ASP.NET Core MVC

Introduzione alle applicazioni web MVC su ASP.NET Core

Ambiente: .doNET Core + ASP.NET Core 1.0

Anno 2016/2017

Indice generale

1 Introduzione

Il tutorial introduce gli elementi essenziali delle applicazioni web realizzate mediante il framework ASP.NET Core e che utilizzano il *design pattern* **M**odel **V**iew **C**ontroller.

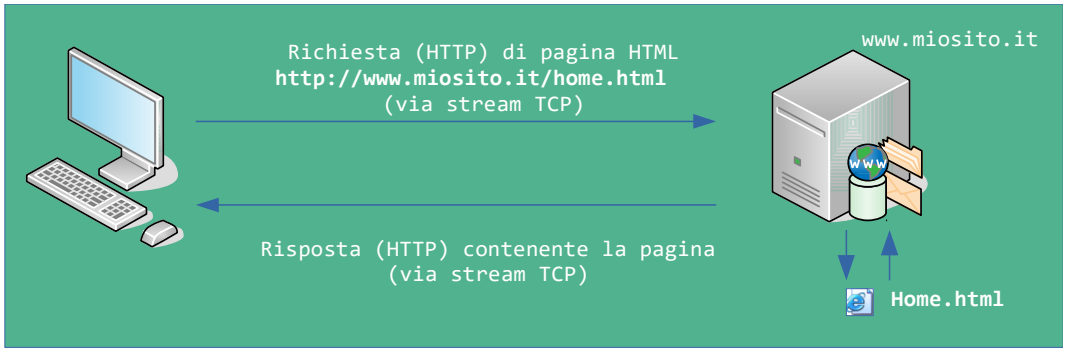
1.1 ASP.NET Core

La sigla identifica l'ultima versione della tecnologia Microsoft, *open source* e *cross-platform*, per la realizzazione di applicazioni web. ASP.NET sta per **A**ctive **S**erver **P**ages su **.NET** Framework. Il termine ***active server page*** (o *pagina server*) si riferisce a pagine web che contengono codice HTML ed esecutivo, sia esso in linguaggio PHP, VB, C#, etc.

Di seguito riepilogo a grandi linee la funzione svolta da questo tipo di pagine e la loro differenza con le normali pagine HTML.

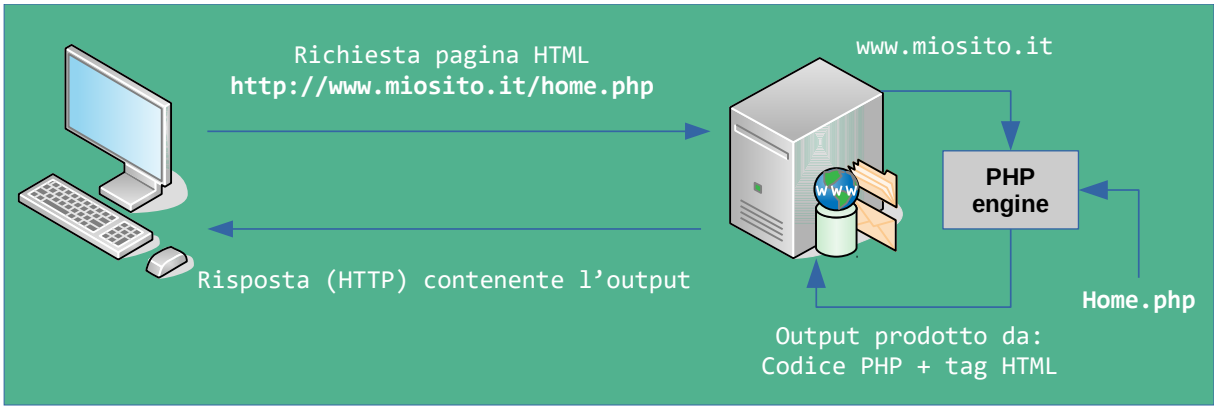
1.1.1 Accesso a una pagina HTML

L'accesso a una normale pagina web implica la richiesta di un file HTML a un *server web*. Questo carica il contenuto del file e lo trasmette al client. Richiesta e risposta utilizzano il protocollo HTTP e dunque "viaggiano" su una connessione TCP.



1.1.2 Accesso a una "pagina server"

Una pagina server viene gestita diversamente. Supponiamo che venga richiesta una pagina PHP:

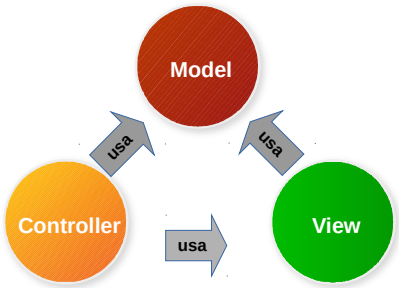


La pagina viene processata da un modulo software in grado di eseguire il codice PHP. Il risultato finale viene inviato al server, che lo ritrasmette al client (il quale riceve dunque del codice HTML).

