

Arnaud Weil



# Learn

# ASP.NET MVC

# Learn ASP.NET MVC

Be ready for coding away next week  
using ASP.NET MVC 5 and Visual Studio  
2015

Arnaud Weil

This book is for sale at <http://leanpub.com/aspnetmvc>

This version was published on 2017-07-07



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2017 Arnaud Weil

*To my wonderful wife and kids. Your love and support  
fueled this book.*

# Contents

<b>Introduction</b> . . . . .	<b>i</b>
What this book is not . . . . .	i
Prerequisites . . . . .	i
How to read this book . . . . .	ii
Tools you need . . . . .	ii
Source code . . . . .	ii
 <b>1. What is ASP.NET MVC and why use it</b> . . . . .	 <b>1</b>
1.1 What is it ? . . . . .	1
1.2 Why use it ? . . . . .	1
1.3 Competing technologies . . . . .	2
 <b>2. Creating our Web Site</b> . . . . .	 <b>4</b>
 <b>3. ASP.NET MVC inner workings</b> . . . . .	 <b>11</b>
3.1 Principles . . . . .	11
3.2 View . . . . .	12
 <b>4. Create an application and modify the home page</b> <b>17</b>	
4.1 Do-it-yourself 1 - Create the application .	17
4.2 Do-it-yourself 2 - Change the home page	19

## CONTENTS

<b>5. Razor</b>	<b>20</b>
<b>6. Understanding ASP.NET MVC</b>	<b>21</b>
<b>7. Typing things up</b>	<b>22</b>
<b>8. Updating server data</b>	<b>23</b>
8.1 Action parameters	23
8.2 Word of caution about URLs	25
8.3 Do-it-yourself 8 - Display product details	27
8.4 HTTP Post parameters	28
8.5 Passing a full blown object	29
8.6 Sit and watch - Basic product calculator	31
8.7 Do-it-yourself 9 - Add a search box to the products list	37
<b>9. Updating data scenario</b>	<b>38</b>
9.1 Steps	38
9.2 Controller	38
9.3 Automated generation of controller and views	40
9.4 Do-it-yourself 10 - Create the products management back-office	45
<b>10. Doing more with controllers and actions</b>	<b>46</b>
<b>11. Basic security</b>	<b>47</b>
<b>12. Going further</b>	<b>48</b>
<b>Do-it-yourself Cheat Sheet</b>	<b>49</b>

## CONTENTS

<b>Definitions . . . . .</b>	<b>50</b>
------------------------------	-----------

# Introduction

## What this book is not

I made my best to keep this book small, so that you can learn ASP.NET MVC quickly without getting lost in petty details. If you're looking for a reference book where you'll find answers to all the questions you may have within the next 4 years of your ASP.NET MVC practice, you'll find other heavy books for that.

My purpose is to swiftly provide you with the tools you need to code your first ASP.NET MVC application and be able to look for more by yourself when needed. While some authors seems to pride themselves in having the thickest book, in this series I'm glad I achieved the thinnest possible book for my purpose. Though I tried my best to keep all of what seems necessary, based on my 14 years experience of teaching .NET.

I assume that you know what ASP.NET MVC is and when to use it. In case you don't, read the following [What is ASP.MVC](#) chapter.

## Prerequisites

In order for this book to meet its goals, you must :

- Have basic experience creating applications with .NET and C#
- Have working knowledge of HTML
- Know what a Web application is

## How to read this book

This book's aim is to make you productive as quickly as possible. For this we'll use some theory, several demonstrations, plus exercises. Exercises appear like the following:



Do it yourself: Time to grab your keyboard and code away to meet the given objectives.

## Tools you need

The only tool you'll need to work through that book is Visual Studio 2015. You can get any of those editions:

- Visual Studio 2015 Community (free)
- Visual Studio 2015 Professional

## Source code

All of the source code for the demos and do-it-yourself solutions is available at <https://bitbucket.org/epobb/aspnetmvc>



It can be downloaded [as a ZIP file<sup>1</sup>](#), or if you installed GIT you can simply type:

```
git clone https://bitbucket.org/epobb/aspnetmvc.git
```

---

<sup>1</sup> <https://bitbucket.org/epobb/aspnetmvc/get/67fdbfbe516a.zip>

# **1. What is ASP.NET MVC and why use it**

This book assumes that you (or your boss) decided to use ASP.NET MVC knowing what it is. In case you do, you can safely ignore this chapter. But if you don't, read this chapter before reading the book and you'll be good to go.

## **1.1 What is it ?**

In a nutshell, ASP.NET MVC is a technology used to create Web applications.

Web applications are used with a browser. Example of Web applications are Facebook, Google, and in fact most of the services you use. When you enter an `http://something` url in your browser, you get a Web application.

Simply put, ASP.NET MVC can be used to create a Web application like Facebook. Or a store, which is what we do in this book's exercises.

## **1.2 Why use it ?**

In case you know .NET and need to create a Web application, using ASP.NET makes sense since you can reuse

your knowledge of the .NET Framework (or .NET Core) and language abilities (like C# and VB.NET).

There are several technologies inside of ASP.NET you may consider:

- ASP.NET Web Forms: good for creating small and large Web applications when you know almost nothing to HTTP and HTML, but have experience developing client applications (for instance Windows Forms or WPF).
- ASP.NET Web Pages: good for creating small Web applications without bothering with the structure.
- ASP.NET MVC: good for creating large Web applications when you know HTTP and HTML.
- ASP.NET Web API: good for creating REST APIs.

Though you can mix all of those technologies in a single ASP.NET Web application, choosing just one is a reasonable choice. Mixing several technologies would be done when you need to maintain for instance ASP.NET Web Forms but want to add new features using ASP.NET MVC.

## 1.3 Competing technologies

There are many technology stacks used to create Web applications. On a technological standpoint the following stacks would for instance allow to develop applications in a way similar to ASP.NET MVC :

- Node.JS + Express
- Ruby on Rails
- Meteor

Selecting one technology or another can be debated for a while. It often boils down to beliefs or preferences, but there can be good reasons. For instance, if you know JavaScript and HTTP but nothing about .NET, you'll probably get an easier time with Node.JS + Express than with ASP.NET MVC.

