td>Saturday</td></tr>
</table>
```



Basic table styling with Bootstrap

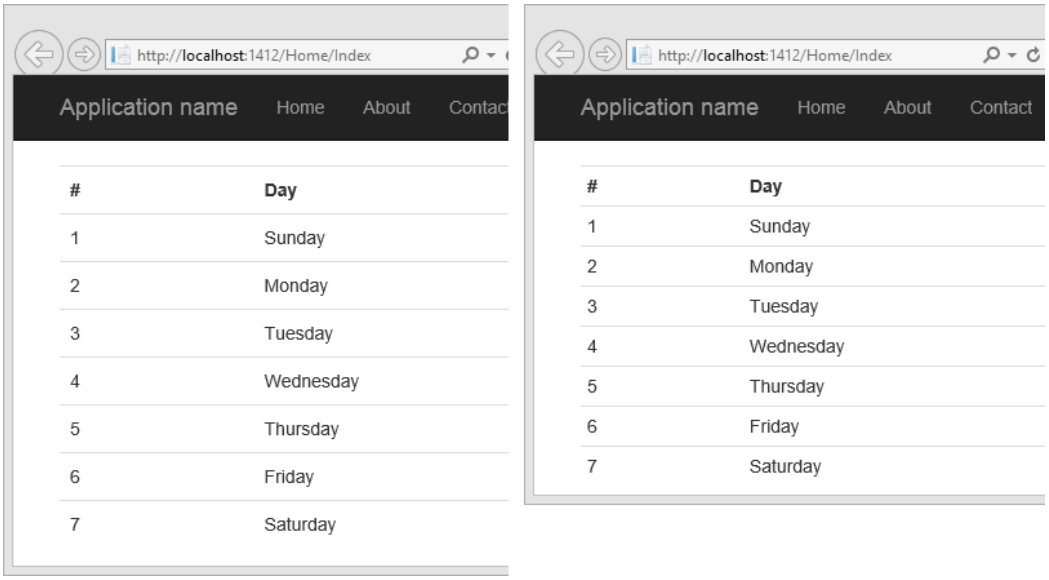
The first Bootstrap class you will add is the **table** class which adds some basic table styling making the columns more airy and display row line dividers.

```
<table class="table">
```

Condensed table

Add the **table-condensed** class to the table if you prefer a more condensed table layout where some of the padding between the lines have been removed.

```
<table class="table table-condensed">
```



Hover effect

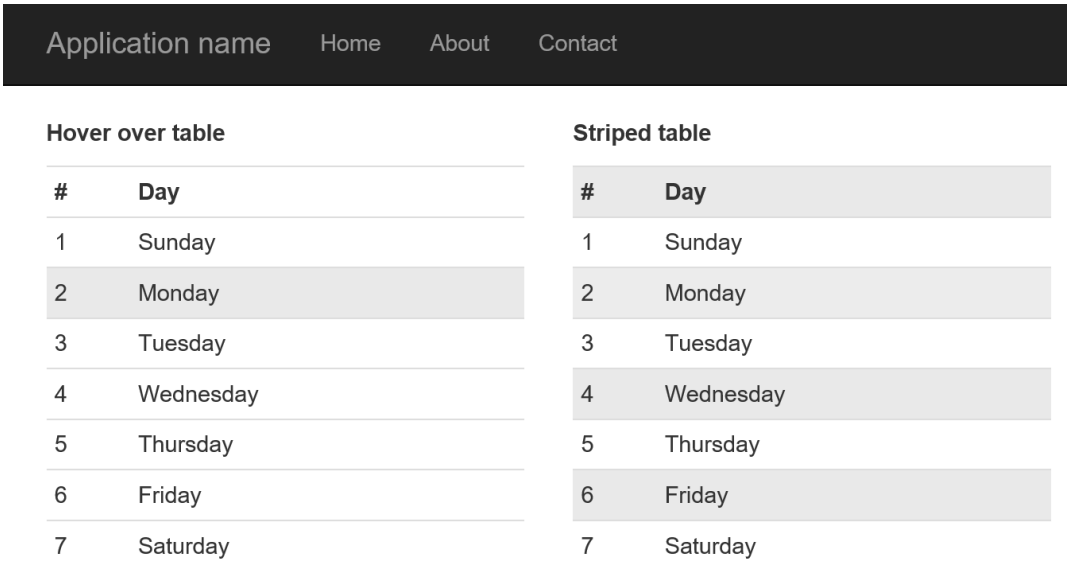
To make it easier for the user to keep track of the rows when reading the data you can add the **table-hover** class which highlights the row where the mouse pointer is hovering. The light gray color used for the hover effect might be hard to see in the image.

```
<table class="table table-condensed table-hover">
```

Striping effect

To make a table more readable you can create a striping effect by adding the **table-striped** class. The light gray color used for the striping effect on the even rows might be hard to see in the image.

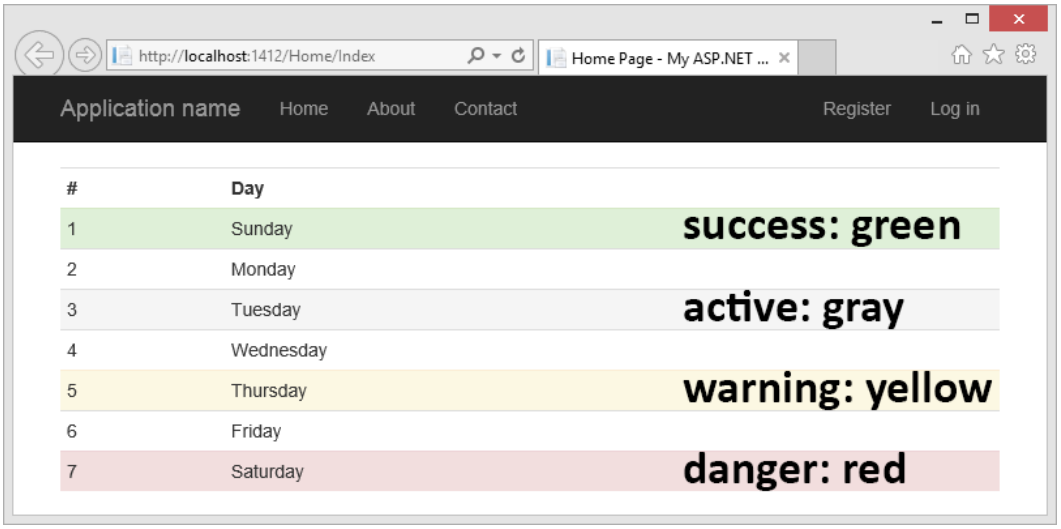
```
<table class="table table-condensed table-striped">
```



Colored rows

If you want a row in the table to have a colorized background one of the **warning**, **danger** and **success** classes can be applied. Or you can use the **active** class to denote that that row is the active row. If you are reading a printed version of this book the table row background colors might be hard to see.

```
<table class="table table-condensed">
    @*Header*@
    <tr><th>#</th><th>Day</th></tr>

    @*Rows*@
    <tr class="success"><td>1</td><td>Sunday</td></tr>
    <tr><td>2</td><td>Monday</td></tr>
    <tr class="active"><td>3</td><td>Tuesday</td></tr>
    <tr><td>4</td><td>Wednesday</td></tr>
    <tr class="warning"><td>5</td><td>Thursday</td></tr>
    <tr><td>6</td><td>Friday</td></tr>
    <tr class="danger"><td>7</td><td>Saturday</td></tr>
</table>
```



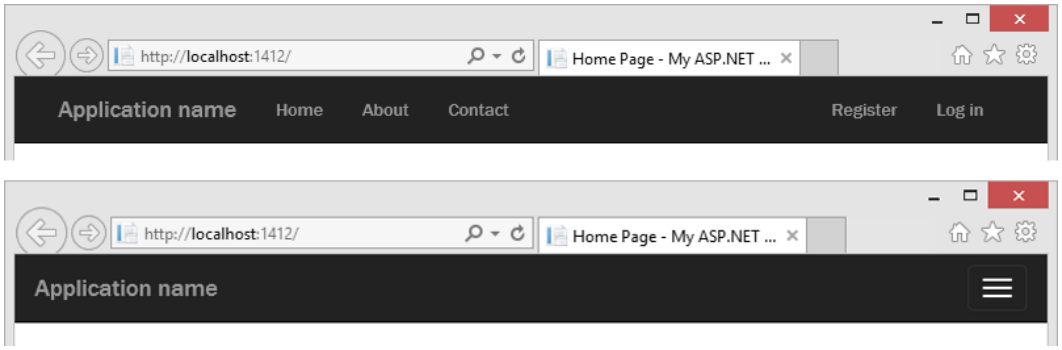
Navigation bar

The navigation bar is the menu that sticks to the top of the browser area as the user scrolls down the page. Let's have a look at the **navbar** classes and find out how this is possible.

The first thing you should note is that if you make the browser small enough the regular menu is replaced by a mobile friendly menu affectionately called *the hamburger menu*; clicking the button opens the menu. This is available out of the box when adding the *bootstrap.css* file to the project.

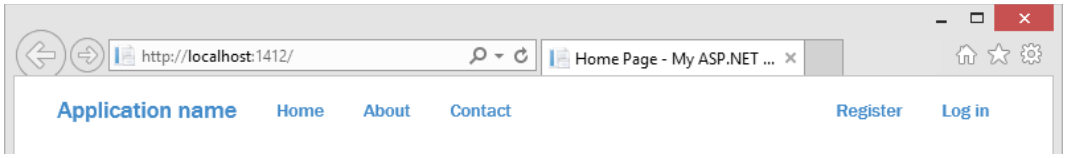
The default implementation of the **navbar** uses three classes. The **navbar** class specifies that the ** ** element should be used as a navigation bar, the **navbar-inverse** specifies that the color should be the inverse of the page background and the **navbar-fixed-top** makes the **navbar** stick to the top of the page even when users scrolls the page content.

```
<div class="navbar navbar-inverse navbar-fixed-top">
```



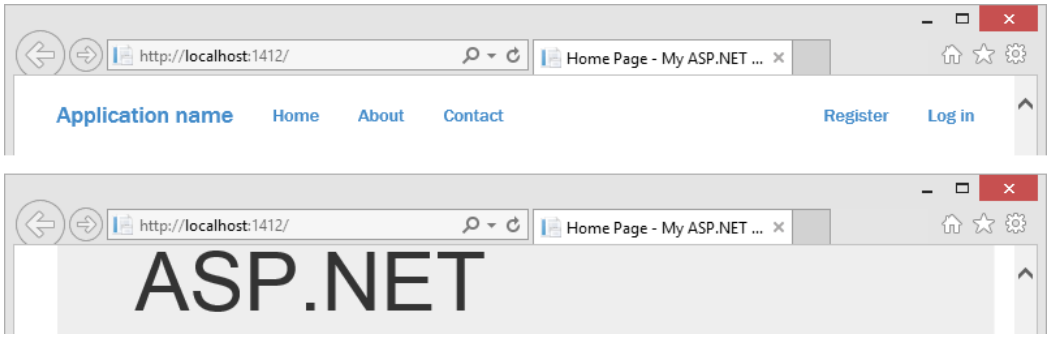
When removing the **navbar-inverse** class the **navbar** is no longer displayed with the inverse background color.

```
<div class="navbar navbar-fixed-top">
```



When removing the **navbar-fixed-top** class the **navbar** will scroll out of view when the user scrolls the page content.

```
<div class="navbar">
```



The structure

The **navbar** is a fairly advanced control which has several areas you need to keep track of as you build or modify a menu. Below is a quick look at the structure needed to create a

rich navigation experience for your users. In upcoming chapters you will get firsthand experience modifying and creating menus using the **navbar**.

The outermost menu element must have at least the **navbar** class present for it to act as a navigation menu. An element with the **container** class applied must be added to the **navbar** element, this element will act as the container for the menu header and the actual drop-down menu.

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
```

The first element inside the **container** element is the **navbar-header** which contains the non-menu item parts of the menu such as the *hamburger menu* mentioned earlier and also the **navbar-brand** area where you specify the name of your brand, company or site.

Adding the **navbar-toggle** class and two **data-** attributes to a **<button>** element makes it possible to open and close the *hamburger menu* when the menu is viewed on a device with a small screen.

The three **icon-bar ** elements are the three lines displayed on the *hamburger menu* button.

The **ActionLink** extension method displays the brand name and turns it into a link. The first parameter is the brand name or text to display in the brand area of the menu, the second is the action method to call when the brand name is clicked, the third is the name of the controller in which the action resides. The fourth parameter is an object containing optional route values to send with the request and the last parameter is an object containing optional HTML attributes you wish to assign to the link which in this case is the **navbar-brand** class which is needed to position the brand correctly in the menu bar.

When the **_Layout** view is rendered during application startup the **ActionLinks** will be rendered as anchor tag (**<a>**) elements. The **ActionLink** extension method is a more convenient way of creating anchor tags and you can be certain that the relative path to the controller action will work when rendered. The provided action and controller names will be used to construct the relative path to the action method.

```
        <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("Application name", "Index", "Home",
                new { area = "" }, new { @class = "navbar-brand" })
        </div>
```

For the **navbar** to act as a menu a second area must be added where the actual menu items are placed. This area must be represented by an element with the **nvabar-collapse** and **collapse** classes defined, these two classes makes it possible to open and close the menu when the *hamburger menu* button is clicked.

An unordered list (**<ul>**) inside the menu area will contain the menu items (**<li>**). The unordered list must be defined with the **nav** and **navbar-nav** classes to act as a menu items container. You can use nested items if you want to create a drop-down menu for one of the top-level menu items, this will be explored in a later chapter. The default menu has three menu items all defined using the **ActionLink** extension method.

The **Partial** extension method renders a partial view containing the *Login, Register, Manage* and *Logout* links located to the right in the navigation bar.

To add a new menu item to the **navbar** a suitable action method must exist in a controller and you have to add an **ActionLink** referencing it inside the **navbar-nav **.

```
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                <li>@Html.ActionLink("About", "About", "Home")</li>
                <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                <li>@Html.ActionLink("Description", "MyAction",
                    "MyController")</li>
            </ul>
            @Html.Partial("_LoginPartial")
        </div>
    </div> @*End of container*@
</div> @*End of navbar*@
```

6. CSS Styling

Introduction

This chapter is all about HTML and CSS and how you can style the elements that make up your application. This is meant as an introduction, not a comprehensive list of all the CSS styles and HTML elements available for you to use. We will have a brief discussion about the elements you will use when completing the exercises in this book, it will be a great starting point for you when you go out and explore more on your own, once you have mastered the basics. You will use CSS throughout the modules in the book to style the elements as you create more complex user controls that sometimes are made up of several components.

You use CSS to control formatting, presentation and layout of a web page by using things like colors, layouts and fonts. Although you can do styling directly on the HTML elements in the HTML markup it is not recommended that you do so unless you really have to. The preferred approach is to create one or more **.css** files that are linked to individual views or the **_Layout.cshtml** file that renders the separate views in a MVC application.

There are several reasons why you want to use **.css** files instead of inline styling or styling in separate views; it makes it much easier to read the HTML markup, easier to maintain the CSS and HTML markup and it enables reuse. By separating out the CSS it can be applied to any element in any view without duplication of the CSS and thus achieving consistency across the web site.

Technologies used in this chapter

- **HTML 5** - HTML elements are used when illustrating how CSS works.
- **CSS** - Used to style HTML elements.
- **F12 Web Developer Tool** - a browser tool for investigating the Document Object Model (DOM).