Ebbene, alla base della tecnologia ASP.NET, in tutte le sue versioni, sottostà lo stesso processo.

1.2 MVC

Il *design pattern* **M**odel **V**iew **C**ontroller stabilisce la separazione dell'applicazione in tre tipi di componenti; il pattern trova il suo naturale impiego nelle applicazioni web.



MVC viene incontro all'esigenza di separare la logica applicativa (*business logic*) da quella di presentazione (interfaccia utente). In questo schema:

1. Nel **Model** è implementato ciò che è relativo ai requisiti dell'applicazione. Ad esempio, relativamente all'applicazione "Library", vi rientrano le classi entità: `Book`, `Author`, `Genre`, etc, le classi *repository* e qualsiasi classe e modulo utilizzato per rispondere ai requisiti (ad esempio, per implementare il prestito di un libro).
2. Della categoria **View** fanno parte i componenti dell'interfaccia utente; questi contengono il codice necessario per presentare i dati. In un'applicazione web tali componenti sono le pagine server.
3. I **Controller** fungono da collante tra View e Model. Definiscono i metodi che rispondono alle richieste dell'utente, i quali ottengono i dati necessari dal Model e caricano le *view* appropriate per presentarli all'utente.

La figura in alto puntualizza che il Model è indipendente da View e Controller: il Model dovrebbe poter essere utilizzato, senza modifiche, in un'applicazione con diversa architettura e/o interfaccia utente: desktop, mobile, web.

Il View dipende ovviamente dal Model, poiché ha la funzione di visualizzarlo, ma è "disaccoppiato" dal Controller, e dunque non ha alcun riferimento ad esso.

Infine, il Controller dipende sia da View che da Model, poiché usa il secondo per soddisfare i requisiti dell'applicazione e chiama il primo per presentare i risultati.

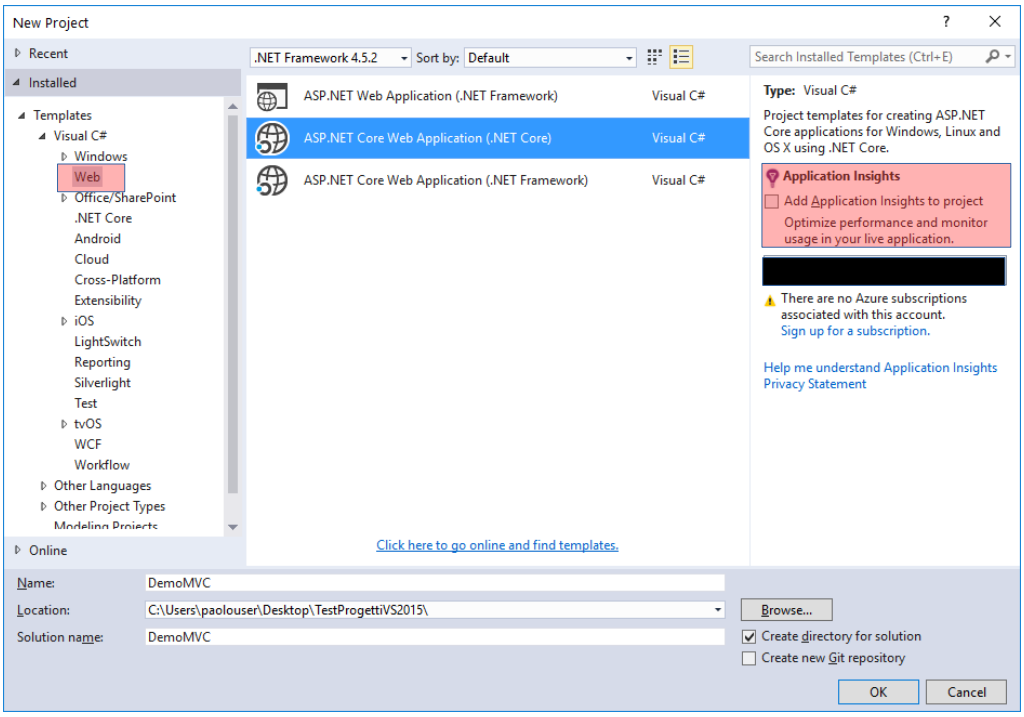
1.3 ASP.NET Core e MVC

MVC è soltanto un *design pattern*; perché sia utilizzabile nella pratica occorre un'infrastruttura che lo implementi, consentendo al programmatore di concentrarsi sui singoli componenti. È ciò che fa ASP.NET Core.