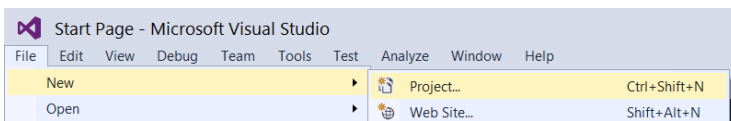
## 2. Creating our Web Site

Here's how we create an ASP.NET MVC Application.

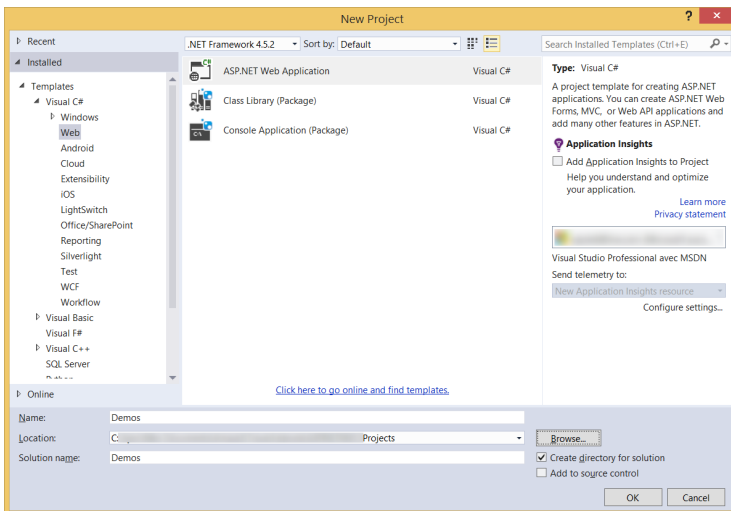
Remember: you watch and memorize this, don't try to do it now. Please bear with me, I promise you'll get fun a bit later.

Let's start Visual Studio 2015 and select `File / New / Project` from the menu. I know `New Web Site` sounds tempting but don't use it, that kind of site doesn't scale up well once you get to real applications.

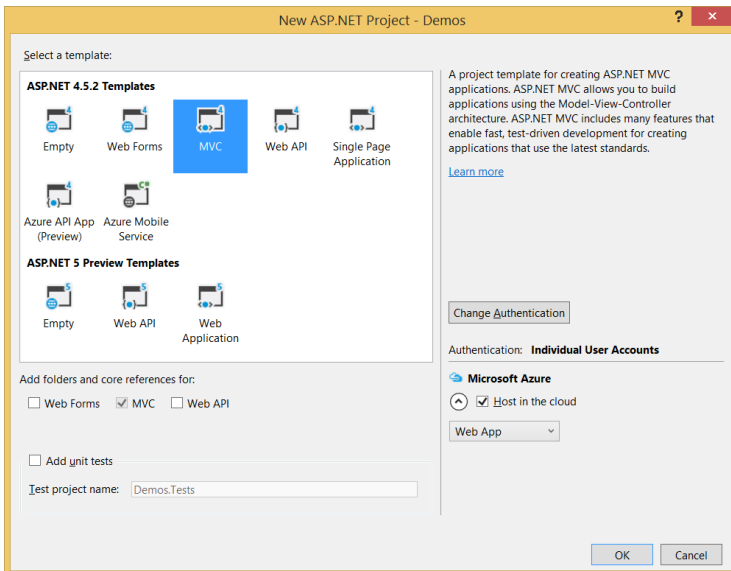
I'm taken to the `New project` dialog box:



From the left menu I'll select `Visual C#` since it's the most common coding language for `.NET`, then `Web`. In the middle pane I'll select `ASP.NET Web Application` and in the right pane I'll ensure the `Add application Insights to Project` is unselected (because I don't need them). In the lower part of the dialog box I'll type the name of my project: `Demos`.



Now I click the OK button and I'm taken to a second dialog: New ASP.NET Project. Time to select the features we want. I'll select MVC and keep the other default options as shown on that screenshot:

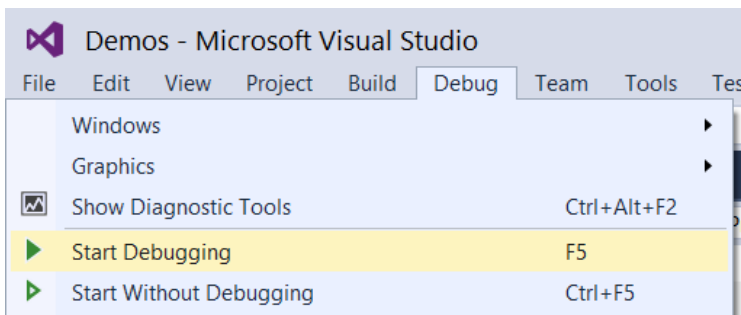


Just a word on the other options, in case you wonder. Add unit test can be unchecked, because it's easy to add unit testing later on when we need it. Host in the cloud is checked and Web app is selected since this will provide me an easy-to-deploy free hosting from Microsoft Azure. And having the Authentication: Individual User Accounts ensures that it's dead-easy to get a local database (or SQL Server with one easy change) based authentication for my application.

Now I click the OK button of the New ASP.NET Project

dialog. I'm taken to a `Configure Microsoft Azure Web App` dialog. This is for my application's later hosting on the Web. I'll just click the `Cancel` button since I want to configure that later – or never in case I deploy my application elsewhere.