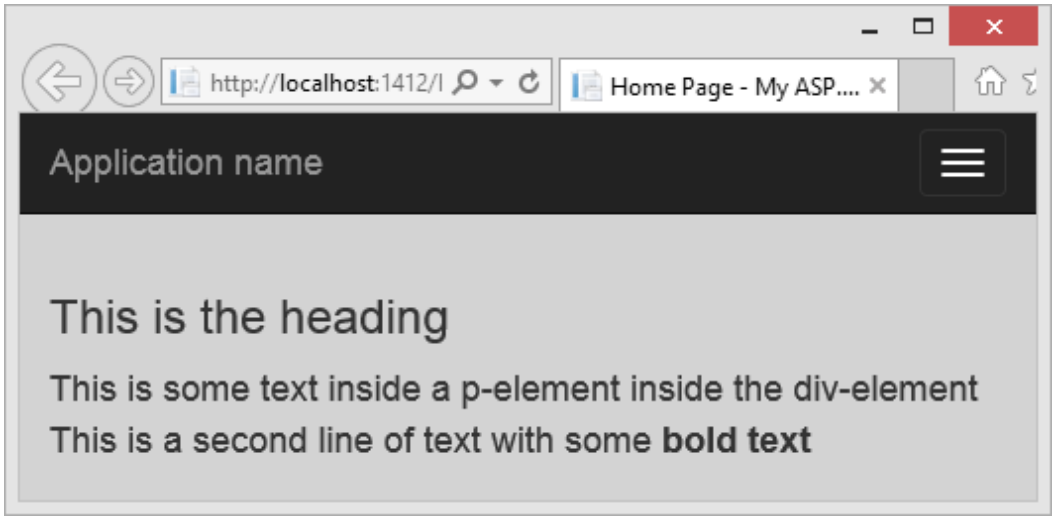
Below is a short example of inline styling. Imagine that the view contain hundreds of lines of HTML markup where inline styling is applied, it would get ugly very fast. Note

that the body element is located inside the MVC application's **_Layout.cshtml** view and is shared among all views.

```
<body style="background-color:lightgray;">
    <h3>This is the heading</h3>
    <div>
        <p style="font-size:large">This is some text inside a paragraph
           element inside the div-element
           <br/>
           This is a second line of text with some <strong>bold
           text</strong>
        </p>
    </div>
</body>
```

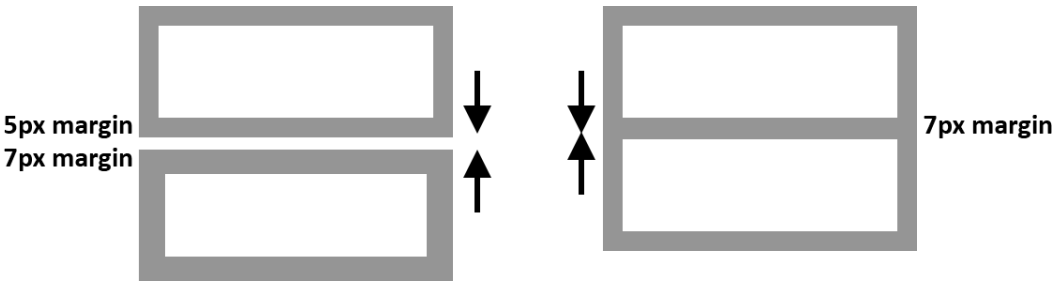
Instead of using inline styling a **<style>** block can be used to define the layout, this would at least keep the CSS separate from the HTML markup.

```
<style type="text/css">
    body { background-color:lightgray; }
</style>
```



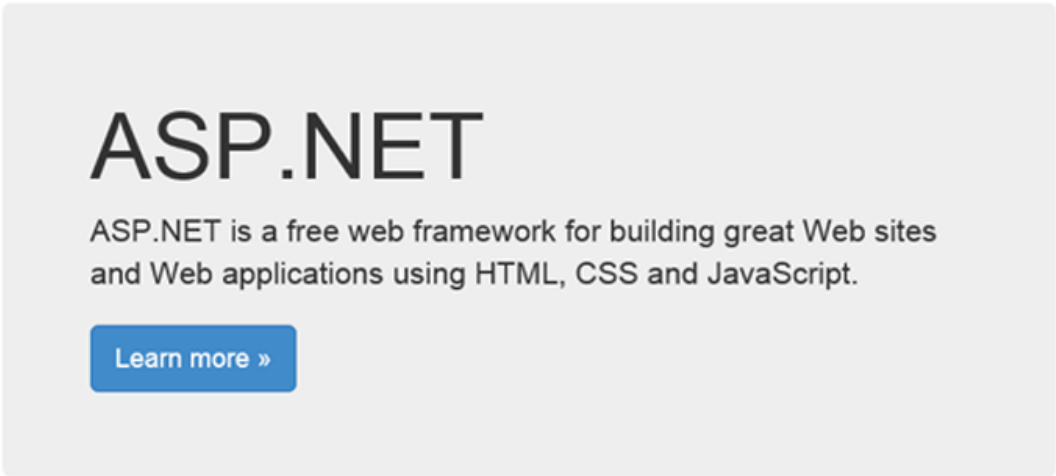
CSS Box Model

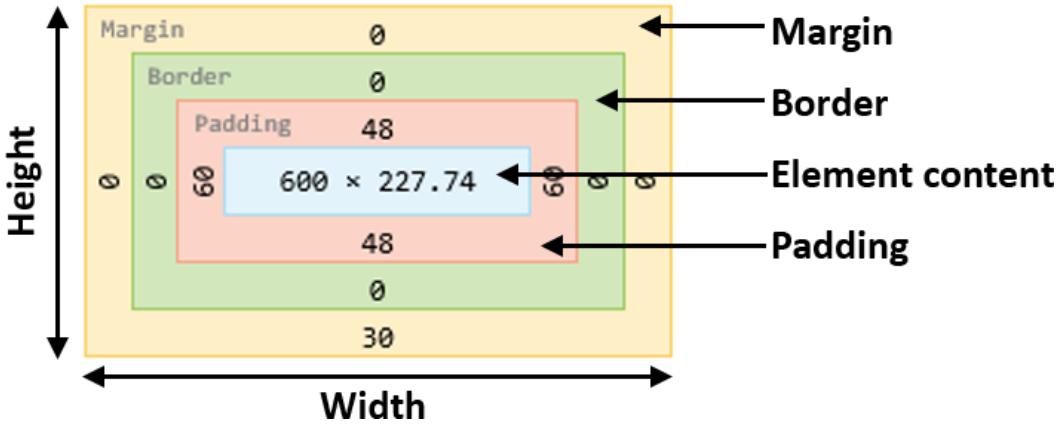
In the CSS box model every element is represented by a square which has a width and a height as well as padding (space used to position the content inside of the element), margin (space added to the outside of the element) and a border. When two vertical margins meet they overlap leaving only the maximum margin between the two elements, this does not apply to horizontal margins.



When specifying a width of an element you are specifying the width of the content area of that element, any border, margin or padding will add to that width. As an example, if you specify an element's width as 100px and the left and right margin as 5px then the total width of the element would be 110px.

In the example below the CSS box model is displayed for a ** ** with the Bootstrap **Jumbotron** class applied.





The Display and Visibility Properties

The **display** property determines how an element should be displayed on screen and is usually set to **block** (placed on a new row), **inline** (placed as an element on the same line. Elements move to a new line when there's not enough room available to fit them all on a single line) or **none** (not displayed and does not leave a space where the element should have appeared). You cannot assign widths to inline elements they will only expand to make room for the content, border and padding. If you want to set the width of an inline element then you have to set its display property to **inline-block**.

The **visibility** property shows or hides an element. A **hidden** element leaves an empty space where the element should have been displayed if visible.

Property	Description
display:block	Elements are displayed on new lines (default for many element types)
display:inline	Places elements on the same line. Element widths cannot be assigned.
display:none	The element is completely removed from the page.
display:inline-block	Elements are displayed on the same line and widths can be assigned.
visibility:hidden	Hides an element and leaves an empty space where the element should have appeared if visible.

Property values

There are many ways to specify values for CSS properties because there are a lot of things you have to specify values for, like borders, background colors, font sizes and many others. You can use keywords to specify some of them like **thin**, **thick** and **large** but these keywords cannot be used everywhere, **thin** as an example can be used to define the width of a border but not the size of a font and **large** can be used to define a font size but not a border width.

You can use physical measurements such as centimeter (cm), millimeter (mm), inches (in), picas (pc) where one pc equals 12pt and points (pt) which often are used to define font sizes when writing text in a word processor, where one point is 1/72 of an inch.

Another popular measurement is pixels (px) where 1px is equal to 1/96 of an inch.

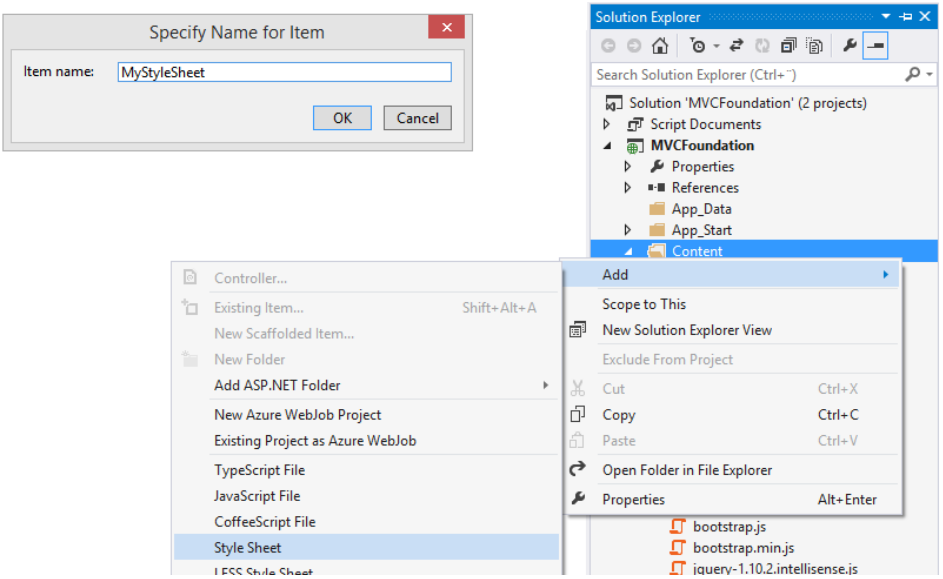
Some prefer to use relative measurements when designing a web site using percent (%) or **em** where 1 **em** is the current font size of the element and 2 **em** is twice that size. Relative measurements like these lends themselves well to designing for mobile devices.

There are also some CSS functions available such as **rgb** for assigning a color value with decimal numbers, one for each of the three colors red, green and blue, and **url** for specifying a background image.

You will see some of these in action throughout this chapter and in the rest of the book.

How to add a CSS style sheet

1. Right click on the **Content** folder in Visual Studio and select **Add-Style Sheet** in the context menu.
2. Give the style sheet a name in the dialog box and click **OK**.



3. Open the **_Layout.cshtm** view and add a reference to the **.css** file. You can do this by dragging it from the solution explorer to the **<head>** element of the view or you can copy an existing **Styles.Render** line and modify it.
   ```
   @Styles.Render("~/Content/MyStyleSheet.css")
   ```
4. When you have add the desired styles to the file you reference them from the HTML markup.

Adding styles to the style sheet

Now that you have created a style sheet and referenced it from the **_Layout** view all styles added to this style sheet will be available throughout the entire web site.

If you want to remove the ugly inline styling from the HTML markup all you need to do is to move it into the style sheet and reference the desired elements.

Move the styles into the *MyStyleSheet.css* file. When adding a style to a style sheet you place the name-value pairs defining the style properties inside curly braces after the element name.

```
body { background-color:lightgray; }
p { font-size:large; }
```

The CSS specification is maintained by the World Wide Web consortium (W3C), to keep up with what is happening in the CSS world you can read all about it at *www.w3.org/Style/CSS*. It is important to follow the news on this page because there is varying support for CSS among the browsers.

You can find out CSS browser compatibility at http://www.quirksmode.org/compatibility.html.

Style properties

A property consist of a *name-value* pair, you can find a listing of all style properties on the W3School web site. The name and the value of a property is separated by a colon (:) and it ends with a semicolon (;). You can specify more than one property for each selector. The following example would make the font size larger and the text red for all paragraph (**<p>**) elements.

```
p {
    font-size:large;
    color:#ff0000;
}
```

Selectors

So far you have seen the **body** and **p** selectors, in this section you will learn about some other powerful selectors.

Simple selectors

The previously mentioned **body** and **p** selectors are known as simple selectors that target element types meaning that the styles are applied to all element of that type. Other simple selectors are **input**, **button**, **div**, **ul** (unordered list), **li** (list item), **span** (groups inline-elements), **form**, **table**, **tr** (table row), **th** (table header), **td** (table data), **i** (part of a text that is different from the surrounding text), **h1-h6** (headers), **a** (link) and many more.

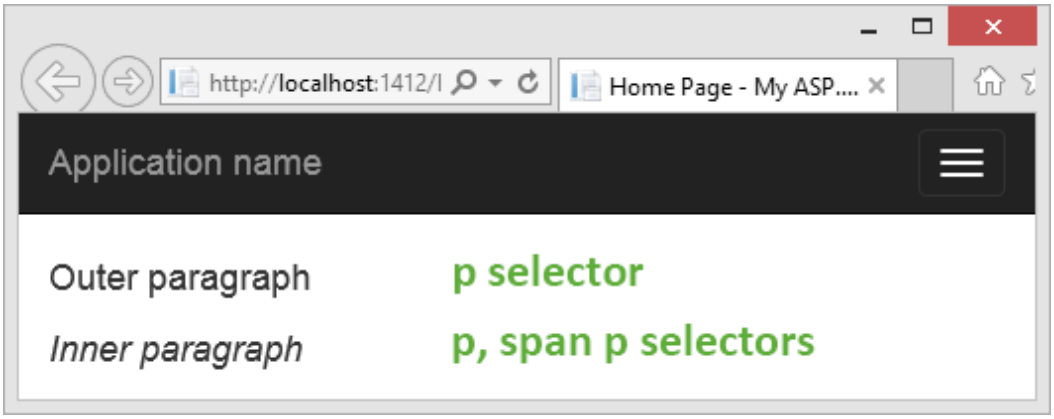
It is possible to stack multiple selectors on the same row targeting them all by separating them by commas (,). It is also possible to target elements within another element by separating them with a space, you can do this for many levels if you have elements nested within elements.

An example could be that you in the following HTML markup want to target the paragraph **<p>** which is defined inside the inner **** but not the one outside the ****.

```
<p>Outer paragraph</p>
<span>
    <p>Inner paragraph</p>
</span>
```

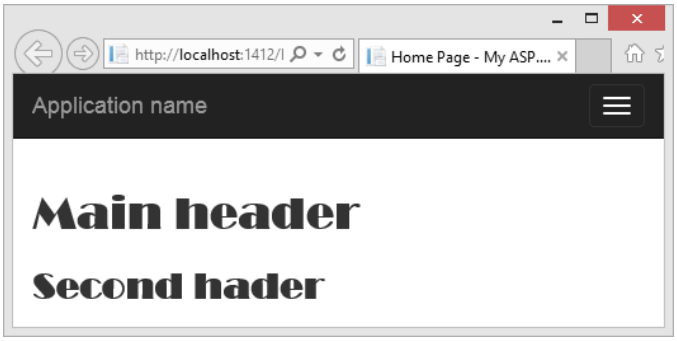
Let's apply the following rules and see what happens. If all goes according to plan the text in the inner paragraph should be displayed italicized and both paragraphs should be displayed with a larger font size.

```
p { font-size: large; }
span p { font-style:italic; }
```



The following example targets all headings, make their text bold and change their font family (note that the font used in the example is not recommended in a real world scenario because it is very hard to read).

```
h1, h2, h3, h4, h5, h6 {
    font-weight:bold;
    font-family: Broadway;
}
```



Here's a list of simple selectors you will work with in the examples in this book.

```
a { }
body, h1, h2, h3, h4, h5, h6 { }
button { }
i { }
ul { }
li { }
div { }
span { }
head { }
body { }
form { }
table tr, table th, table td { }
input { }
```

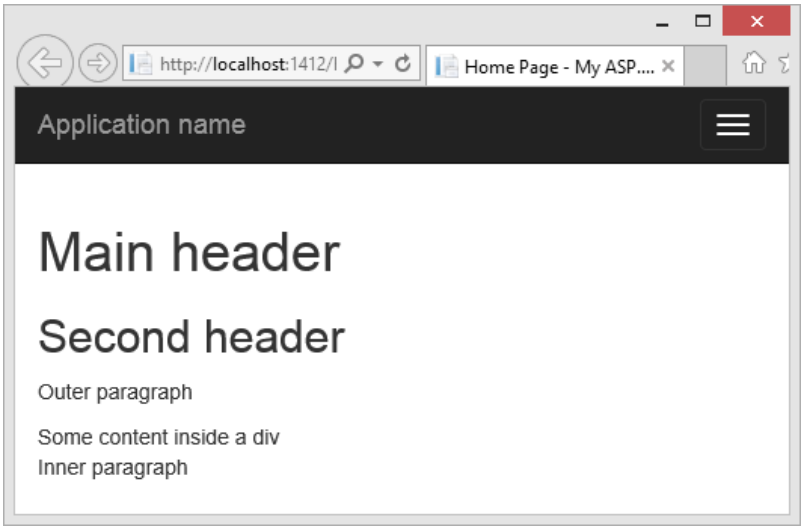
Other selectors

There are other selectors which are frequently used when styling a web page. Instead of targeting elements of a certain type you might want to target an element by its unique id using the **#**-sign, class names which can be used on multiple elements with a dot (.) prefix or square brackets ([]) which target elements with a certain attribute or attribute value. You can create your own attributes by prefixing the name with **data-**, these attributes will be ignored by the browser but can be used in CSS and JavaScript.