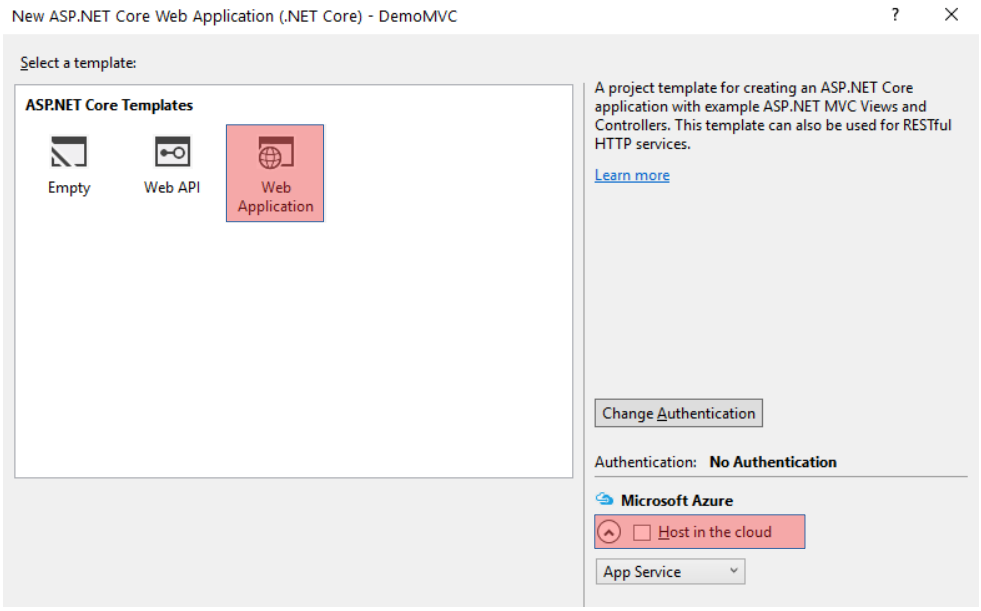
2 Architettura dei progetti ASP.NET Core MVC

2.1 Creazione di un progetto

Si crea un progetto ASP.NET Core MVC selezionando il **web template** e quindi la versione **.NET core** del progetto. Per ridurre le dimensioni del progetto e aumentare le performance conviene disabilitare la voce **Application Insights**:

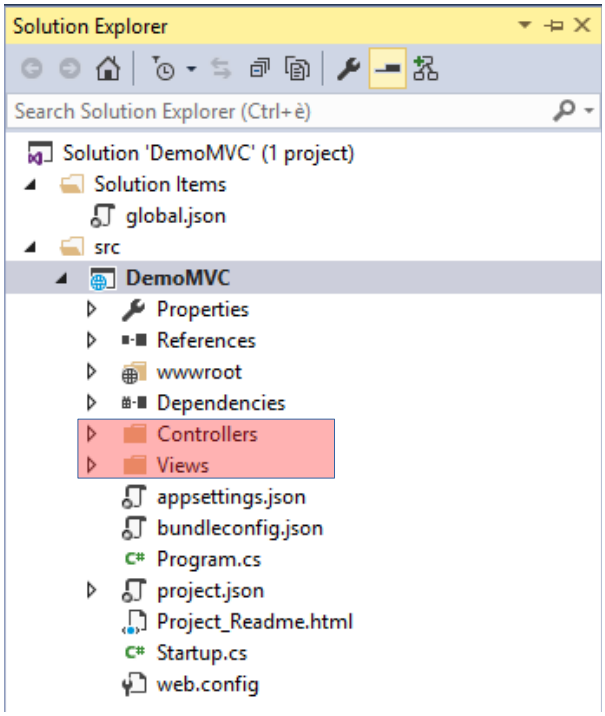


Nella schermata successiva occorre selezionare **Web Application**; ciò fa sì che venga creato un progetto dotato degli elementi principali e già funzionante. È opportuno disabilitare la voce **Host in the cloud**.