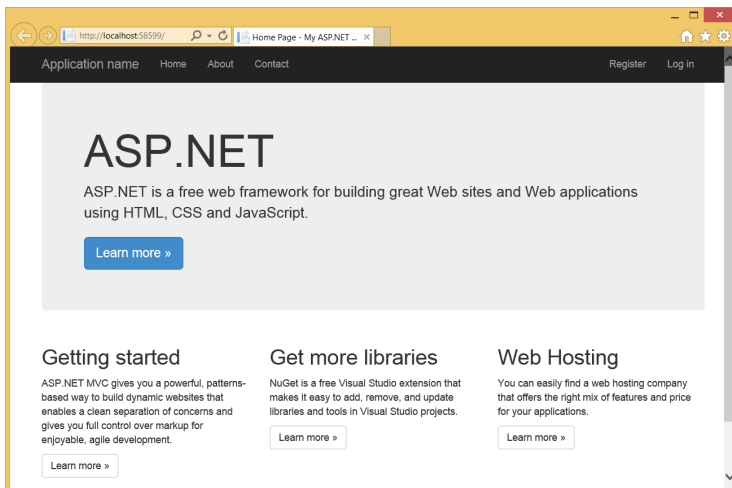
That's it for now: my application is ready. In order to run it inside my browser, I'll select `Debug / Start Debugging` from the menu, or use the `F5` shortcut:



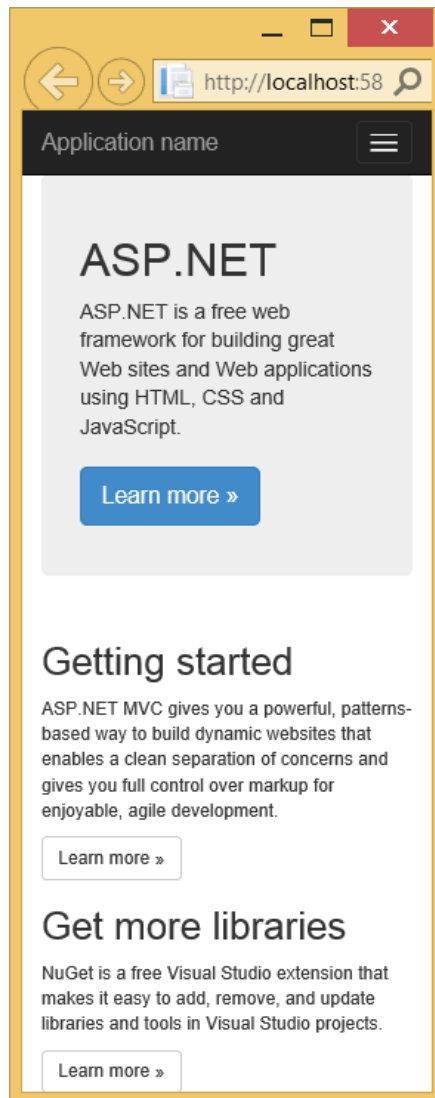
When I do this, Visual Studio starts a local Web server (IIS Express), runs my browser and points it to the URL of my site on that server, and finally attaches to the running code in `debug` mode so that it can catch any exception or show the code when I read a breakpoint.

This is what I get in my browser:





What's nice is that the site is responsive (the default template uses Bootstrap):



That application also has a menu above, with links that take me to almost empty “About” and “Contact” pages. Also note that we already have fully-functional “Log in” and “Register” links for user authentication. If I use them, a local SQL Server database will be created and used for storage.

Well, we have kind of a nice startup after just a few clicks. Now let’s see some theory before we can extend our application. Don’t worry, I’ll keep it short: this book focuses on getting you up and running quickly.

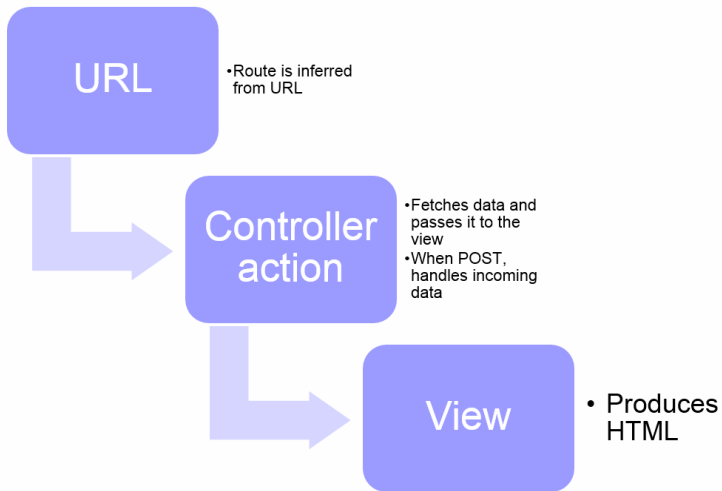
## 3. ASP.NET MVC inner workings

### 3.1 Principles

When our browser queries an ASP.NET MVC URL, there are three elements that work together in order to produce an HTML page:

- A *View* produces the HTML;
- A *Controller*:
  - fetches the data and provides it to the view
  - selects the view
- A *Route* selects the controller.

That's an easy process. Here's a schema of it:



## 3.2 View

Once inferred from the URL, the view is looked for into the following paths:

- Views/[Controller]/[Action].aspx
- Views/[Controller]/[Action].cshtml
- Views/Shared

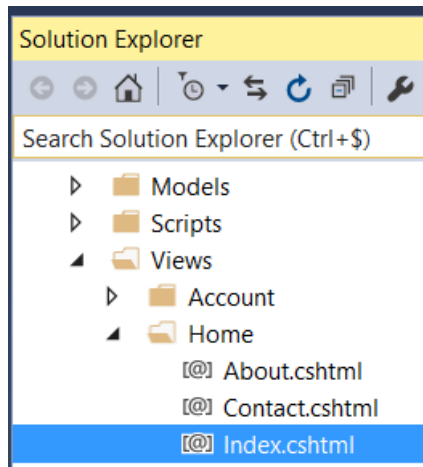
When coding the view, you can choose between one of two syntaxes (or add another one):

- ASPX
- Razor

We'll use Razor since it's a concise view language crafted specifically for ASP.NET MVC. ASPX is a syntax that will sound familiar to those coming from ASP.NET Web Forms.