Selector	Description
#id-name	Selects a specific HTML element by its unique id.
.class-name	Selects all HTML elements decorated with a specific class name.
[attribute-name]	Selects all HTML elements decorated with a specific attribute.

The HTML markup for this example has no inline styles and the style sheet has been cleared so that only Bootstrap styles are applied.

```
<h1>Main header</h1>
<h2>Second header</h2>

<p>Outer paragraph</p>
<div>Some content inside a div</div>
<div>
    <p>Inner paragraph</p>
</div>
```

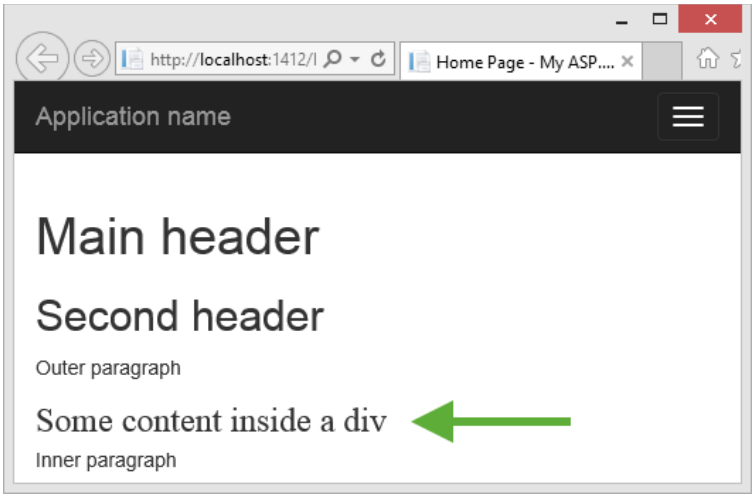


Let's change the styling using an id-selector on the first ** **. To do this you have to add a unique id to the element, the id *some-content* is used in this example.

```
<div id="some-content">Some content inside a div</div>
```

Use the **#**-symbol before the name in the style sheet to target this specific ** **. Let's make the text stand out more by changing its font to **Time New Roman** which is a *serif* font. *Serif* fonts have small embellishments added to them which their counterpart *sans-serif* don't have. In this example more than one font has been specified in case **Times New Roman** is not available in the operating system, the last font is the generic **serif** family which can be any serif font.


```
#some-content {
    font-family:'Times New Roman', Times, serif;
    font-size:x-large;
}
```



Let's add the class name *paragraph* to the first **<p>** element and the second ** ** and the attribute *data-heading* to the two headings. Then let's add selectors to them in the CSS file. With the added class, attribute and the previously added id the HTML markup will look like this.

```
<h1 data-heading>Main header</h1>
<h2 data-heading>Second header</h2>

<p class="paragraph">Outer paragraph</p>
<div id="some-content">Some content inside a div</div>
<div class="paragraph">
    <p>Inner paragraph</p>
</div>
```

When adding a color property to a selector in a style sheet a color bar with the most recently used colors in the project will automatically be displayed for you to choose

from, if you can't find the color you are looking for there is a more advanced color picker you can open by clicking on the small button with the two arrows at the end of the color bar.

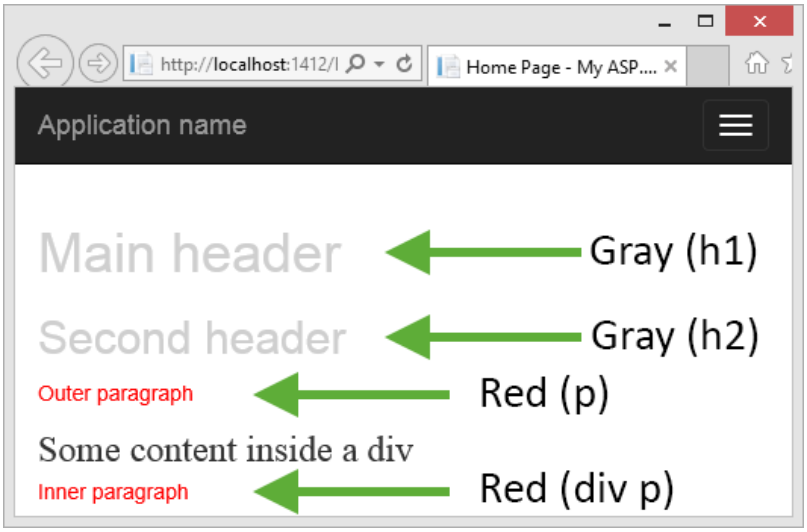
A color can be assigned to a property in 3 ways; with the **#**-sign followed by the hexadecimal RGB color values, specifying the color by name or by calling one of the **rgb** or **rgba** CSS methods passing in decimal color values for red, green, blue and alpha (opacity).

```
color:#cfcfcf;
color:rgb(207,207,207);
color:lightgray;
```

A light gray color (**#cfcfcf**) will give the headings a more dimmed look and a red color added to the elements using the *paragraph* selector will make them stand out from the rest of the text.

```
.paragraph { color:red; }
[data-heading] { color:#cfcfcf; }
```

You might be surprised that the text in the innermost paragraph residing in the ** ** element turns red despite not having the *paragraph* class defined. The reason is that the **<p>** element inherits the color from its parent which is the ** ** with the *paragraph* class.



More complex selectors

You can run into scenarios where you need to target elements within elements of simple types or a certain sub-type like the **input** selector which can represent a button, input field, a checkbox or a radio button. The more commonly used complex selectors for these scenarios are described in the upcoming sections.

Elements separated by space

In a selector where the elements are separated by a space the right element has to reside inside the left element. The same rule applies when classes (.) or a unique id (#) is involved.

- div p
  Targets all **<p>** elements located inside ** ** elements.
- .class1 .class2
  Targets all elements decorated with a class named **class2** located inside an element decorated with a class named **class1**.
- #id-name .class1
  Targets all elements decorated with a class named **class1** located inside an element with the unique id **id-name**.

In the following example all text inside **<p>** tags which reside inside a ** ** will be blue.

```
div p { color:blue; }
```

```
<div>
    <p>Outer paragraph</p>
    <div>
        <ul>
            <li><p>Inner paragraph</p></li>
        </ul>
    </div>
</div>
```

Elements separated by a greater than sign (>)

For elements separated by a greater than sign (>) the right element has to be a direct descendant of the left element, it cannot be a child element to an element inside the left element. The same rule applies when a class (.) or a specific id (#) is involved.

- div > p
  Targets **<p>** elements located directly inside a ** ** (parent). The **<p>** element (child) cannot be nested inside another element in the ** **.
- .class1 > .class2
  Targets elements decorated with the class **class2** located directly inside an element decorated with the class **class1**.
- #id-name .class1
  Targets elements decorated with the class **class1** located directly inside an element with the unique id **id-name**.

In the following example all the **<p>** (child) elements has to reside directly inside a ** ** (parent). The **<p>** element (child) cannot be nested inside other elements in the ** **. The first **<p>** element ("Outer paragraph") will turn blue but not the nested one because it is not a direct descendant of a ** ** element.

```
div > p { color:blue; }
```

```
<div>
    <p>Outer paragraph</p>
    <div>
        <ul>
            <li><p>Inner paragraph</p></li>
```

```
        </ul>
    </div>
</div>
```

Selectors directly following each other

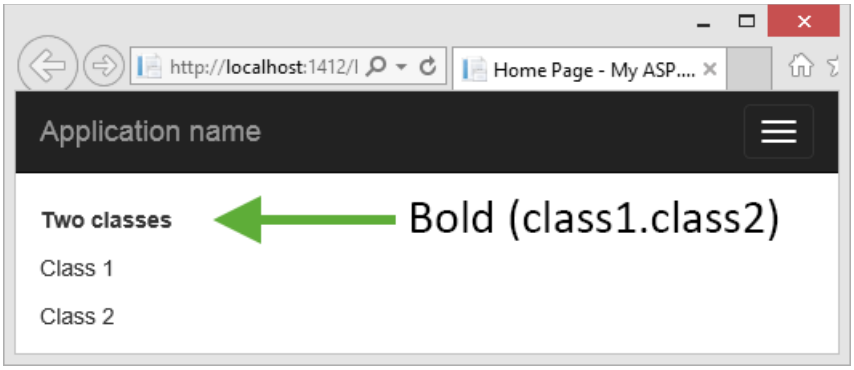
Selectors without any space or other character in between, for instance two class names back to back, requires the target element to have all involved selectors present for the style to be applied.

- .class1.class2
  Targets elements which are decorated with both **class1** and **class2**.
- [attribute-name].class1
  Targets elements which are decorated with the attribute **attribute-name** and the class **class1**.

In the following example elements decorated with the two classes **class1** and **class2** will have bold text.

```
.class1.class2 { font-weight:bold; }
```

```
<p class="class1 class2">Two classes</p>
<p class="class1">Class 1</p>
<p class="class2">Class 2</p>
```



Elements of a certain type

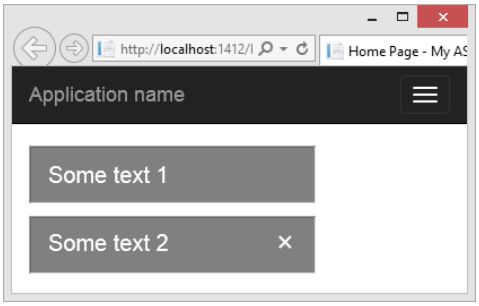
Elements can be selected based on element type. An **input** element can for instance be a button, a textbox, a submit button, a checkbox or a radio button. To target a specific

type of **input** element you need to specify the desired type inside square brackets because it's an attribute on the element; **input[type="text"]** for textboxes, **input[type="checkbox"]** for checkboxes, **input[type="submit"]** for a submit button, **input[type="button"]** for buttons and **input[type="radio"]** for radio buttons.

The following example displays all textboxes on separate rows (**display:block;**) and gives them a bottom margin of 10px. Space is added inside the textboxes (padding), the font size is 20px, the background color gray and the text color white.

```
input[type="text"] {
    display:block;
    margin-bottom:10px;
    padding:10px 15px 15px 15px;
    font-size:20px;
    background-color:gray;
    color:white;
}
```

```
<input type="text">
<input type="text">
```



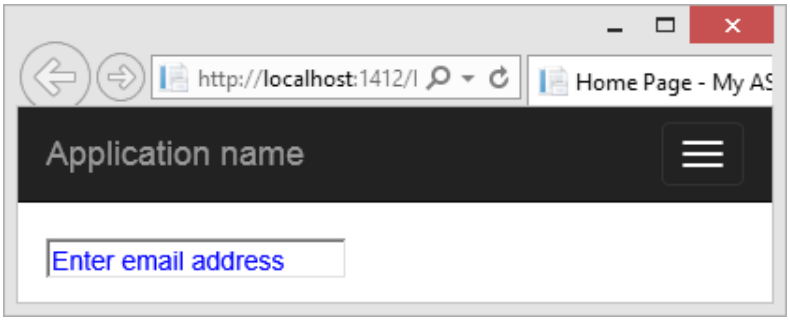
Target elements by attribute

Elements can be selected based on the attributes they are decorated with like the placeholder text inside a textbox.

The following example would change the placeholder text (the text describing what should be entered into the textbox) from black to blue.

```
input[placeholder] { color:blue; }
```

```
<input type="text" placeholder="Enter email address"/>
```



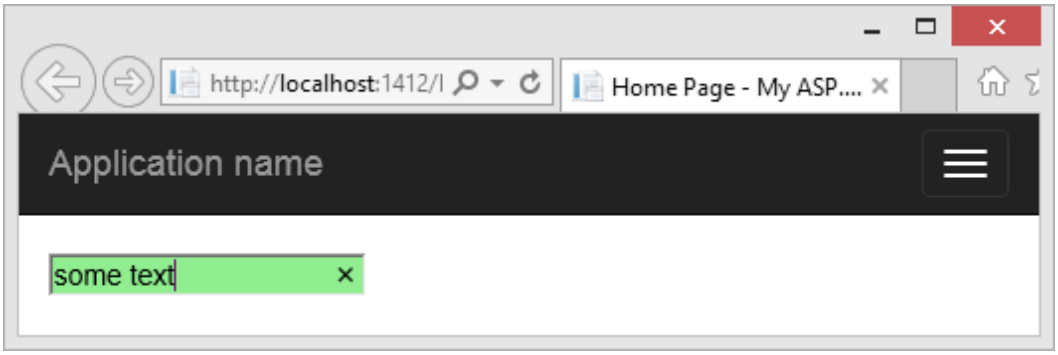
Pseudo classes

Pseudo class selectors are not explicitly assigned to an element but are available as a part of it, among them are **hover**, **visited** and **focus**. The **hover** class is activated when a user hovers with the mouse pointer over an element, the **visited** class is assigned when a user has clicked on a URL link created with an **<a>** element and the **focus** class is activated when a control is the currently used control (is in focus).

Add a colon (:) followed by the name of the desired pseudo class at the end of a CSS selector without a space to target that specific pseudo class. The **focus** pseudo class can for instance be used to change the background color of a textbox when a user clicks or tabs into it to enter a value.

```
input[type=text]:focus {
    background-color:lightgreen;
}
```

```
<input type="text"/>
```



Cascading

CSS styles are applied in a cascading manner to the HTML elements meaning that the most specific rule wins when conflicting rules are resolved.

The following example has three conflicting rules. The first states that the background color of paragraphs (**<p>**) residing in a ** ** will be green, the second rule states that paragraphs (**<p>**) should have blue backgrounds and the third rule states that all elements should have a red background. The question is which color will be applied to the two paragraphs (**<p>**)?

```
div > p { background-color: green; }
p { background-color: blue; }
* { background-color: red; }

<div>
   <p>This is a paragraph</p>
   <p>This is a second paragraph</p>
</div>
```

Because the **div > p** selector is more specific than the other two selectors the background color will be green in the two paragraphs.

A style can be applied from a variety of sources. You as a developer can provide linked in style sheets, script sections directly in the HTML markup or through the **style** attribute directly on elements. But styles can also be provided from a user-defined style sheet assigned through the browser settings and from the default styles provided by the browser itself. So which style will be applied? The developer styles always rank higher than user- and browser settings.

One way a property in a user style sheet can have more importance than the same property assigned from a style sheet linked to the web application is by using the **!important** keyword in the user style sheet. You as a developer can use this keyword to force an override of a property from another style sheet.

If you look at the previous example and make one small alteration to the CSS you can force the paragraphs to have a blue background by applying the **!important** keyword.

```
p { background-color: blue !important; }
```

But what happens if the same rule is assigned more than once by the developer in one or more style sheets?

```
p { background-color: blue; }
p { background-color: green; }
```

In this scenario the last selector would win and the background would be green because they have the same weight and the last change to the property will always be applied.

If two different style sheets have the same selector defined the order in which the **.css** files have been linked to the web application in the **_Layout.cshtml** file will be significant because then the selector in the style sheet that is linked in last will win.

Specificity

Each selector is given a rating by the browser to determine which selector's property will be applied, the higher specificity rating the more important the selector is. You can think of the ranking system as having three parts A, B and C where A has a higher ranking than B and B has a higher ranking than C.

A = The number of id selectors present in the rule.
B = The number of class and attribute selectors present in the rule.
C = The number of type selectors present in the rule.

```
*                /* A=0, B=0, C=0 ---   0 */
p                /* A=0, B=0, C=1 ---   1 */
div p            /* A=0, B=0, C=2 ---   2 */
p.paragraph      /* A=0, B=1, C=1 ---  11 */
#content         /* A=1, B=0, C=0 --- 100 */
```

In the image above the **#content** selector will have the highest rating with a rating of 100 and the **p.paragraph** element and class selector will have a rating of 11, all the way down to the * selector which has the lowest rating of 0. There is however a way that a property can outtrumph all other changes to that property and that is if it's assigned through an inline style using the **style** attribute directly on an element.

```
<p style="background-color:purple;">This is a paragraph</p>
```

A winning selector only replaces the conflicting CSS properties not all the defined properties. If lesser ranked selectors contain CSS properties that aren't part of the winning rule then those properties will still be in effect after the selectors have been evaluated.