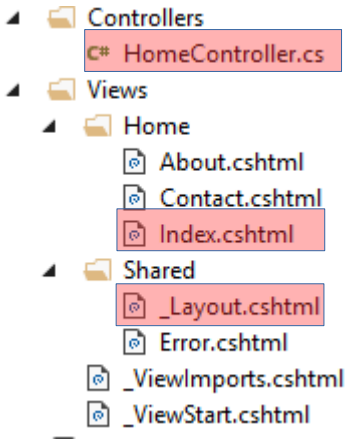


2.2 Struttura generale del progetto

Una volta creato, in *Solution Explorer* appare la seguente schermata:



Oltre a contenere tutti gli elementi necessari al suo funzionamento, il progetto definisce già un prototipo di applicazione, comprensivo di *view*, *controller*, librerie javascript, fogli di stile e immagini. Per il momento, la maggior parte dei file può essere ignorata; ciò che conta si trova nelle cartelle *Controllers* e *Views:*

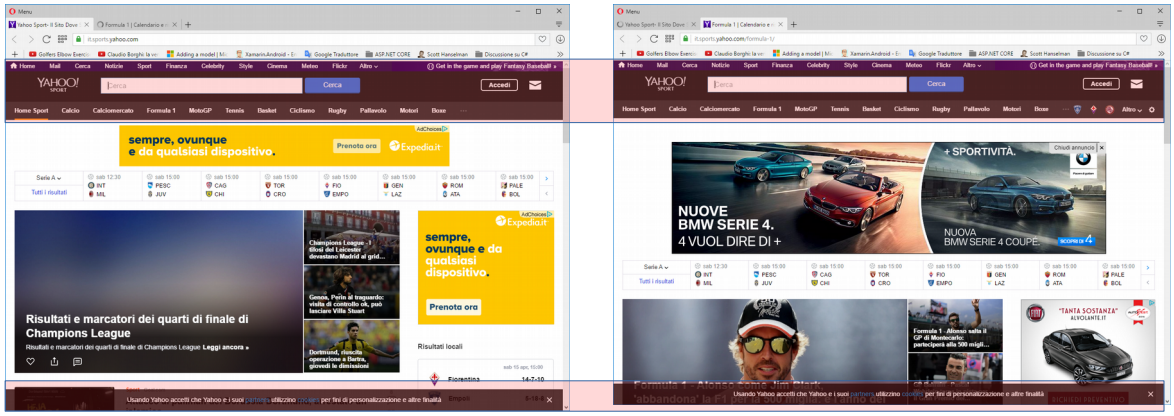


Di seguito mi focalizzerò sui file **HomeController**, **Index** e **_Layout**, i quali rappresentano l'ossatura di una applicazione MVC.

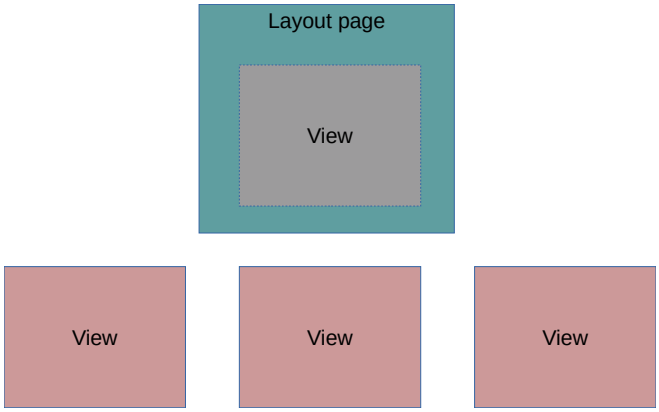
2.3 View, layout page e pagine web

Una *view* è una pagina web con estensione **.cshtml** che contiene codice HTML e C#. Diversamente dalle normali pagine HTML, le *view* non vengono referenziate direttamente, attraverso un *link* o un URL digitato dall'utente, ma sono caricare da codice. Esiste inoltre un'altra differenza: di norma,

le *view* non definiscono l'intera struttura della pagina HTML, ma soltanto i contenuti specifici che devono presentare. Il perché va ricercato nel fatto che, nella maggior parte dei siti web, le pagine condividono delle aree comuni (banner, menù, footer, etc), come mostrato nello *screen shot* di due pagine di *yahoo.it*:



Per questo motivo, ASP.NET adotta un meccanismo che consente alle *view* di essere incorporate nella stessa pagina web, chiamata *layout page*; questa definirà i contenuti comuni e sarà dunque riutilizzata per tutte le *view*, evitando di dover specificare ripetutamente lo stesso codice HTML.