Before we begin learning about the Razor syntax, we're going to have a look at what's part of the Visual Studio template that was created.

Let's take a look at the project structure. For this I'll use the [Solution Explorer](#).



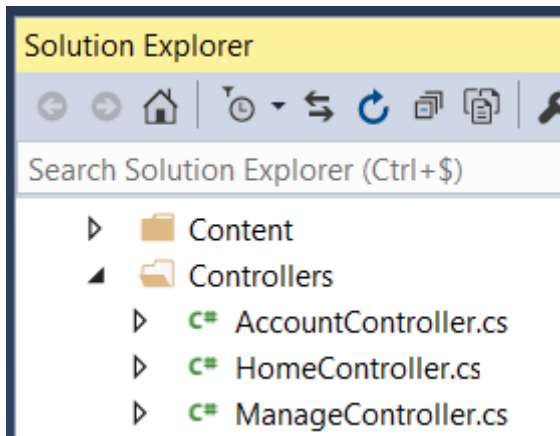
You can see that there is a "Views" folder containing a "Home" sub-folder, inside of which there is an "Index.cshtml" file. As seen above, when ASP.NET looks for the view corresponding to the "Index" action of the "Home" controller, it will get that `Views\Home\Index.cshtml` file. Sound abrupt? Bear with me.

Let's open the `Views\Home\Index.cshtml` file. Apart for the

first 3 lines, it contains pure HTML markup. That's the markup that will be rendered to the browser. And it is also Razor code.

So you just learned a secret: Razor code is basically HTML code. In fact, it's HTML in which we'll add some special statements using the @ mark. Yeah, I know you're beginning to love ASP.NET MVC, or at least Razor.

Inside the [solution](#), we can also find a `Controllers\HomeController.cs` file:



Let's open that file.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    // ...
}
```

You can see that the class is named `HomeController`, and it contains a public method named `Index`. When ASP.NET looks for the view corresponding to the “Index” action of the “Home” controller, it will get this method.

The “Index” method contains a simple `return View();` statement. It means that the default view should be returned for that action.

We’re almost there. Now let’s have a look at the `App_Start\RouteConfig.cs` file. It contains the following code:

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

This is the routing configuration. See the `url` property? It states how ASP.NET MVC will parse incoming URLs. As of



now this is the only line, but you may add as many as you wish here.

What `url: "{controller}/{action}/{id}"` means is that if someone types the `http://mysite/home/index` URL in their browser ASP.NET will invoke the `Index` action from the `Home` controller. It also states that if someone types the `http://mysite` URL in their browser ASP.NET will invoke the `Index` action from the `Home` controller because they are the defaults.

Got it? Now let's sum up what happens for a sample request:

1. Someone types `http://mysite/home/index` in their browser.
2. ASP.NET MVC understands it has to look for the `Index` action from the `Home` controller, so it invokes the `Index` method from the `HomeController` class.
3. The `return View();` statement means the ASP.NET needs to return the default view.
4. ASP.NET fetches the `Views\Home\Index.cshtml` file and renders it to the browser. It's HTML which can be enhanced using the Razor syntax.

Easy, isn't it? Well that's almost all there is to ASP.NET MVC. Read those four steps again until you get them right, because they are the backbone of the ASP.NET process.

Time to put in practice all of that stuff you just learned. And I bet your fingers are itchy for coding.

## 4. Create an application and modify the home page

Do-it-yourself time! You should do the following two exercises. If having a hard time, you can look at the detailed steps in the [do-it-yourself cheat sheet at the end of this book](#), but I'm sure you won't need it.

### 4.1 Do-it-yourself 1 - Create the application



Create a new ASP.NET MVC application project. Ensure that authentication will be done using accounts from a database (that's the default option).

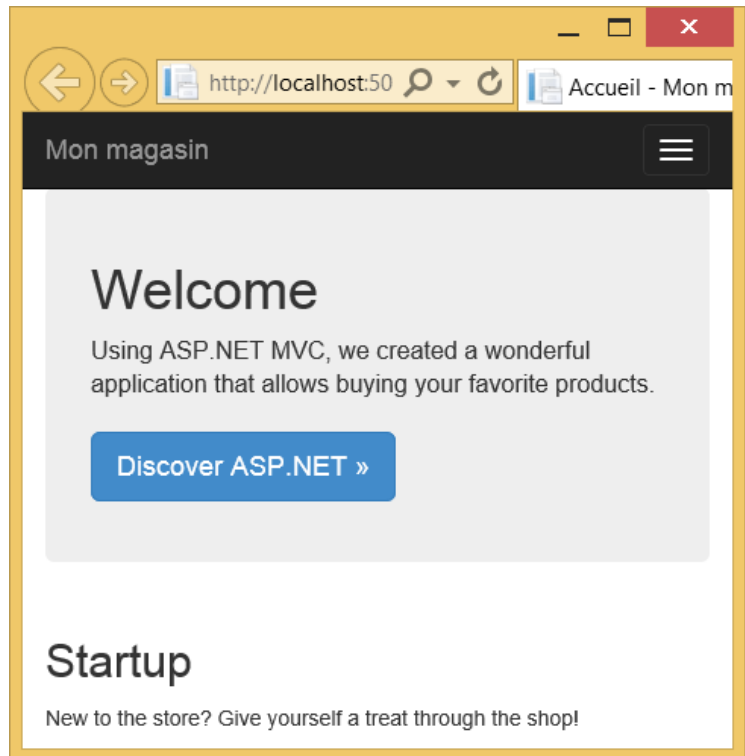
[Step-by-step solution at the end of this book](#)



## 4.2 Do-it-yourself 2 - Change the home page



Change the home page so that it displays a welcome message and a “Startup” section. As a result it should look like the following:



Step-by-step solution at the end of this book

## 5. Razor



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

## 6. Understanding ASP.NET MVC



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

## 7. Typing things up



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

# 8. Updating server data

## 8.1 Action parameters

When updating server data, you need to pass information from the browser to your application. This is simply handled by the routing system which will call your actions with parameters.

Let's look again at the default route in the `Global.asax` file