Important: Only conflicting properties will be replaced.

If you go back to the example from the previous section you can now see why the paragraphs had a green background after the style had been applied. The **div > p** selector only applies to paragraphs who are immediate children of ** ** elements, for other paragraphs the second selector would win and the background would be blue.

```
div > p { background-color: green; }     /*A=0, B=0, C=2 ---   2*/
p { background-color: blue; }            /*A=0, B=0, C=1 ---   1*/
* { background-color: red; }             /*A=0, B=0, C=0 ---   0*/
```

Inheritance

Certain CSS properties are inherited from the parent to its children. If you for instance set the **color** property using the **body** selector all elements inside the **<body>** element will be affected and get the same text color. This is only true for some CSS properties like the text color, which you probably only want to assign once and maybe override for specific elements.

In this example all paragraphs except the one with the **content** id will have yellow text and the paragraph with the **content** id will have green text.

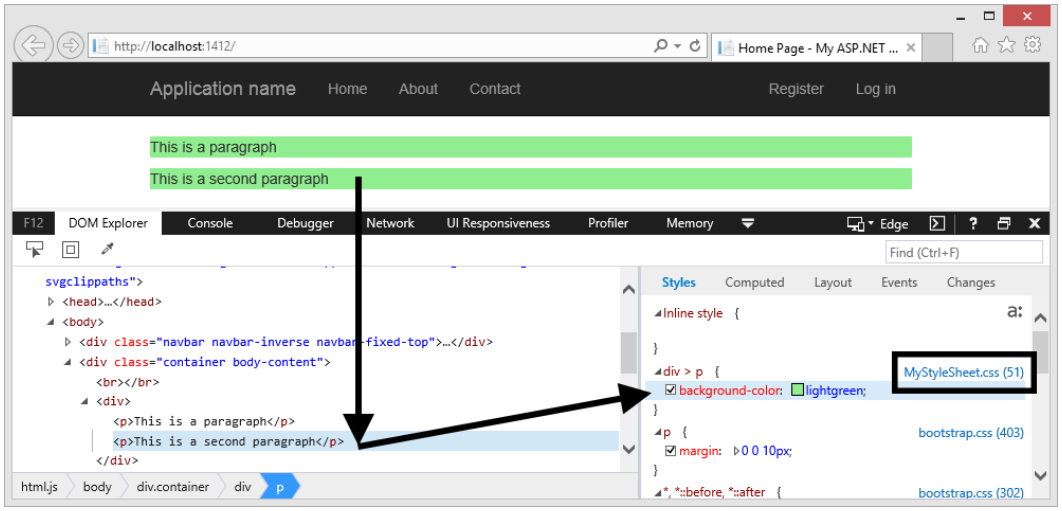
```
body { color:yellow; }
#content { color:green; }

<body>
   <p id="content">This is a paragraph</p>
   <p>This is a second paragraph</p>
   <p>This is a third paragraph</p>
</body>
```

Developer Tools (F12)

One of the most important tools when designing a web page is the Developer Tools which you can open by pressing the **F12** key in the browser. This tool allows you to inspect the HTML elements and their associated styles, it can be invaluable in situations where you are unsure why a style hasn't been applied the way you thought it would be.

In this section Internet Explorer is used to illustrate the **F12** tool, other browsers have similar functionality which you easily can learn by yourself once you have seen how it works in IE.

This example inspects the second paragraph in the Document Object Model (DOM) tree view which shows that the **background-color** property has been applied through *MyStylesheet.css*.



Inspecting an element by clicking it in the web page

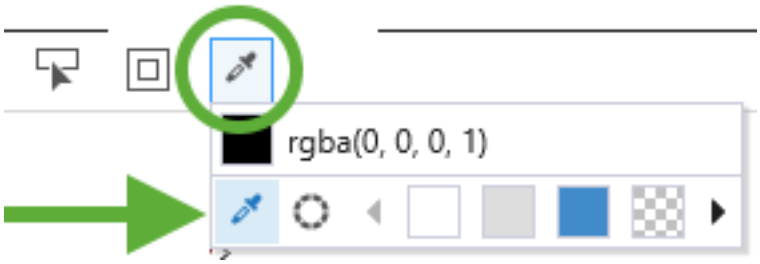
To inspect an element on the web page using Internet Explorer you click the left most button in the toolbar and then the element on the web page that you want to inspect.

Inspecting an element by selecting it in the tree view

To inspect an element by hovering over or clicking an element in the tree view you click the middle button in the toolbar and then click the element you want to inspect in the tree view.

Inspecting the color of an element

To inspect an element's color you click the third button in the toolbar and then the eyedropper button. Finally you click on an element on the web page and copy the color as an **rgba** function from the textbox and paste it into a style sheet.



Change style rules directly in the browser

When designing a page you might want to see what the result would be if you changed one or more CSS rules. You can temporarily change CSS rules directly in the browser by using the **F12** tool and simply add, delete or change rules in the CSS section of the tool window. You can make changes by clicking or right clicking on a CSS property or inside the curly braces of a rule.

Change HTML directly in the browser

When you are designing a page you might want to see what the result would be if you changed one or more HTML elements. You can temporarily change elements directly in the browser by using the **F12** tool and simply add, delete or change elements in the HTML tree in the tool window. You can make changes by clicking or right clicking on an element.

7. JQuery/Ajax

Introduction

This is meant to be a short and to the point introduction to JQuery and Ajax. The content prerequisites that you have basic knowledge of JavaScript, HTML and CSS, but you don't have to be an expert.

The first questions I asked myself when starting out with jQuery was *What is jQuery?* The answer is that it is a single powerful cross-browser JavaScript file, which means that you don't have to worry about which browser the web application is running in.

One of its strengths is the flexible way you can use *selectors* to reach into the Document Object Model (DOM) tree and manipulate the HTML elements that make up the web page. It also handles events in a cross-browser fashion, gone are the days when you had to handle events differently depending on the browser.

And last but not least you have rich Ajax (Asynchronous JavaScript and XML) support, nowadays XML often take the back seat to JSON (JavaScript Object Notation) which is more frequently used when handling objects in memory.

A nice thing about JQuery is the abundance of free plug-ins that you can use to save time by not having to re-invent the wheel.

JQuery shortens the time it takes to achieve many goals which would take a lot of effort with plain JavaScript, such as selecting all elements that have a certain class. Instead of having to loop through the DOM to find the elements you can achieve it with a single line of code and JQuery will do all the heavy lifting in the background for you. Another area is styles, with JQuery you can apply multiple styles to one or many elements with a single or a few lines of code, no looping needed.

JQuery is included by default when creating a MVC project and is referenced from the **_Layout.cshtml** view through one of the bundles in the *BuindleConfig.cs* file.

If you need to look up a certain JQuery feature I suggest that you start at the *jquery.com* web page, it has the complete JQuery documentation with short concise easy to understand examples.

Technologies used in this chapter

- **JavaScript/JQuery** - Will be used for client-side coding.
- **Ajax** - Will be used for asynchronous calls to the server. This technique can be used when updating part of a page without reloading the whole page.
- **HTML 5** - HTML elements will be targeted using JQuery, for instance attaching **click** events to buttons.
- **CSS** - To style elements.

$(document).ready()

In JQuery code you will frequently come across the **$**-sign which is an abbreviation for JQuery. But when is it used? You can use it to add JQuery functionality to DOM elements to get a richer set of functions to work with, this saves a lot of time when developing a web application because you can utilize a prebuilt function library.

The following code lines show different ways of wiring up the DOM ready method which is executed when the DOM tree has been loaded but before all the images and CSS has been loaded. This enables you to manipulate the DOM elements, like hiding certain elements or adding new elements into the DOM tree.

Note that you have to pass in a function as a parameter to the **ready** function, this is often an anonymous inline function like in the examples below. This function will be executed when the DOM tree has been loaded.

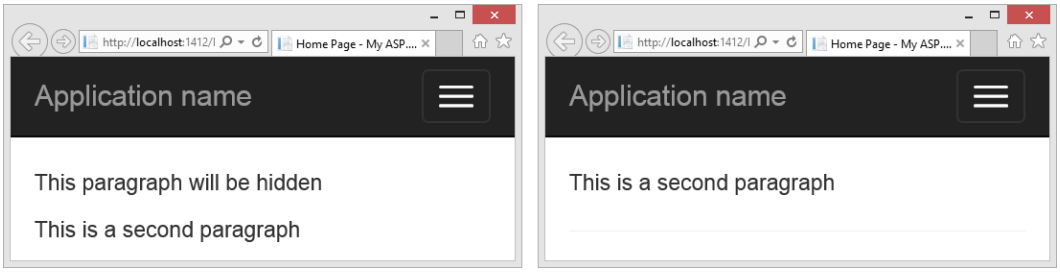
```
jQuery(document).ready(function(){ … });

$(document).ready(function(){ … });

$(function(){ … });
```

To use JavaScript and JQuery you can define a script block in the HTML document and use the two together, but a cleaner way is to create a **.js** file which holds the JavaScript and JQuery code; the **.js** file is then referenced or linked to the HTML documents where

needed, typically in the **_Laout.cshtml** view or a specific view. Using **.js** files is nice because it separates concerns, adds reusability and gives cleaner code.

In the following example one of the paragraphs will be hidden when the **ready** function is called, note that the script code is in the HTML document accompanied by a referenced JQuery file.

```
<div>
    <p id="hideMe">This is paragraph will be hidden</p>
    <p>This is a second paragraph</p>
</div>

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script type="text/javascript">
    $(function () {
        $('#hideMe').addClass('hide');
    })
</script>
```



Selectors

A selector is a way to access an element (node) like a **div**, **p** (paragraph) or a **span** in the DOM to manipulate it. In this section you will learn how to find and manipulate element content by using element-, id-, class name-, attribute-, child- and descendant selectors.

You can use the **$**-symbol when accessing elements through a selector by writing the selector expression in quotation marks.

```
$('selector expression')
```

By wrapping the selector(s) in **$()** you get access to JQuery functionality that can save a significant amount of time because you don't have to invent the wheel again and again using traditional JavaScript code.

Selector	Description
p, div, span, …	Select elements by the element type.
#myId	Select elements by the element's id.
.className	Select elements based on an assigned class.
[attribute-name]	Select elements based on an assigned attribute.
span:first-child	Select the first child element in **** elements.
Table tr,	Select all rows in a table.
div > p	Select all paragraphs that are immediate descendants (children) of a ** ** element.

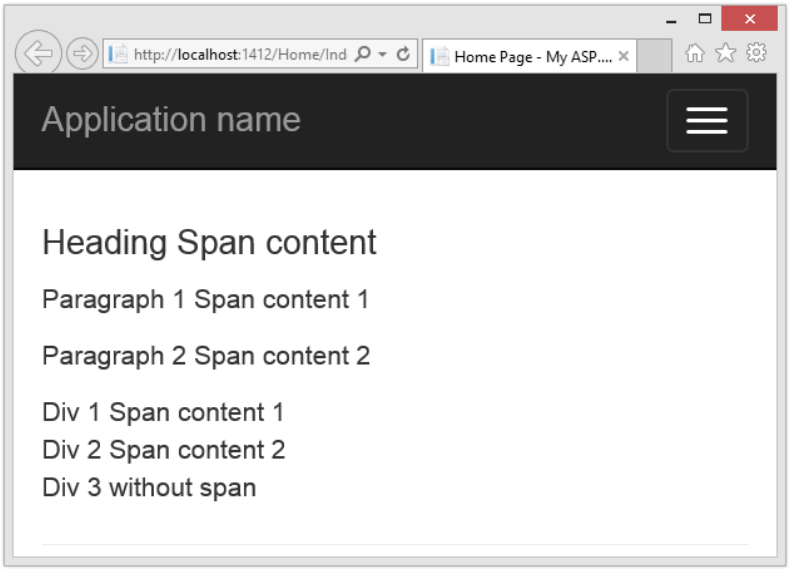
By Element Type

You can select an element by its type or multiple element types by separating them with commas in the JQuery wrapper.

Selector	Description
$('p')	Select all paragraphs.
$('div')	Select all divs.
$('a')	Select all anchor tags (links).
$('span')	Select all spans.
$('p, div, span')	Select all paragraphs, divs and spans.
$('table tr')	Select all rows in a table.
$('div > a')	Select all anchor tags who are immediate children of a ** **.

The upcoming examples use the following HTML markup to describe how selectors can be used in conjunction with commonly used JQuery functions.

```
<div>
    <h4>Heading <span>Span content</span></h4>
    <p>Paragraph 1 <span>Span content 1</span></p>
    <p>Paragraph 2 <span>Span content 2</span></p>
    <div>Div 1 <span>Span content 1</span></div>
    <div>Div 2 <span>Span content 2</span></div>
    <div>Div 3 without span</div>
</div>

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script type="text/javascript">
    $(function () {

    })
</script>
```

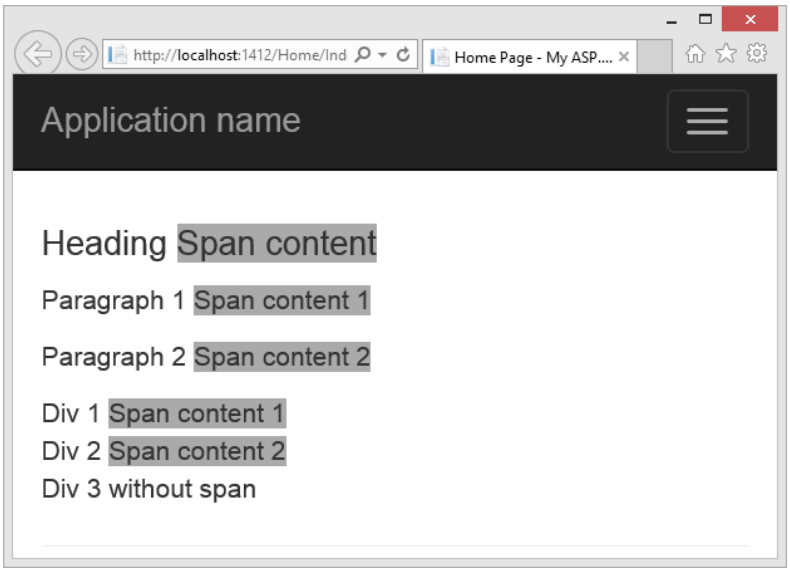


The .css function

You can use the **.css** JQuery function to manipulate elements styles like their color, background color, font size, and so on.

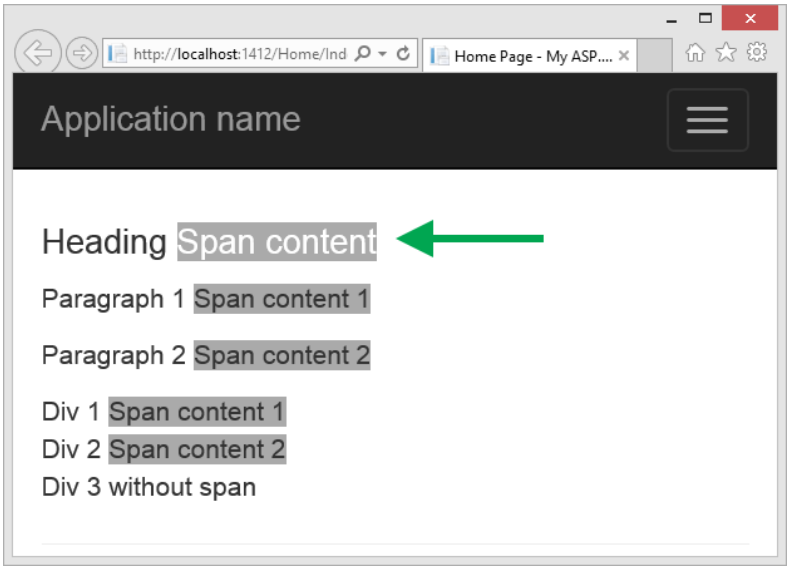
In this example all **** elements inside ** ** elements will get a dark gray background color.

```
<script type="text/javascript">
    $(function () {
        $('div span').css('background-color', '#aaa');
    })
</script>
```



The following example changes the text color to white for all **** elements that reside inside an **<h4>** element which reside inside a ** ** element.

```
<script type="text/javascript">
    $(function () {
        $('div span').css('background-color', '#aaa');
        $('div h4 span').css('color', '#fff');
    })
</script>
```

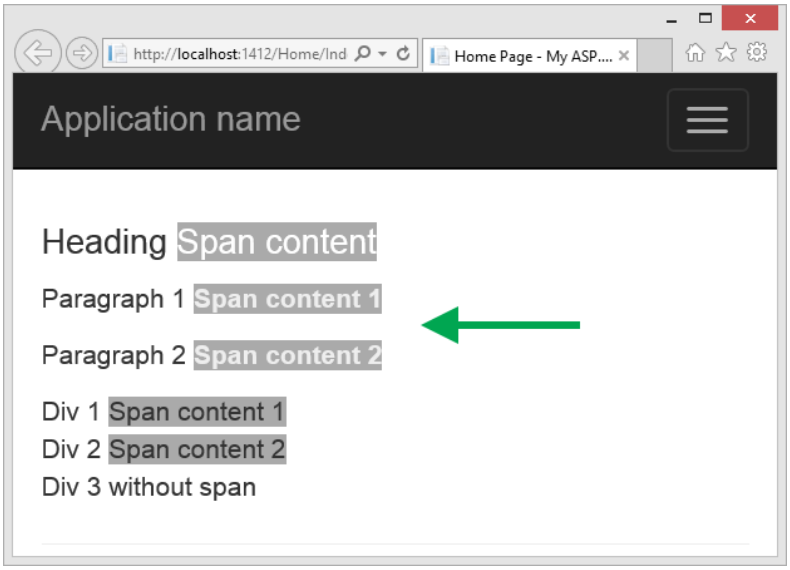


The following example changes the font weight to bold and the text color to a light gray (#eee) for all **** elements that reside inside a paragraph **<p>** element which reside in a ** ** element. Note that you can chain methods when manipulating elements or use an anonymous JSON object to assign values to multiple properties.

```
<script type="text/javascript">
    $(function () {
        $('div span').css('background-color', '#aaa');
        $('div h4 span').css('color', '#fff');
        $('div p span').css('font-weight', 'bold')
                       .css('color', '#eee');
    })
</script>
```

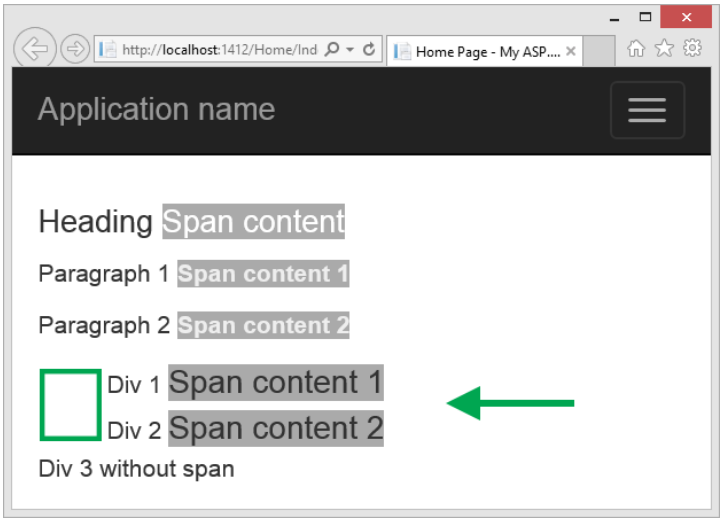
Note that the last statement can be written using an anonymous JSON object.

```
$('div p span').css({
    'font-weight': 'bold',
    'color': '#eee'
});
```



The following example changes the font size to 14pt for all **** elements that are direct children of a ** ** element which reside inside another ** ** element. It also adds a left margin of 40px to the ** ** elements which contain the **** elements, not to the **** elements themselves. The JQuery **parent** function is called to achieve the latter result.

```
<script type="text/javascript">
    $(function () {
        $('div span').css('background-color', '#aaa');
        $('div h4 span').css('color', '#fff');
        $('div p span').css('font-weight', 'bold')
                       .css('color', '#eee');
        $('div > div > span').css('font-size', '14pt')
                             .parent()
                             .css('padding-left', '40px');
    })
</script>
```

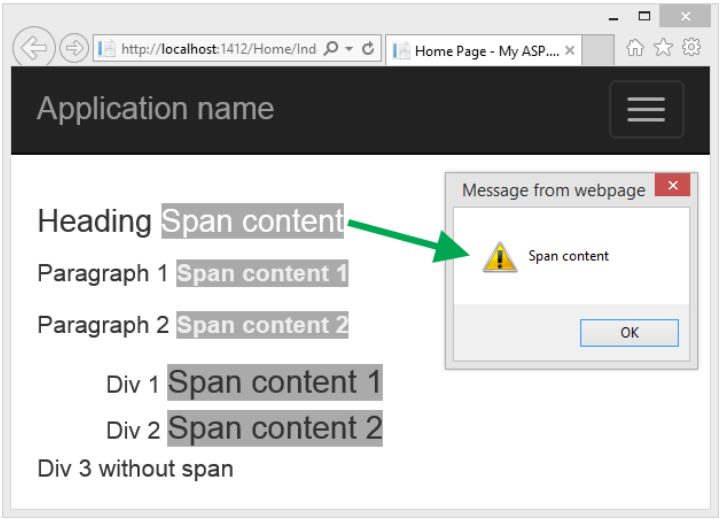


The .html function

You can use the **.html** JQuery function to manipulate the HTML content of elements effectively reading or changing the element content.

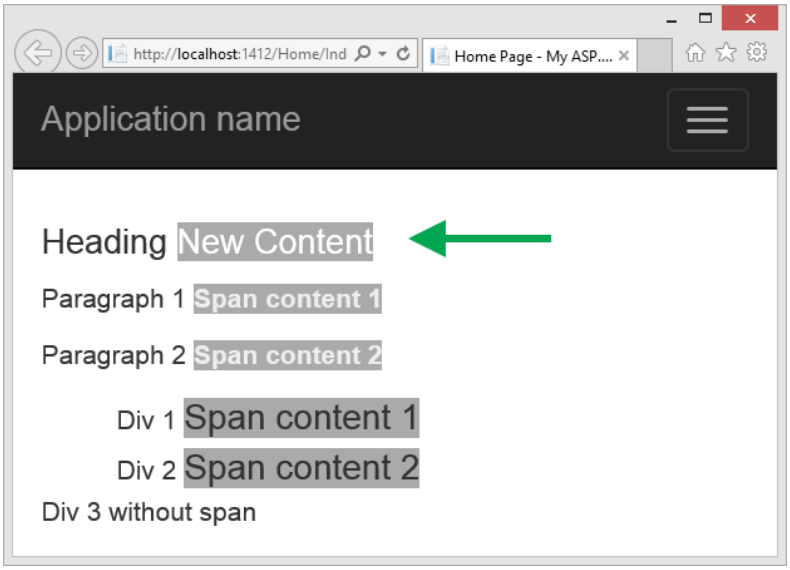
Let's start by reading the content of the **** in the **<h4>** element and displaying the text in a pop-up.

```
alert($('div h4 span').html());
```



Next the content of that same **** element is changed to "New content".

```
$('div h4 span').html('New Content');
```



Looping with the .each function

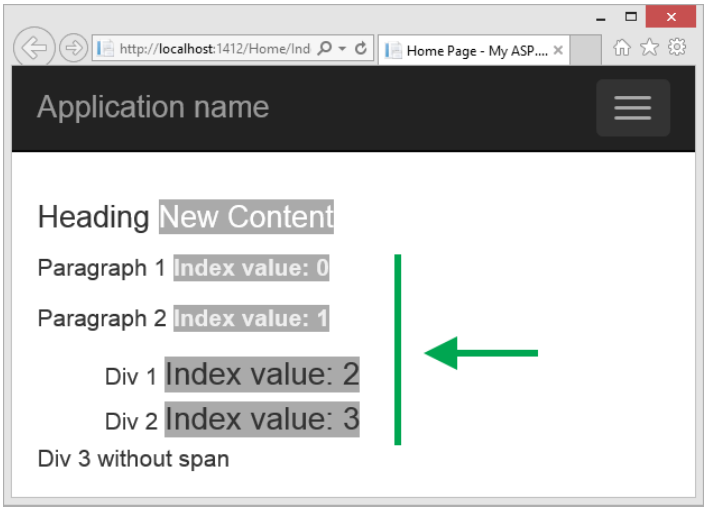
You can use the **.each** JQuery function to loop over elements matching the selector expression.

In this example you want to change the content of the **** elements that either reside in a paragraph **<p>** or in a paragraph that reside in a ** ** (**div p span**), or where the **** is the immediate child of a ** ** which is an immediate child of another ** ** (**div > div > span**).

Important: *The **this** object inside the callback function of the **.each** loop is the raw DOM object of the current element being processed by the loop. By wrapping it in a JQuery wrapper you can use JQuery functions on that element.*

You don't have to pass in the index parameter, I did it here to show how easy it is to get the current loop index. Here the index is appended to the text displayed in the affected elements.

```
$('div p span, div > div > span').each(function (index) {
    $(this).html('Index value: ' + index);
});
```



By Id (#)

The id selector is the fastest selector because the DOM is optimized for it when traversing the DOM tree. It is also the most specific of the selectors since it should be unique. You can use it to narrow down the focus to a specific area of the DOM tree and continue to narrow it down even further by using multiple selectors with the id.

In this section you will work with an unordered list which is a very flexible type of list that often is used to do navigational components. As you can see in the HTML below there are two nested lists.

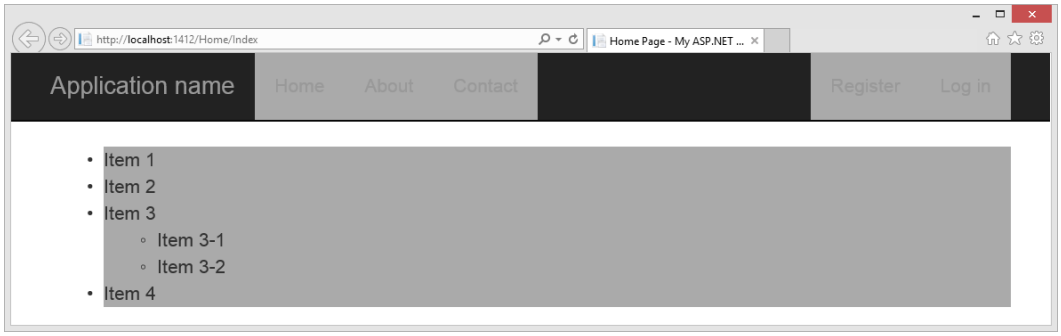
```
<div id="nav-div">
    <ul id="outer-list">
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3
            <ul id="inner-list" >
                <li>Item 3-1</li>
                <li>Item 3-2</li>
            </ul>
        </li>
        <li>Item 4</li>
```

```
    </ul>
</div>
```

The first thing you want to accomplish is to assign a background color to all the list items **<li>**. To do this you could try to add a selector that finds all the list items **<li>** located in unordered lists **<ul>** which resides in a ** ** element using a selector like:

```
$('div ul li').css('background-color', '#aaa');
```

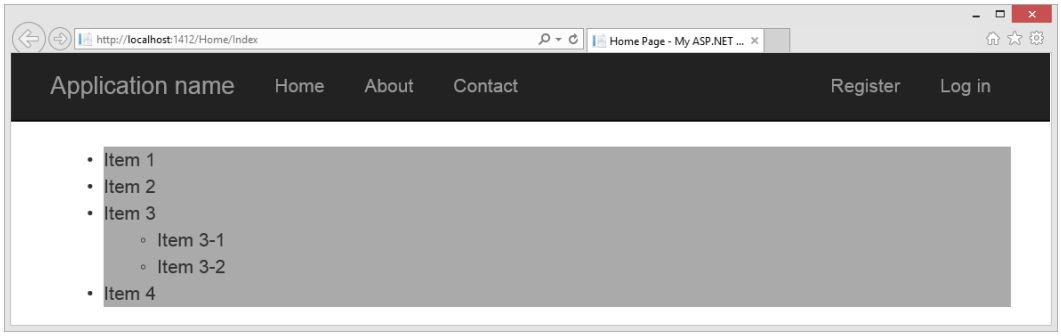
This would work fine if it wasn't for the page navigation menu which also contains unordered lists inside ** ** elements.



So how do you solve this dilemma? Well, you could use an id to pinpoint the beginning of your lists like the ** ** container and work your way from there. To target an element id you prepend the element id with a hash tag (#) also known as a pound symbol.

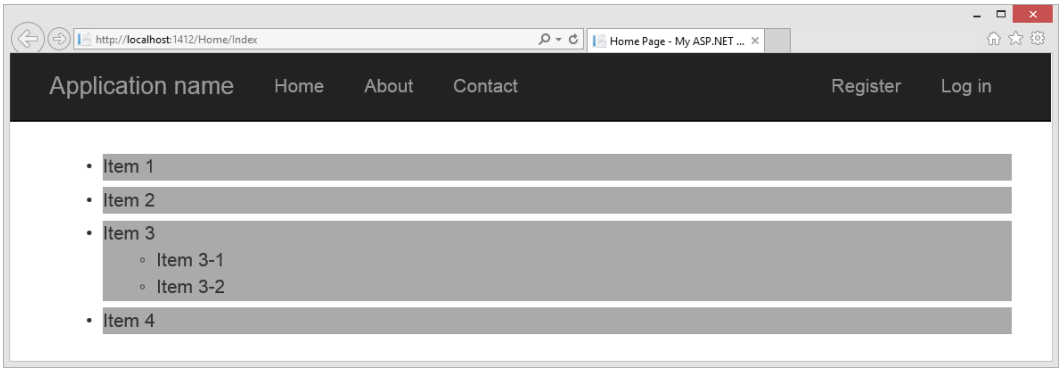
```
$('#nav-div ul li').css('background-color', '#aaa');
```

Note that the menu no longer has a gray background.



Next you want to target only list items who are immediate children to the outer list (first level) and assign an upper margin of 5px to them to add some space between the items. To achieve this you can use the outer **<ul>** element's **outer-list** id to target its children in the DOM. Use the greater than sign (**>**) between two selectors to target immediate child elements.

```
$('#outer-list > li').css('margin-top', '5px');
```



Next you want to remove the decoration (the bullet) to the left of the items in the inner list. To do this you can target the list using a selector starting at the outer list and find its **<ul>** children or you can use the inner **<ul>** list's **inner-list** id.

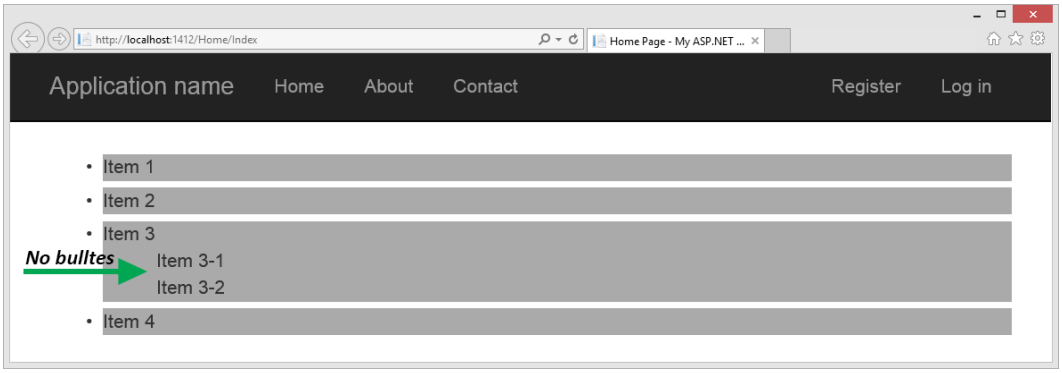
Below are a few selector examples targeting unordered lists inside the outer list and removing the bullet from the items. To remove the bullets you set the **list-style-type** of the **<ul>** element surrounding the list items (**<li>**) to **none**.

```
// Targets the inner list directly by its id
$('#inner-list').css('list-style-type', 'none');

// Finds immediate <li> children of the outer list
// that has an unordered list as its immediate child
$('#outer-list > li > ul').css('list-style-type', 'none');

// Finds all <li> elements in the outer list that
// has an unordered list as its immediate child

$('#outer-list li > ul').css('list-style-type', 'none');

// Finds all unordered lists in the outer list in all levels
```

```
$('#outer-list ul').css('list-style-type', 'none');
```

And here is another example accomplishing the same thing using the **children** JQuery method to target the children of an element.

```
$('#inner-list').children().css('list-style-type', 'none');
```



By Class Name

Class names can be an effective way to target one or more elements in the DOM. They are also perfect for toggling CSS settings at run-time since they can be added and removed as needed using JQuery. A class can also be used like the id selector to target elements with the difference that a class don't have to be unique and therefore can target multiple elements at the same time.