```
RouteTable.Routes.MapRoute(..., "{controller}/{action\
}/{id}",
    new { "...", id=UrlParameter.Optional } );
```

Note the `{id}` part? It means that anything coming after the action name (and separated from it by a slash sign) will be considered as an `id` parameter. And that this parameter is optional.

In order to get this parameter, all you have to do is add a parameter with the same name to your action method. For instance :



```
public class ProductsController : Controller
{
    public ActionResult Action1(string id)
    { ... }
}
```

Considering the default route, this action can be called simply by typing the following URL :

`http://site/products/action1/abcd`

In that case, our `ProductsController.Action1` method will be invoked with a value of `abcd` for its `id` parameter.

We could add as many parameters as needed to our action and modify the route accordingly. Needless to say that parameters should be short, since they add up to the URL length.

As far as the type of the parameter is concerned (here we used `string`), ASP.NET MVC gracefully handles any conversion. So we could have declared our action with an `id` parameter of type `int` for instance. Of course, `abcd` would be rejected if we did so.

Whatever happens, we can also pass action parameters using query parameters. Consider for instance the following action :

```
public class ProductsController : Controller
{
    public ActionResult Rename(int id, string newName)
    { ... }
}
```

In order to invoke it with an `id` value of 15 and a `newName` value of `sandwich`, we can simply use the following URL

`http://site/products/rename/15?name=sandwich`

Note that `sandwich` is passed as a query parameter since it's not part of the route definition. We could also modify the route definition - or add a new one - in order to make it part of the URL.

## 8.2 Word of caution about URLs

As we saw earlier, route definitions may change later, for instance in order to improve SEO or provide user-friendly URLs. Which means it's important not to rely on them in your views.

The following view code will break if the route change, so avoid using it :

```
@* bad *@
<a href="/home/index/3">Some page</a>
```

Instead, you should prefer :

```
@* good *@  
@Html.ActionLink("Some page", "Index", "Home", new { \  
id=3 }, null)
```

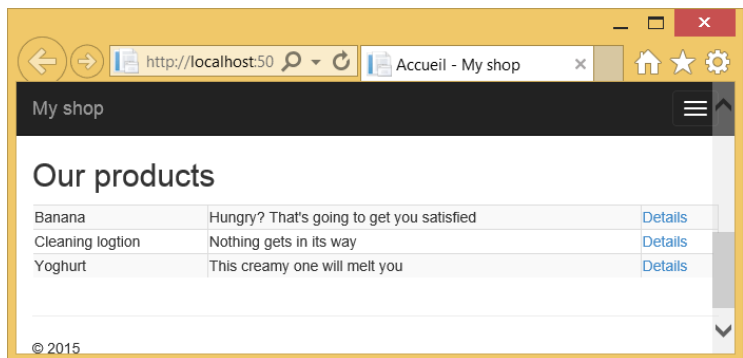
or even :

```
@* good *@  
<a href="@Url.Action("Index", "Home", new { id=3 })">\  
Some page</a>
```

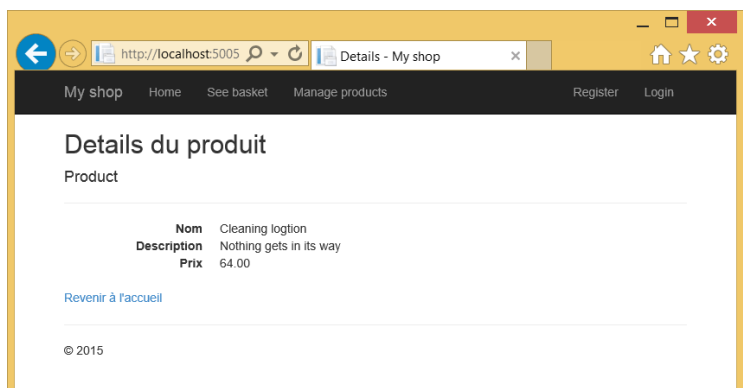
## 8.3 Do-it-yourself 8 - Display product details



You must modify the Home/Index view so a *Details* link is displayed next to each product like in the following screenshot :



Now, make sure that when the user click the *Details* link she is taken to a new *Details* view displaying the clicked product details like the following one :



Step-by-step solution at the end of  
this book

## 8.4 HTTP Post parameters

Providing action parameters through the URLs is adapted for short parameters that can appear to the user. In case you want to pass longer action parameters or hide them from the URL, you can pass them as form data. It's almost as simple.

The view should declare a form in order to have the parameters sent :

```
<form method="post">  
  First Name: <input name="firstname" type="text" />  
  <input type="submit" />  
</form>
```

Of course we're not limited to the POST verb. You can use other HTTP verbs when you see fit.

In order to get the parameter, all that our view needs to do is declare a parameter with the same name :

```
public class ProductsController : Controller
{
    [HttpPost]
    public ActionResult Action1(string firstname)
    { ... }
}
```

Note the `[HttpPost]` attribute attached to the action method. By default, an action only reacts to HTTP GET requests, that's why we need it for a HTTP POST.

Now, if you don't want ASP.NET MVC to generate part of your form, you can use an HTML helper. Our view above would become

```
@using (Html.BeginForm())
{
    <input name="firstname" type="text" />
    <input type="submit" />
}
```

The same would go for the HTML input field and submit.

## 8.5 Passing a full blown object

ASP.NET MVC can make your job easy when you need to update full objects. Let's suppose you created a `Person` class :

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

You can write an action that takes a `Person` object as a parameter

```
public class PersonController : Controller
{
    [HttpPost]
    public ActionResult CreatePerson(Person p)
    { ... }
}
```

All you need to do is to make sure that the incoming HTTP request contains parameters whose names match the properties declared in the `Person` class. ASP.NET will create a `Person` instance and pass it to your action.

For instance, we could write the following matching view :