Here's a short recap from the CSS chapter. You can create CSS classes containing multiple CSS properties if needed. The selector can be declared in a style block in the HTML code or in a separate CSS file (**.css**) which then is linked into the view or the **_Layout.cshtml** layout view.

The **active** CSS class below changes the background color to black and the text color to white for elements decorated with that class.

```
.active {
    background-color:black;
    color:#fff;
}
```

Continuing with the previous example you will run into a CSS property specificity issue. Since the background color was assigned using the **css** method it was added to the **styles** attribute on the elements which mean that it always have precedence over other CSS rules such as the one you are trying to use through the **active** CSS class.

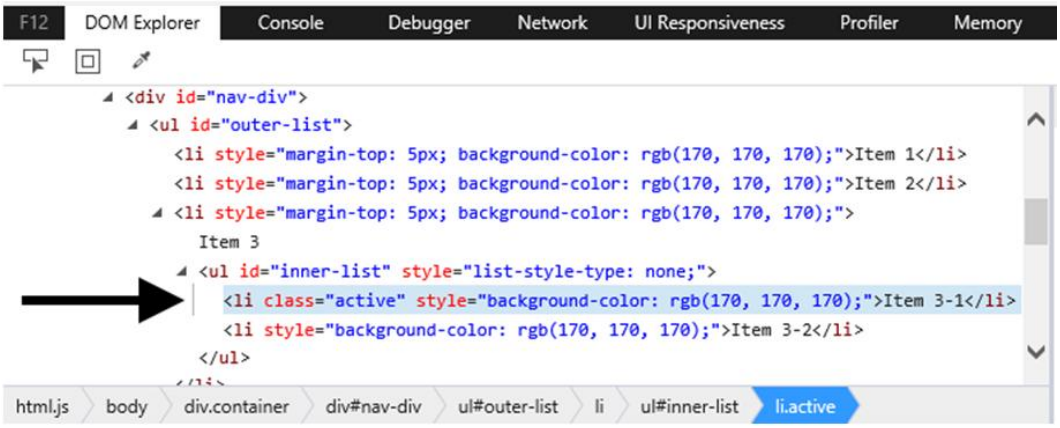
```
// Can cause specificity problems because it is
// assigned to the styles attribute on the elements
$('#nav-div ul li').css('background-color', '#aaa');
```

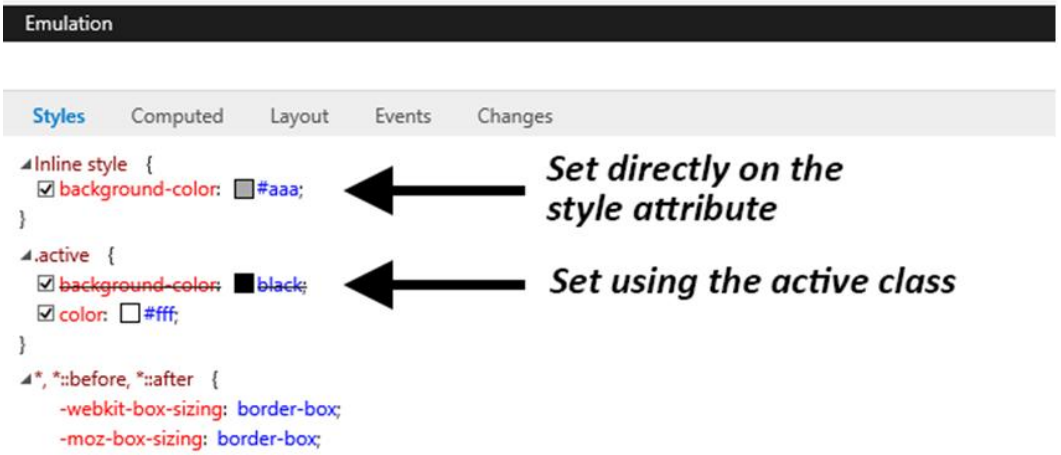
The .addClass function

When you add the **active** class using the **addClass** JQuery method it is added to the **class** attribute of the elements giving it a lower specificity than adding a CSS property to the **style** attribute directly.

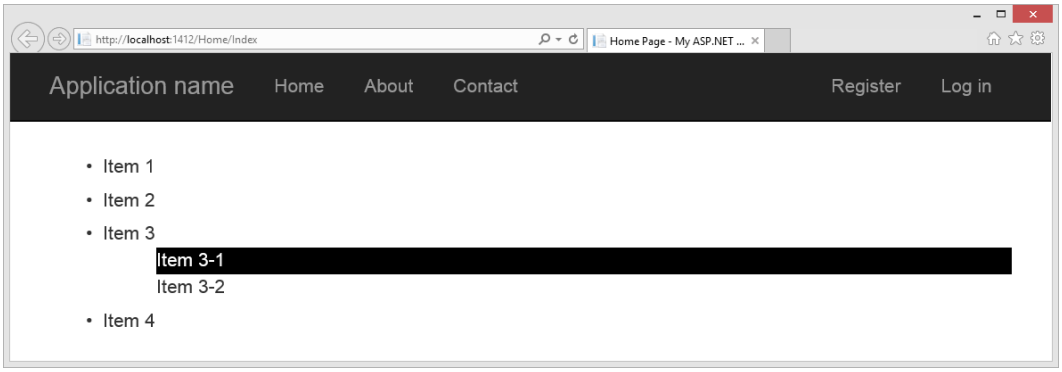
```
$('#inner-list li:first-child').addClass('active');
```

The problem is apparent if you use the **F12** browser tool to examine the CSS settings for the **<li>** element decorated with the **active** class. It has one background color assigned by the **active** class which is overridden by the background color assigned using the **style** attribute. The conclusion is that it is not advisable to set a CSS property with the **css** method if you later intend to override it using a CSS class.





If you remove the previous background color assignment added with the **css** method you can see the intended result.



The .removeClass function

If you want to remove a CSS class from an element you can call the **removeClass** JQuery method passing in the name of the class to remove. The following example removes the **active** class from all **<li>** elements in the inner list.

```
$('#inner-list li').removeClass('active');
```

By removing the **active** class its CSS properties are no longer affecting the element.

The .hasClass function

This method examines an element to find out if a specific class is assigned to it, the method can be used with an if-statement to alter the program flow.

The following example checks if the first immediate child **<li>** element of the unordered list with the **outer-list** id has the **active** class assigned to it. The result is displayed in a pop-up dialog.

```
var isActive = $('#outer-list > li').hasClass('active');
alert(isActive);
```

The .toggleClass function

This method toggles a specific class on an element adding it if it is missing and removing it if it is present. This can be very useful when hovering over an element using the **hover** method to change CSS styles when entering and leaving the element. You can see an example on how to wire up the **hover** event under the headline **.hover()** in the **Events** section.

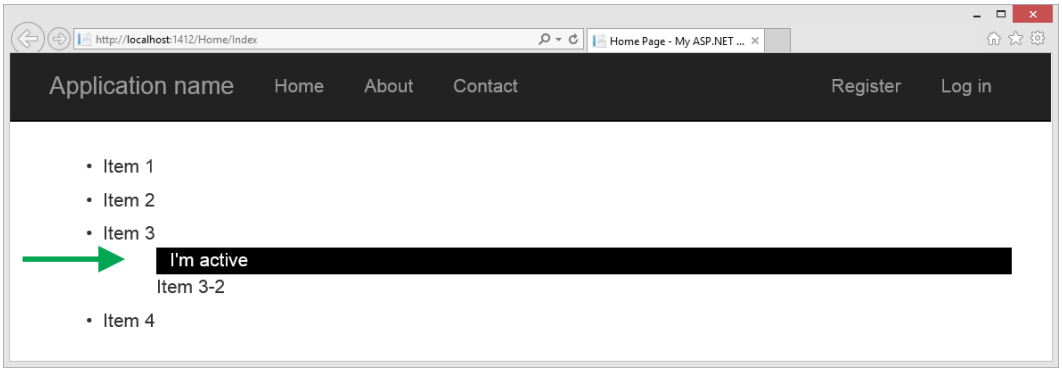
The following example will toggle the **active** class on and off on the ** ** with the id **nav-div** changing its background and text color with the style rules in the active CSS class. You can place the code in a button's **click** event to let the user decide when to toggle.

```
$('#nav-div').toggleClass('active');
```

Class name selector

Add a dot (**.**) in front of the class name in the JQuery selector to select elements based on a class name in the DOM. Continuing with the **active** class example; you could manipulate the active element changing its text to "I'm active" and add 10px padding to the left of the text using the **html** and **css** methods with the **.active** selector.

```
$('.active').html("I'm active").css('padding-left', '10px');
```



Multiple class name selectors

You can use multiple class names with a JQuery selector. By placing a space in between the classes (**.class1 .class2**) the second class (**.class2**) must be nested within an element decorated with the **.class1** class.

If you place the classes next to one another without a space in between (**.class1.class2**) both classes have to be present on the same element for it to be selected in the DOM.

In this example the **selected** class must be present on an element inside another element decorted with the **active** class to be selected in the DOM.

```
<li class="active"><span class="selected">Item 1</span></li>
```

```
$('.active .selected').html("I'm active").css('padding-left', '10px');
```

In this example both the **active** and **selected** classes must be on the same element for it to be selected in the DOM.

```
<li class="active selected">Item 1</li>
```

```
$('.active.selected').html("I'm active").css('padding-left', '10px');
```

By Attribute

You can select one or more elements by targeting an attribute. It is also possible to read and assign an attribute's value. Reading and assigning attribute values using the **data-** attribute can be very useful if you want to store extra data on an element. Any attribute

starting with **data-** is disregarded by the browser when rendering the page but will be added to the element.

Attributes can also be used when wiring up events to elements.

In the following example the button will be wired up to a **click** event and when clicked will add some text from one of the button's attributes to the textbox. It is important to note that the event could have been wired up using HTML events or using an id on the button.

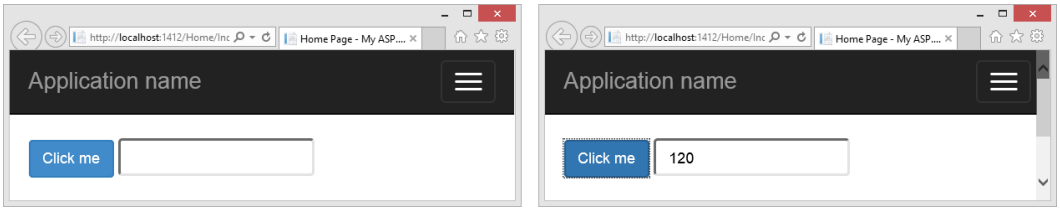
```
<button class="btn btn-primary btn-sm" data-product-id="120" data-
submit-button>Click me</button>
<input type="text" class="input-sm" data-result />
```

The **attr** method fetches a value from or assigns a value to an attribute; pass in the value as a second parameter to the method to assign a value.

The **val** method fetches a value from or assigns a value to an element; pass in the value as a parameter to the method to assign a value.

Use square brackets to target an element by one or more of its attributes. It is possible to target an element with or without specifying the attribute value (**button[data-submit-button]** or **button[data-product-id = "120"]**). Several elements can have the same attribute assigned and the attributes can have different values. One scenario could be a "product details" button which is repeated in a table for each product and when one of the buttons is clicked its id stored in the **data-product-id** attribute is sent to the server to fetch detailed information about that particular product. Another example is that many elements, maybe in a list or a form, can be targeted at the same time and be modified in some way.

```
// Wire up the button click event
$('button[data-submit-button]').click(function () {

    // Read product id from data- attribute
    var prodId = $('button[data-submit-button]').attr('data-product-id');

    // Write the product id to the textbox using a data- attribute
    $('input[data-result]').val(prodId);
});
```



Selector	Description
$(a[title = "some text"])	Select all anchor tags with a **title** attribute containing the exact text "some text".
$(a[title ^= "some text"])	Select all anchor tags where the **title** attribute text begins with "some text".
$(a[title $= "some text"])	Select all anchor tags where the **title** attribute text ends with "some text".
$(a[title *= "some text"])	Select all anchor tags with a **title** attribute where their text contain "some text".

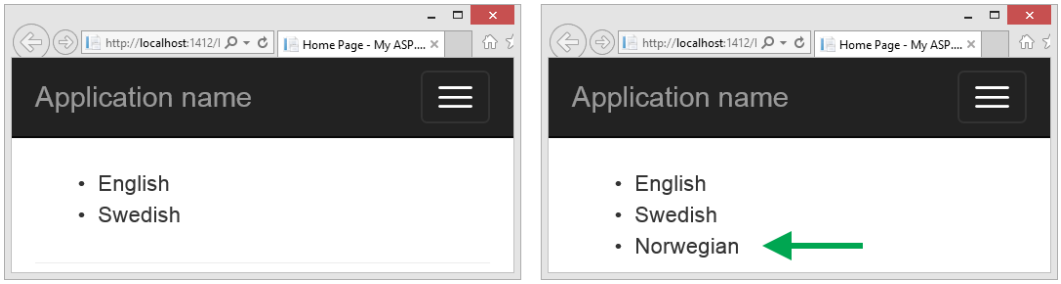
Adding/Removing Elements

The **append** and **prepend** JQuery methods adds elements or HTML dynamically to the page; pass in the element you wish to add as a parameter. It is also possible to remove or hide existing elements by calling either of the **remove** or **hide** methods.

Append

Let's say you have an unordered list of languages and you want to add new language to the end of the list. To achieve this you can call the **append** method.

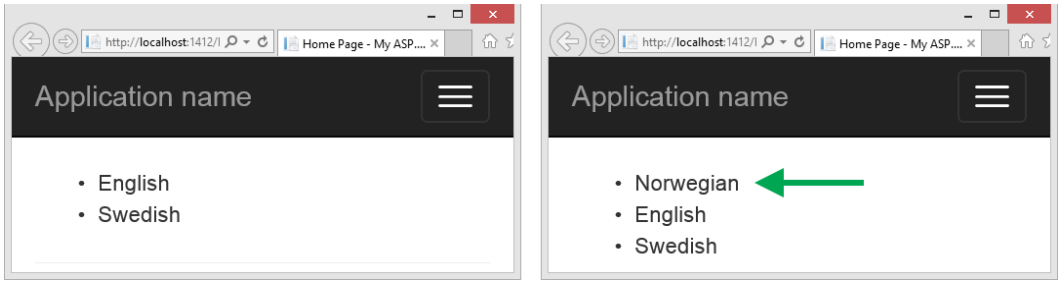
```
<ul id="languages">
   <li>English</li>
   <li>Swedish</li>
</ul>

$('#languages').append('<li>Norwegian</li>');
```



Prepend

Let's say you have an unordered list of languages and you want to add new language to the beginning of the list. To achieve this you can call the **prepend** method.

```
<ul id="languages">
    <li>English</li>
    <li>Swedish</li>
</ul>

$('#languages').prepend('<li>Norwegian</li>');
```



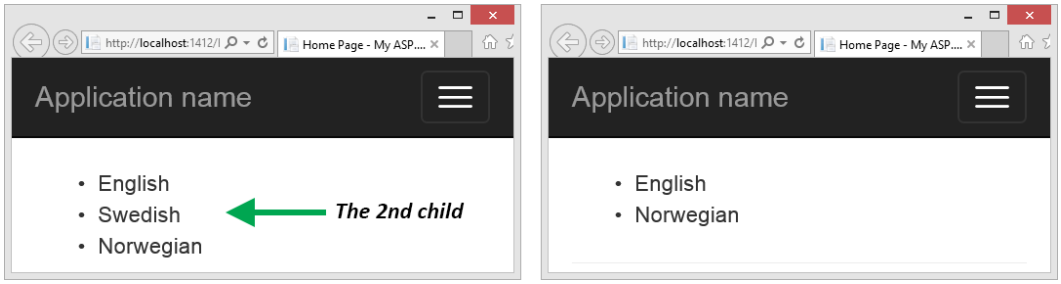
Remove

Let's say you have an unordered list of languages and want to remove a language at the n^{th} position in the list. To achieve this you can target the specific list item using the **:nth-child** selector and the **remove** method.

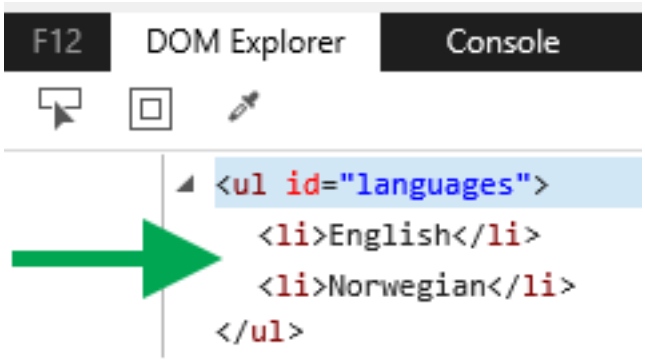
In this example *Swedish* will be removed from the list using the **n^{th}-child** selector. Note that the space between the **id** and **n^{th}-child** selector is important and that the child index starts with 1.

```
<ul id="languages">
    <li>English</li>
```

```
    <li>Swedish</li>
    <li>Norwegian</li>
</ul>

$('#languages :nth-child(2)').remove();
```



If you open the **F12** Web Tool you can see that the element has been removed from the DOM and no longer is available.



Hide/Show

You can use the **hide** and **show** JQuery methods to hide an element temporarily and display it at a later time.

```
<ul id="languages">
    <li>English</li>
    <li>Swedish</li>
    <li>Norwegian</li>
</ul>
```

In this example *Swedish* is hidden using the **hide** method.

```
$('#languages :nth-child(2)').hide();
```

If you open the **F12** Web Tool you can see that the element still exists but is hidden in the DOM. Compare this to the **remove** method described in the previous section which completely removes the element from the DOM.

In this example the hidden node containing *Swedish* is displayed in the list again using the **show** method.

```
$('#languages :nth-child(2)').show();
```

Events

Event is the mechanism for handling user interaction with your web application, it can for instance be mouse movement, the click of a button or hovering over an element. The event declares a callback method to be called when the event is triggered by the user or the system.

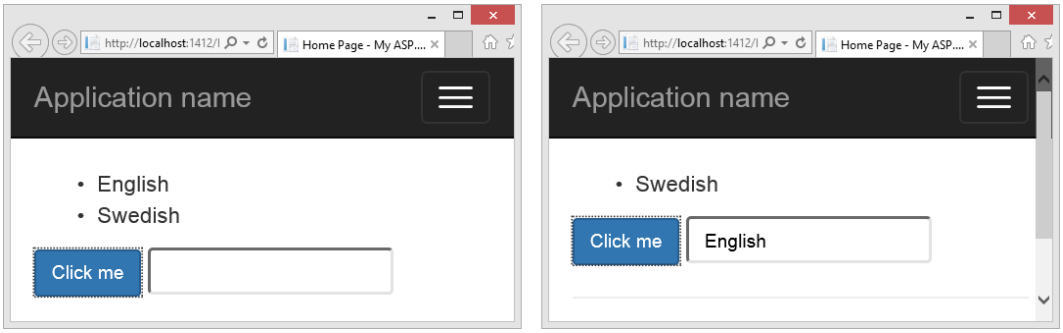
.click()

The **click** event is triggered when the user clicks on an element, usually a link or a button. You can target the element that you want to bind the event to using the previously described selector expressions.

When wiring up the **click** event you can use an anonymous function for the function that will be called when the event is triggered.

In this example the **click** event is bound to the **Click Me** button with the *hideLanguage* **id**. The click event is bound to the button element using the **id** selector. When the user clicks the button in the example below the text of the first item in the list is displayed in the textbox decorated with the **result** id and is then hidden.

```
<ul id="languages">
    <li>English</li>
    <li>Swedish</li>
</ul>

<button id="hideLanguage" class="btn btn-primary btn-sm">
    Click me</button>
<input id="result" type="text" class="input-sm" />


$('#hideLanguage').click(function () {
    var hiddenLanguage = $('#languages :first-child').html();
    $('#result').val(hiddenLanguage);
    $('#languages :first-child').hide();
});
```



.change()

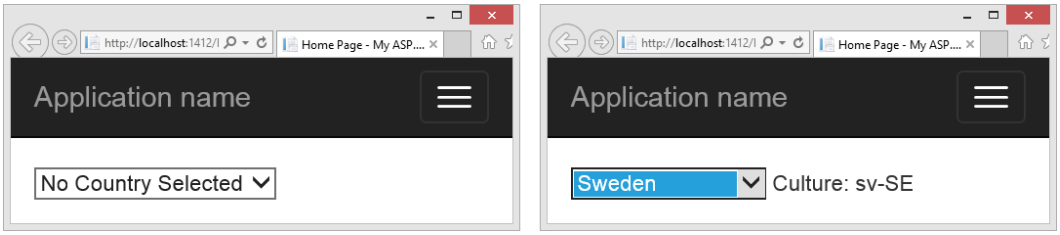
The **change** event is most frequently used with a **<select>** list (drop down), the event is triggered when the user selects an item.

In the example below the **change** event will be triggered when the user select a country in the drop down list. In the function hooked up to the event the value (culture) of that country is displayed in a ** ** element decorated with the **culture** id.

```
<select id="country">
    <option value="">No country selected</option>
    <option value="en-GB">United Kingdom</option>
    <option value="sv-SE">Sweden</option>
    <option value="nn-NO">Norway</option>
</select>

<div id="culture" style="display:inline-block;"></div>
```

The **this** object contains the selected raw DOM element in the drop down list and by wrapping it in a JQuery object using **$()** you can call JQuery functions on it such as **val** which fetches the element's value.

```
$('#country').change(function () {
    $('#culture').html("Culture: " + $(this).val());
});
```



.on()

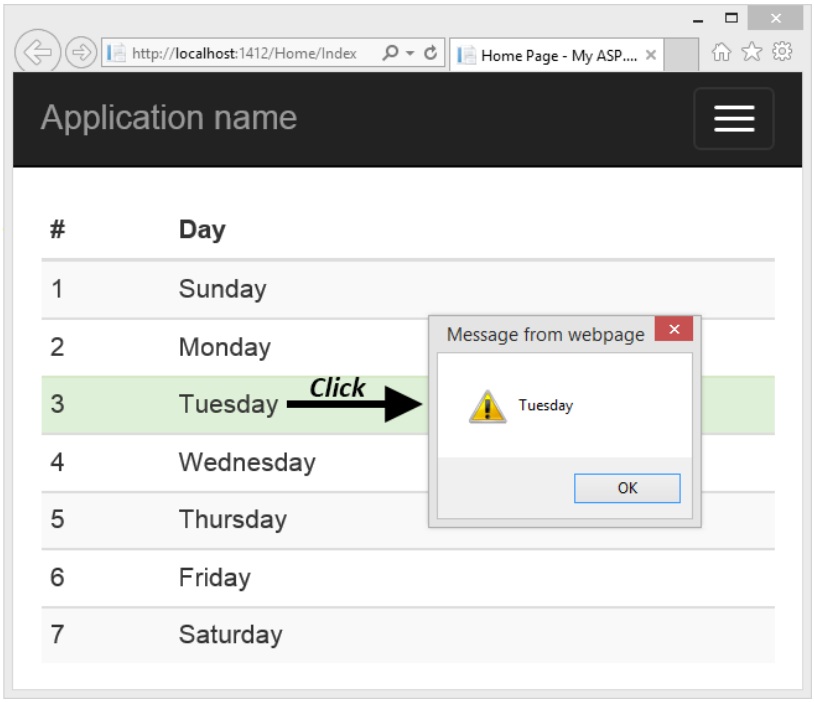
The JQuery **on** method is a way to wire up different events, if you for instance are going to use the **click** event you can use the **click** method or you can achieve the same result with the **on** method. A benefit of using the **on** method is that you can take advantage of the event bubbling that is built in to the browsers. Bubbling means that if the event is not found on the targeted element itself the ancestor elements are searched for that event. This can be a huge memory saver if you are processing a lot of data. Imagine that you are presenting data in a table where you want to enable interaction with the user on row or even cell level by handling the **click** event. Using the **click** method to wire up the **click** event would entail registering the event with every single row or cell which could be a substantial number of times, with 100.000 rows the event would have to be registered 100.000 times! As you can see it adds up quickly, luckily there is a way around this.

By using the bubbling mechanism and the **on** method you can register the event on the table body **<tbody>** element instead of registering it on each cell **<td>** or table row **<tr>** element, this way you only register it once and it can be used by all rows or cells even if they are added dynamically at later time.

Display data from a clicked table cell

In this example the **click** event is registered with the **<tbody>** element but is triggered when the user clicks a table cell; an alert dialog is displayed showing the clicked value. The **<tbody>** element is cashed in a variable for efficiency because it is used multiple times in the code. The **this** object is the raw DOM element for the clicked cell, by wrapping it in a JQuery object with **$()** you get access to JQuery functionality for that element.

```
<table id="weekdays" class="table table-condensed table-striped">
    <thead>
        <tr><th>#</th><th>Day</th></tr>
    </thead>
    <tbody>
        <tr><td>1</td><td>Sunday</td></tr>
        <tr><td>2</td><td>Monday</td></tr>
        <tr><td>3</td><td>Tuesday</td></tr>
        <tr><td>4</td><td>Wednesday</td></tr>
        <tr><td>5</td><td>Thursday</td></tr>
        <tr><td>6</td><td>Friday</td></tr>
        <tr><td>7</td><td>Saturday</td></tr>
    </tbody>
</table>

$(function () {
    // Cache the table body element
    var tbody = $('#weekdays tbody');

    // Wire up the click event for the
    // table cells (<td> elements) of the table
    tbody.on('click', 'td', function () {

        // Display the text of the clicked
        // cell in an alert dialog
        alert($(this).text());
    });
});
```

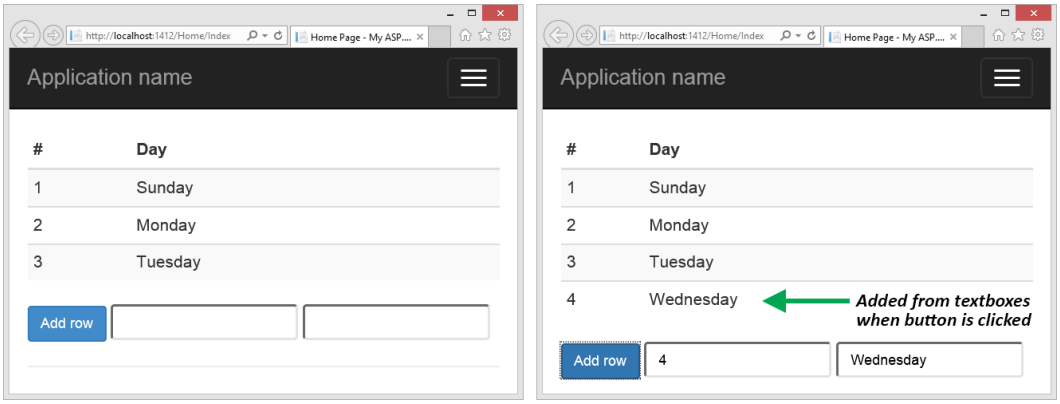


Add a new table row

In this example a new row will be added to the table and when it is clicked the **click** event is triggered even though it was not explicitly registered for that row or its cells. This is possible because the **click** event already is registered with the **<tbody>** element.

The new table row is added when the **Add row** button is clicked and the row values are fetched from the two textboxes.

```
<button id="add-row" class="btn btn-primary btn-sm">Add row</button>
<input id="weekday-id" type="text" class="input-sm"/>
<input id="weekday" type="text" class="input-sm"/>

$(function () {
    $('#add-row').on('click', function () {
        // Add a new row to the bottom of the table
        tbody.append('<tr><td>' + $('#weekday-id').val() +
            '</td><td>' + $('#weekday').val() + '</td></tr>')
    });
});
```

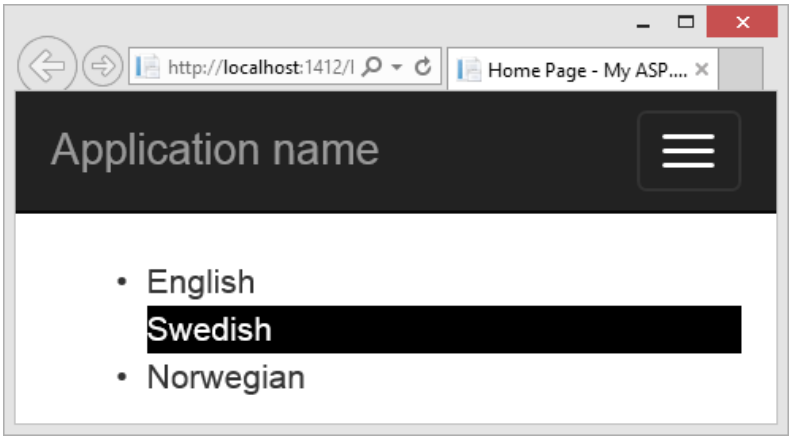


.hover()

Hovering can be used to highlight the element that the mouse pointer is pointing to (hovering over) to make it stand out more. It can also be used to open hidden elements such as a menu that opens when a user points to or clicks on a link.

In this example the list item, **<li>** element, that the mouse pointer hovers over will get a different background and text color and when the mouse pointer leaves the element the original color is restored. The hovering effect can be achieved by adding and removing a specific CSS class on the element. Adding and removing a class can be achieved by calling the **toggleClass** method.

```
<ul id="languages">
    <li>English</li>
    <li>Swedish</li>
    <li>Norwegian</li>
</ul>

$(function () {
    $('#languages li').hover(function () {
        $(this).toggleClass('active');
    });
});
```



Ajax

The *A* in *Ajax* stands for *A*synchronous which mean that you temporarily leave the current thread servicing the web application and call the web server on a separate thread in parallel with the web application thread. This enables the user to interact with the web application while the asynchronous call to the server action (method) is made and when the server call returns its result can be used to update part of (partial view) or the whole web page (view). This is especially useful for long running tasks which you as a developer have no control over such as a web service call to fetch remote data or to fetch data from a database using Entity Framework.

In short, Ajax is a way for the client to execute code on the server without having to reload the entire web page (view) when the call return a result, reloading will occur with a regular synchronous call. When you update a section of the page using a partial view the page won't lose its scroll position.

Here are three ways in which you can use Ajax in a Razor view on the client side:

- @Ajax.BeginForm
- @Ajax.ActionLink
- Client validation

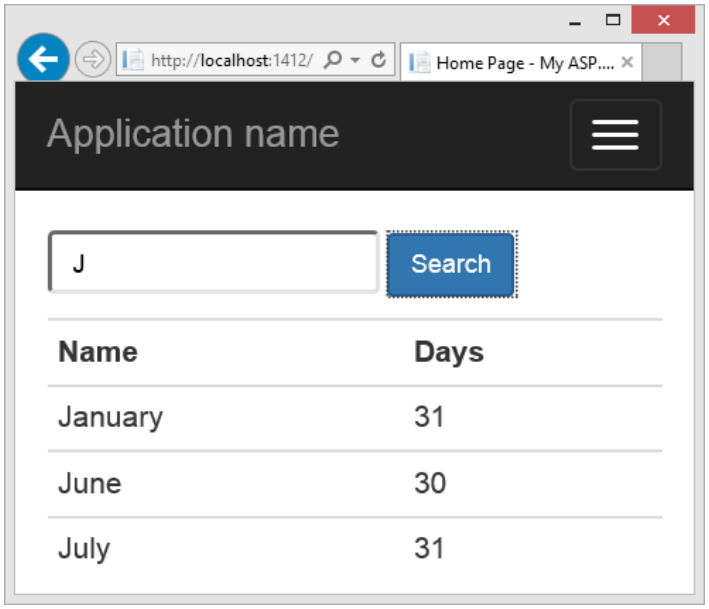
You can also use JQuery to make asynchronous requests to the server which is what you will be learning in this section.

Imagine that you are creating a page which will display an article based on a user selection and only part of the page needs to be updated for the actual article text to be displayed. To achieve this you could do a URL call which will load a new page (or reload the entire existing page) with all of its elements, rendering the page unusable while the page is loading. Or you could do an asynchronous Ajax call to the server fetching only the article text (or HTML) and update only the part of the page displaying the actual article (in a partial view rendered on the page).