```
@using (Html.BeginForm("CreatePerson", "Person"))
{
    <input name="firstname" type="text" />
    <input name="lastname" type="text" />
    <input type="submit" />
}
```

## 8.6 Sit and watch - Basic product calculator

Let's begin with a simple example. I want to create a page that allows users to compute the product of two numbers. Easy. I'll first write a ViewModel class :

```
public class ComputeProductViewModel
{
    public ComputeProductViewModel(decimal? number1, \
decimal? number2)
    {
        Number1 = number1 ?? 0;
        Number2 = number2 ?? 0;
        Result = Number1 * Number2;
    }

    public decimal Number1 { get; private set; }
    public decimal Number2 { get; private set; }
    public decimal Result { get; set; }
}
```

As seen earlier, a ViewModel can have properties for everything that is input by and output to the user.

For such a simple example, a ViewModel is simply overkill and we could easily do without. Nonetheless I encourage you to work with ViewModels in simple cases, so that it becomes a straightforward way to

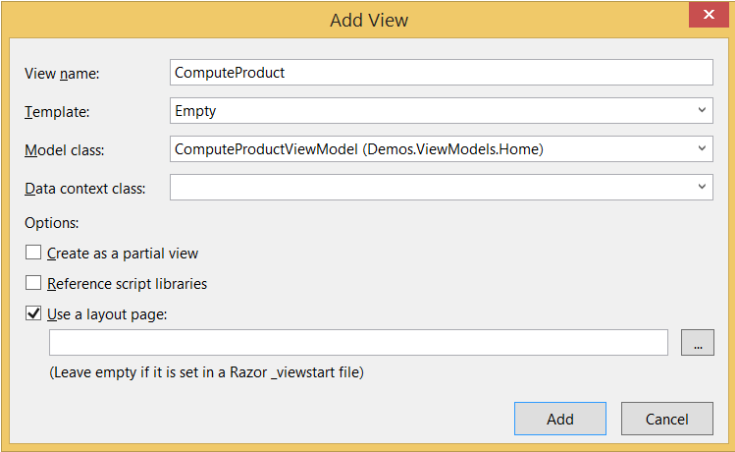


do for you. When more complex views come in your way, you'll simply apply the same techniques and go on seamlessly.

I'll then add a `ComputeProduct` action method to my `HomeController`. Note that it takes two parameters: the numbers to multiply :

```
public class HomeController : Controller
{
    public ActionResult ComputeProduct(decimal? number1, decimal? number2)
    {
        var viewModel = new ComputeProductViewModel(number1, number2);
        return View(viewModel);
    }
}
```

Now I'll right-click the `ComputeProduct` method and select `Add View...` from the contextual menu, then select an *Empty* template and my `ViewModel` as a model for my view :



View name: ComputeProduct

Template: Empty

Model class: ComputeProductViewModel (Demos.ViewModels.Home)

Data context class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

(Leave empty if it is set in a Razor \_viewstart file)

Add Cancel

As my view code I'll type :

```
@model Demos.ViewModels.Home.ComputeProductViewModel
```

```
<h2>Product of two numbers</h2>
```

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

```
{
```

```
    @Html.TextBoxFor(m => m.Number1)
```

```
    @Html.TextBoxFor(m => m.Number2)
```

```
    <input type="submit" value="Compute" />
```

```
}
```

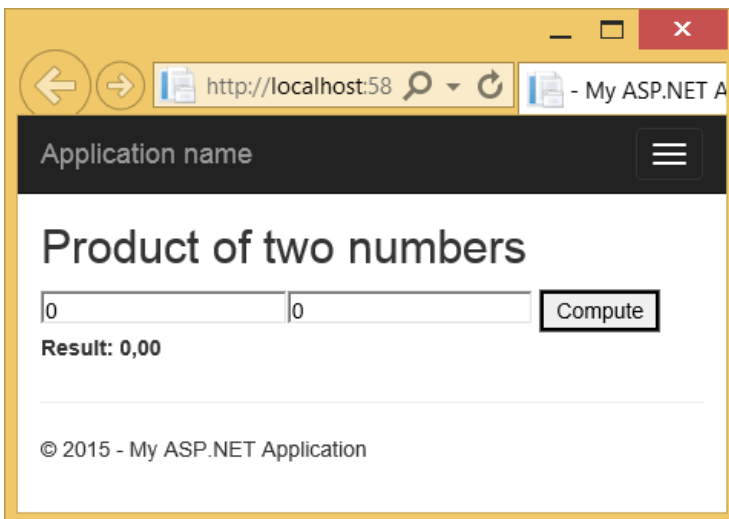
```
<label>Result: @Html.DisplayFor(m=>m.Result)</label>
```

Not much new here, except for the use of two HTML helpers

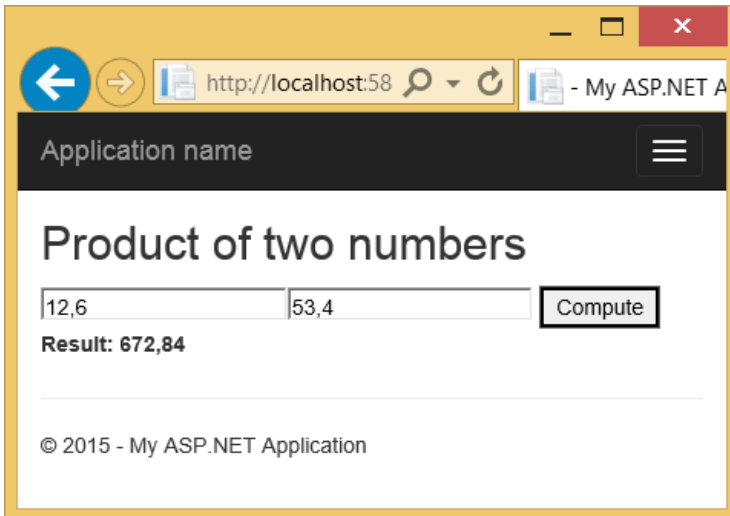
- `Html.TextBoxFor` generates an input field for a given model member
- `Html.Display` generates a display element for a given model member

Both come in very handy since they take a lambda expression that provides your model and expects the member for which to generate the HTML element. Since we have a typed model thanks to the `@model` directive, it's plain easy to type thanks to IntelliSense and C# type-inference. Try it in your own Visual Studio and you'll be surprised how streamlined the experience is.

Let's run our view (Debug / Start Without Debugging menu). We get the following result in our browser :



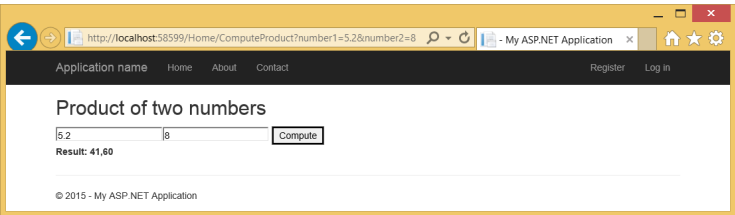
Of course, it works. When we input some numbers and click the `Compute` button, we get our result :



It works. No surprise. Just one more thing: remember we can provide action parameters using URL query parameters? It works here too. If we call our calculator view using the following URL :