Exercise 7-1: Ajax and a partial view

In this exercise you will learn how to make an Ajax call to an MVC action (method) on the server (local host) and display the returned data in a table on the page using a partial view, the image below illustrates what the end result will look like.

The user can enter the beginning of a month's name and click the button, the response back from the server is then displayed in the table. The controller action must also be able to load all of the months and display them in the table synchronously when the page initially loads or when the page is refreshed. The months are stored as **Month** objects in a **List<Month>** to avoid involving a database in this example, in a real world scenario the filtered data could just as easily be fetched from a database table.



The Month Model

First you need to define a class called **Month** which will be the view model determining what data will be rendered to the client page. The **Month** class should be created in the **Models** folder and have two **string** properties the name of the month and the number of days the month have.

1. Right click on the **Models** folder in the Solution Explorer.
2. Select **Add-Class** in the context menu.
3. Name the class **Month** in the **Name** textbox.
4. Add the two **string** properties (**Name** and **Days**) to the class.

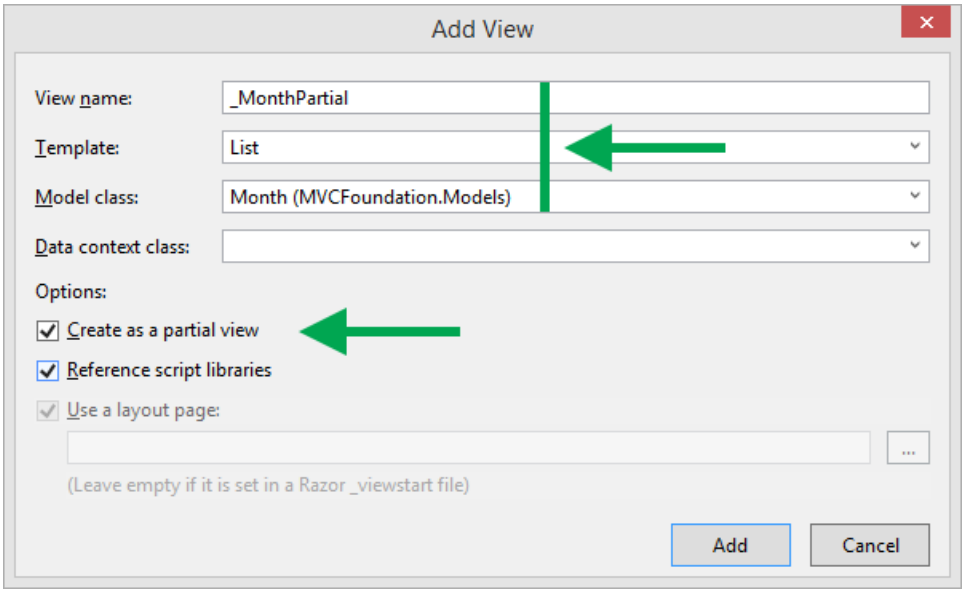
The complete **Month** view model would look like this.

```
public class Month
{
    public string Name { get; set; }
    public string Days { get; set; }
}
```

Adding the partial view _MonthPartial

Next you will create the partial view which will display the months in a table based on the model data passed into the partial view. You can create the **_MonthPartial** partial view from scratch if you like or you can scaffold and alter it (scaffolding is used in the detailed description below). Because you pass in an **IEnumerable<Month>** as the view model you can iterate over the model when displaying the months in the table body.

1. Right click on the **Views-Shared** folder in the Solution Explorer.
2. Select **Add-View** in the context menu and assign the following values:
    a. View Name: **_MonthPartial**
    b. Template: **List**
    c. Model class: **Month**
    d. Check the **Create as partial view** checkbox.
3. Click the **Add** button to create the partial view.



4. Remove the following HTML markup from the partial view. You don't need the links in the table since the user shouldn't be able to add, alter or delete the months displayed in the table.

```
<p>@Html.ActionLink("Create New", "Create")</p>

<th></th>

<td>
    @Html.ActionLink("Edit", "Edit", new { /* id=... */ }) |
    @Html.ActionLink("Details", "Details", new { /* id=... */ }) |
    @Html.ActionLink("Delete", "Delete", new { /* id=... */ })
</td>
```

5. Add the **table-condensed** Bootstrap class to the **<table>** element to make it more compact.
6. Add the attribute **data-table-area** to the **<table>** element. This attribute will be used from the JavaScript code to find and replace the content in the table when the user clicks the **Search** button and the result is returned from the server.
7. Add the attribute **data-table-action-url** to the **<table>** element and assign the controller and action URL (/Home/Index/) that you want the asynchronous call to reach when the user clicks the **Search** button.

```
<table class="table table-condensed"
    data-table-area
    data-table-action-url="/Home/Index/">
```

The finished partial view should have the following markup:

```
@model IEnumerable<MVCFoundation.Models.Month>

<table class="table table-condensed" data-table-area
data-table-action-url="/Home/Index/">
    <tr>
        <th>@Html.DisplayNameFor(model => model.Name)</th>
        <th>@Html.DisplayNameFor(model => model.Days)</th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Name)</td>
            <td>@Html.DisplayFor(modelItem => item.Days)</td>
        </tr>
    }
</table>
```

Alter the Index view

Next you will add the partial view **_MonthPartial** to the **Index** view and change its view model to **IEnumerable<Month>**, this model will be passed as a parameter to the partial view. You will also add an input element with the attribute **data-search-field** added to it, this attribute will later be used to find the textbox and read its value when calling the server. Then you need to add a button with the attribute **data-search-button** added to it, this attribute will later be used to find the button and wire up its click event using JQuery. You can of course use unique ids instead of attributes to target the controls if you like.

1. Open the **Index.cshtml** view in the Solution Explorer.
2. Add the **Month** model as an **IEnumerable** at the very top of the **Index** view.
   ```
   @model IEnumerable<MVCFoundation.Models.Month>
   ```
3. Add the **<input>** textbox with a **data-search-field** attribute to the view.
   ```
   <input type="text" class="input-sm" data-search-field />
   ```

4. Add the **<button>** element with a **data-search-button** attribute to the view.
   ```
   <button class="btn btn-primary btn-sm" data-search-
   button>Search</button>
   ```
5. Add the partial view to the **Index** view and pass in the model to it so that the table can be rendered when the view is created.
   ```
   @Html.Partial("_MonthPartial", Model)
   ```

The **Index** view should now contain at least the following HTML markup:

```
@model IEnumerable<MVCFoundation.Models.Month>

<input type="text" class="input-sm" data-search-field />
<button class="btn btn-primary btn-sm" data-search-
button>Search</button>

@Html.Partial("_MonthPartial", Model)
```

Adding the month list to the controller

Next you will add the **months** collection to the controller. Remember that this is example code and that the months most likely would be fetched from a database table or a cached object to improve performance in a real world scenario, it certainly wouldn't be hard coded in the controller.

The model sent to the view should be an **IEnumerable<Month>** which mean that the action method must return a list or an enumeration of months. To make this example less complex you will store the months in a **List<Month>** called **months** directly in the **HomeController** class, it will be queried later using LINQ when the user do a search.

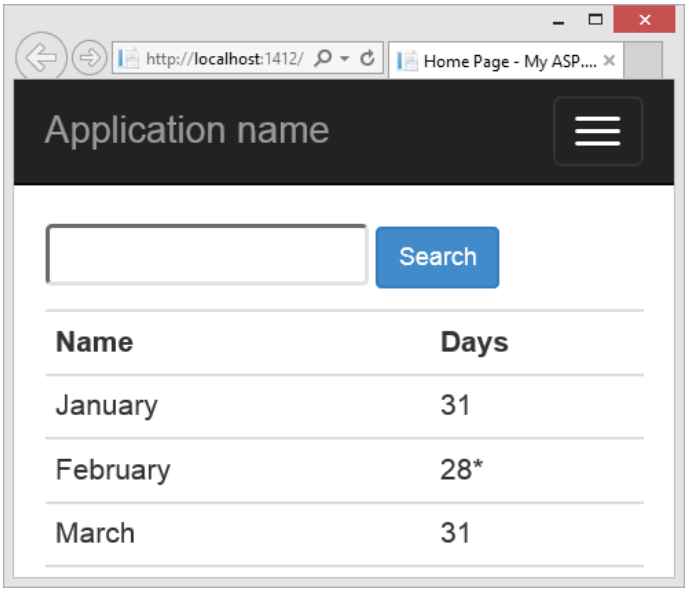
1. Open the **HomeController** class in the Solution Explorer.
2. Add a **List<Month>** called **months** to the **HomeController** class and add the months as instances of the **Month** class.
   ```
   public class HomeController : Controller
   {
       List<Month> months = new List<Month> {
           new Month { Name = "January", Days = "31" },
           new Month { Name = "February", Days = "28*" },
           new Month { Name = "March", Days = "31" },
           new Month { Name = "April", Days = "30" },
           new Month { Name = "May", Days = "31" },
   ```

```
        new Month { Name = "June", Days = "30" },
        new Month { Name = "July", Days = "31" },
        new Month { Name = "August", Days = "31" },
        new Month { Name = "September", Days = "30" },
        new Month { Name = "October", Days = "31" },
        new Month { Name = "November", Days = "30" },
        new Month { Name = "December", Days = "31" }
    };
```

Altering the Index action to render the model with the view

Next you will alter the **Index** action of the **HomeController** to send the **months** collection to the **Index** view when it is rendered. You do this by passing in the list of months you added to the controller as a parameter to the **View** method which renders the view.

1. Open the **HomeController** class in the Solution Explorer.
2. Change the return command to render the view with the **months** collection as its data.
   ```
   return View(months);
   ```
3. Start the application and verify that the months are listed in on the page. *The search button will not work yet*.



Adding the Ajax search functionality

Next you will add a JavaScript file in which you wire up the **click** event to the **Search** button, fetch the value from the textbox, call the server asynchronously using Ajax and display the result in the partial view's table.

Create the JavaScript file

Add a JavaScript file called *AsyncSearch.js* to the **Scripts** folder and add a link to it in the **_Layout** view to make the code accessible from any view in the application.

1. Right click on the **Scripts** folder in the Solution Explorer.
2. Select **Add-JavaScript File** (or **Add-New Item** and select **JavaScript File** in the list) and name it **AsyncSearch** in the textbox.
3. Open the **_Layout.cshtml** view in the Solution Explorer.
4. Add a link to the **AsyncSearch.js** file after the last **Scripts.Render** call in the **_Layout** view.

```
@Scripts.Render("~/Scripts/AsyncSearch.js")
```

Cache the components in variables and wire up the click event

Next you will add the document **ready** function *$(function(){…});* in the **AsyncSearch** JavaScript file and add three variables to it. The first variable called **table** will cache the table that you target using it's **data-table-area** attribute. The second variable named **searchbutton** will be used when wiring up the **click** event to the **Search** button, you can target the button by using the **data-search-button** attribute. The third variable named **searchfield** will cache the **search** textbox which you can target using the **data-search-field** attribute.

The variables will be used later when the asynchronous call is made.

1. Open the **AsyncSearch.js** file in the Solution Explorer.
2. Add the document ready function to the JavaScript file.

```
$(function () {
    // The code goes here
});
```

3. Add the **table** variable to the document ready function.

```
var table = $("table[data-table-area]");
```

4. Add the **searchbutton** variable in the document **ready** function and wire up the **click** event to the button. You will add the **onSearchAsyncClick** function later, it will be called when the button is clicked.
```
var searchbutton = $("button[data-search-button]")
.click(onSearchAsyncClick);
```
5. Add the **searchfield** variable which references the **search** textbox in the document **ready** function.
```
var searchfield = $("input[data-search-field]");
```

The code in the JavaScript file should look like this:

```
$(function () {
    var table = $("table[data-table-area]");
    var searchbutton = $("button[data-search-button]").click(
        onSearchAsyncClick);
    var searchfield = $("input[data-search-field]");
});
```

Add the onSearchAsyncClick function

This function contains the code for the asynchronous call which is triggered when the user clicks the **Search** button. The **onSearchAsyncClick** function is wired up to the **search** button's **click** event and will be executed when user clicks the button. Store the value from the **data-table-action-url** attribute in a variable called **url**. Fetch the value that the user has entered into the **search** textbox and store it in a variable called **value**. To be on the safe side you should check that the **url** variable has a value before making the asynchronous call to the server.

You can use the **$.post** JQuery function to make the asynchronous call to the server passing in the value from the **value** variable as a parameter called **searchTerm**. In the anonymous **success** function you then replace the current content of the **_MonthPartial** partial view with the result returned from the **Index** server action method in the **Home** controller using the **html** method on the **table** variable.

You can chain on the **fail** JQuery function to the **$.post** success function to handle errors that might occur.

1. Open the **AsyncSearch** JavaScript file located in the **Scripts** folder in the Solution Explorer.
2. Add a function called **onSearchAsyncClick** to the document **ready** function under the variables you added earlier.
```
function onSearchAsyncClick() {
    // The code goes here
}
```
3. Store the value from the table's **data-table-action-url** attribute in a variable called **url**.
```
var url = table.attr("data-table-action-url");
```
4. Store the textbox value (the user input) in a variable called **value**.
```
var value = searchfield.val();
```
5. Check that the **url** variable has a value and make the asynchronous call to the server using the **$.post** JQuery function. Note that the **url** variable is used to send in the **/Home/Index/** URL to the **$.post** function and the **value** variable is sent with the call as a parameter named **searchTerm** inside an anonymous object. The anonymous function declared within the **$.post** function will handle a successful result and replace the existing data in the partial view with the fetched data. The **fail** method chained to the **$.post** success method handles any error that might occur by displaying an alert dialog.
```
if (url != "") {
    // Make an asynchronous call to the server passing in
    // the value from the textbox as the search term
    $.post(url, { searchTerm: value }, function (data) {
        // Update the partial view with the result
        table.html(data);
    }).fail(function (xhr, status, error) {
        // Handle errors
        alert("Error");
    });
}
```

The complete **onSearchAsyncClick** function:

```
function onSearchAsyncClick() {
    // Get the action url to call on the server
    var url = table.attr("data-table-action-url");
```

```
    // Get the value from the search textbox
    var value = searchfield.val();

    // Only call the server if there is a url present on the table
    if (url != "") {
        // Make an asynchronous call to the server passing in
        // the value from the textbox as the search term
        $.post(url, { searchTerm: value }, function (data) {
            // Update the partial view with the result
            table.html(data);
        }).fail(function (xhr, status, error) {
            // Handle errors
            alert("Error");
        });
    }
}
```

Altering the Index action to handle asynchronous calls

The last thing you need to do before using the search functionality is to handle any asynchronous Ajax calls in the **Index** action of the **Home** controller. Add a **string** parameter called **searchTerm** to the **Index** action method; it will receive the value from the JavaScript **value** variable through the **searchTerm** parameter in the anonymous object of the **$.post** method.

Check that the incoming call is an Ajax request by calling the **IsAjaxRequest** method on the .Net **Request** object inside the **Index** action method. Return the full view with the model data if it isn't an Ajax request otherwise use LINQ to filter out the moths in the **months** collection; use the **searhTerm** parameter passed into the action to filter out the months and return the **_MonthPartial** partial view with the filtered search result using the **PartialView** method.

1. Open the **HomeController** class located in the **Controllers** folder in the Solution Explorer
2. Add the **searchTerm string** parameter to the **Index** action method.
   ```
   public ActionResult Index(string searchTerm)
   ```
3. Check if it is an Ajax call by calling the **IsAjaxRequest** method on the **Request** object.
   ```
   if (Request.IsAjaxRequest())
   {
   ```

```
    // The filtering code goes here
}
```

4. Provided it is an Ajax call, filter out the months with names starting with the value passed in through the **searchTerm** parameter and return the **_MonthPartial** partial view with the filtered result as its model. Use the **ToLower** method to ensure matching values.

```
var model = months.Where(m =>
m.Name.ToLower().StartsWith(searchTerm.ToLower()));

return PartialView("_MonthPartial", model);
```

5. If it is not an Ajax call then return the full view with all the months.

```
else
    return View(months);
```

Now the search functionality is complete and you can try it out. Enter the string 'Ju' in the textbox and click the button, 'June' and 'July' should be displayed in the table. Enter different values and verify that the table content changes without reloading the whole page when you click the button.