```
http://localhost:58599/Home/ComputeProduct?number1=5.\2&number2=8
```

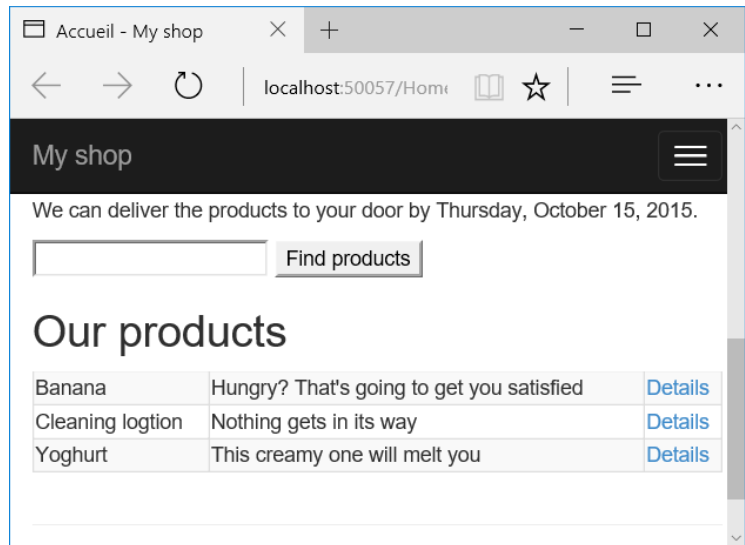
...we get the following result without event clicking the `Compute` button (notice the URL) :



## 8.7 Do-it-yourself 9 - Add a search box to the products list



You must modify the `Home/Index` view: add a textbox input and a button that allows the user to search products typing a part of their name. Search results must appear in the existing list, still only the top 10 ordered by name. The result looks like :

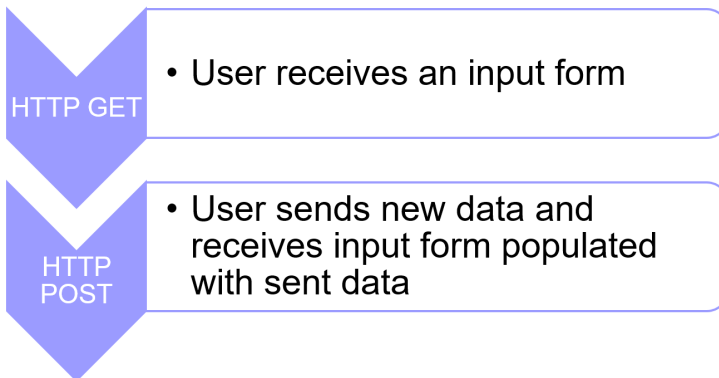


Step-by-step solution at the end of this book

# 9. Updating data scenario

## 9.1 Steps

Now we need to support data input in a real-world scenario, not just sending some partial data. When you input data in a Web application, you do so through an HTML form. In fact, that process can be broken in two steps, whatever the server-side technology



## 9.2 Controller

Those two steps mean rendering a view twice, once for HTTP GET and once for HTTP POST. Which in turn means

two actions on our controller. Your controller would typically look like :

```
public class ProductsController : Controller
{
    public ActionResult Edit(int id)
    {
        ... // fetch data from data source
    }

    [HttpPost]
    public ActionResult Edit(int id, Product p)
    {
        ... // update the data source
    }
}
```

When the data source update is successful, you probably don't want the user to remain on the data input form. Which means the `POST` action would most likely contain a redirect statement. There is a `RedirectToAction` method for this on the base `Controller` class. It comes in handy since we provide the name of the action we want to redirect to :



[HttpPost]

```
public ActionResult Edit(int id, Product p)
{
    // ...
    if(successful) {
        return RedirectToAction("Index");
    }
}
```

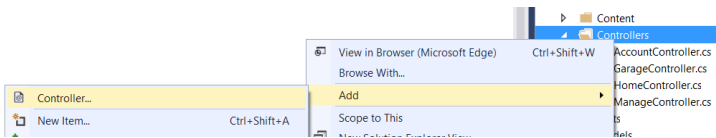
## 9.3 Automated generation of controller and views

Now is time for me to show you a gem. In case we use Entity Framework as a data access layer, Visual Studio can generate in a breeze the controller, actions and views needed for a full CRUD scenario. What's more, the generated code isn't scary and can be modified to fit your needs.

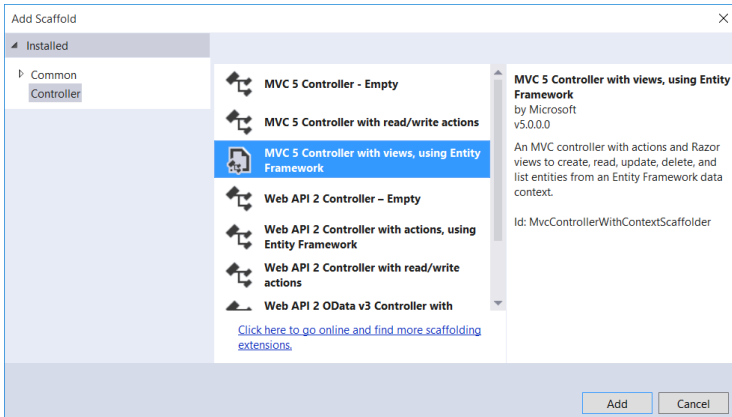
Let's see how it works. Remember we have the following model that we wrote a while ago :

```
public class Car
{
    public int ID { get; set; }
    public string Model { get; set; }
    public double MaxSpeed { get; set; }
}
public class GarageFactory : DbContext
{
    public DbSet<Car> Cars { get; set; }
    // ...
}
```

All I have to do is right-click the **Controllers** directory in Solution Explorer and select **Add / Controller...** from the contextual menu :



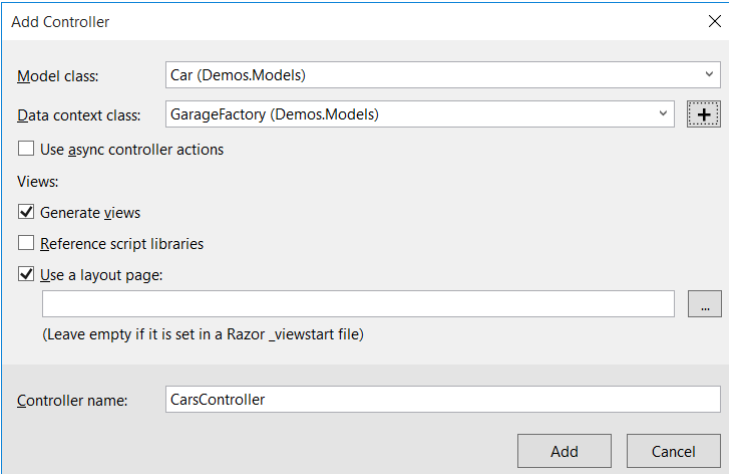
In the **Add Scaffold** dialog box I select “**MVC 5 Controller with views, using Entity Framework**” and click the **Add** button :



That gets me another dialog box, Add Controller, which I'll fill in the following way :

- Model class: Car
- Data context class: GarageFactory
- Generate views: checked (default value)

I'll leave the rest of the options to their default values. Here's the dialog box before I press the Add button :



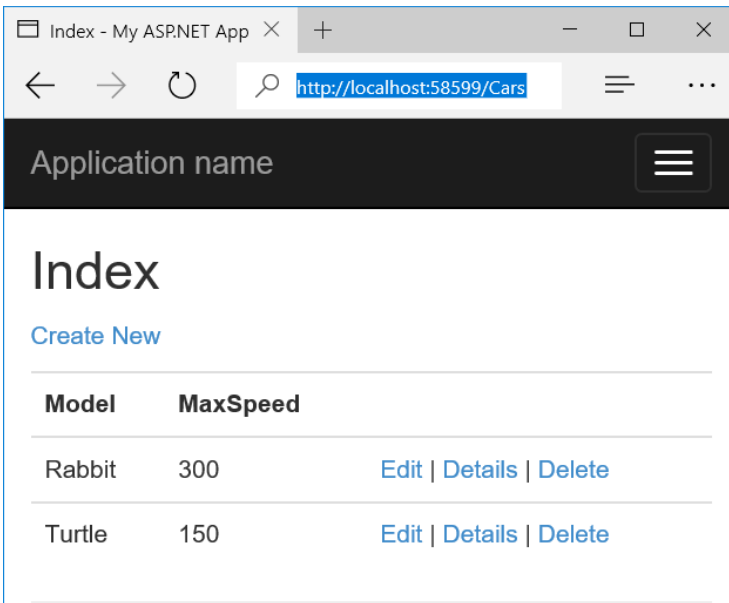
The screenshot shows the 'Add Controller' dialog box with the following configuration:

- Model class:** Car (Demos.Models)
- Data context class:** GarageFactory (Demos.Models)
- ☐ Use async controller actions
- Views:**
  - ☒ Generate views
  - ☐ Reference script libraries
  - ☒ Use a layout page: [Empty text box]
- Controller name:** CarsController

Buttons: Add, Cancel

That's all there is to it! I now have full actions and views for each CRUD operation.

Let's go for instance to the list of cars. We type the `http://localhost:58599/Cars` URL in our browser and get a nice list :



What's more, there are links to create, edit, delete and details views, all working. Time to get the paycheck and go surfing for the next week.

Think you cannot use this wizard in real-life? Think again: if you look at the generated actions and views, they are concise and can be easily modified. Nothing is hidden from you. This is a real time-saver simply put.

Well, I can feel that your fingers are getting itchy again.

You want to try this by yourself? Great, that's just what we're about to do!

## 9.4 Do-it-yourself 10 - Create the products management back-office



You must now cater for the shop products management by the shopkeepers. You are to add the necessary pages. For now, they are open for public access, but don't worry: we'll change that later. Anyway, our application isn't published yet.

Add pages that allow to :

- list all of the products from the database (and add a link to that page in the site top menu);
- add a product to the database;
- edit a product in the database;
- delete a product from a database.

[Step-by-step solution at the end of this book](#)

## 10. Doing more with controllers and actions



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

# 11. Basic security



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>



## 12. Going further



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

# Do-it-yourself Cheat Sheet

This section contains the step-by-step solutions to the do-it-yourself exercises. You normally shouldn't need it if you follow the book in order. But I know you have only a week to learn ASP.NET MVC and I don't want you to remain stuck in any exercise. So here are the steps.



This is just a sample of the full book.

Get this chapter buying the full book here:

<https://leanpub.com/aspnetmvc>

# Definitions



This is just a sample of the full book.

If you like it, get your full version here: <https://leanpub.com/aspnetmvc>