Di seguito mostro la *view* **Index**, che rappresenta la home page , e **_Layout**, che rappresenta la *layout page* e dunque definisce la struttura comune alle pagine del sito. (Ho ridotto entrambe all'essenziale, eliminando quasi tutto il codice originale):

Index.cshtml

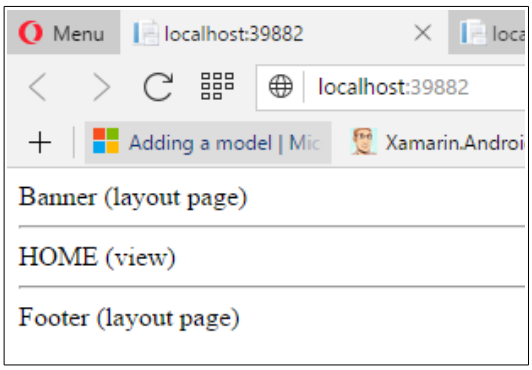
```
@{Layout = "_Layout";}

<div>
    HOME (view)
</div>
```

_Layout.cshtml

```
<!DOCTYPE html>
<html>
<body>
    Banner (layout page)
    <hr />
    @RenderBody()
    <hr />
    Footer (layout page)
</body>
</html>
```

La chiamata al metodo `RenderBody()` identifica la collocazione della *view* all'interno della *layout page*. Eseguendo l'applicazione, poiché **Index** viene caricata automaticamente, si ottiene il seguente risultato, che mostra l'integrazione delle due pagine:

Output del browser



Pagina HTML inviata al browser

```
<!DOCTYPE html>
<html>
<body>
    Banner (layout page)
    <hr />
    <div>
        HOME (view)
    </div>

    <hr />
    Footer (layout page)
</body>
</html>
```

2.3.1 Usare automaticamente la layout page: _ViewStart.cshtml

In realtà non è necessario, nelle *view*, dichiarare la *layout page*, poiché questa è già dichiarata nel file **_ViewStart.cshtml**. Quest'ultimo consente di definire il codice da eseguire al caricamento di ogni *view*. Nella versione predefinita, contiene soltanto l'istruzione:

```
@{Layout = "_Layout";}
```

Questa viene importata automaticamente in tutte le *view*, che saranno dunque integrate nella *layout page* senza il bisogno di dichiararla esplicitamente.

2.3.2 Impostare il titolo della pagina nelle view: dizionario ViewData

Il titolo delle pagine web viene visualizzato sulla barra del titolo del browser e, nel codice HTML, viene impostato mediante il tag **title** contenuto nella sezione **head**. Ciò pone un problema: la pagina visualizzata è rappresentata dalla *view*, ma la sezione **head** è specificata nella *layout page*, che è la stessa per tutte le *view*. La soluzione è impostare il titolo nella *view*, memorizzandolo nel dizionario `ViewData`. La *layout page* utilizzerà il valore nel tag **title** della pagina:

Index.cshtml

```
@{ViewData["Title"] = "Home";}

<div>
    HOME (view)
</div>
```

_Layout.cshtml

```
<!DOCTYPE html>
<html>
    <head>
        <title>@ViewData["Title"]</title>
    </head>
<body>
    ...
</body>
</html>
```

`ViewData` è utile per scambiare dati tra i componenti dell'applicazione e, come vedremo, può essere impiegato anche per passare dati dai *controller* alle *view*.

2.4 Controller e View

Un *controller* è una classe che deriva dal tipo `Controller` e ha la funzione di stabilire quale *view* caricare in risposta alle richieste dell'utente. I metodi della classe sono convenzionalmente chiamati *action*: a ogni metodo corrisponde una *view*.

Segue l'`HomeController` (ho eliminato il codice non essenziale):

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();  // il metodo View() carica la view di nome "Index"
    }
    ...
}
```

Nota bene, il metodo `Index()`:

- Ha lo stesso nome della *view* **Index**.
- Chiama il metodo `View()`: è questo a stabilire la *view* da caricare. Per default carica la *view* con lo stesso nome dell'*action* (**Index**, appunto).

2.4.1 Percorso di ricerca delle view

Per trovare la *view* da caricare, ASP.NET non si limita al nome del metodo *action*, ma utilizza anche il nome del *controller*. Infatti, le view sono memorizzate nella cartella **"Views/<*controller*>"**. Ciò consente di definire lo stesso metodo *action* in più *controller*; ASP.NET sarà comunque in grado di caricare la *view* corretta.

2.4.2 Tipo restituito dai metodi action

Un metodo *action* può restituire qualsiasi valore, ma per caricare una *view* occorre che restituisca un oggetto di tipo `ViewResult`. È ciò che fa `View()`; infatti, il codice precedente potrebbe essere riscritto nel seguente modo:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return new ViewResult() { ViewName = "Index" };
        return View();
    }
    ...
}
```

(`ViewResult` implementa l'interfaccia `IactionResult`.)

2.4.3 Passare dati alla view: uso di ViewData

Le *view* hanno la funzione di visualizzare i dati forniti dai *controller;* a questo scopo può essere usato il dizionario `ViewData`. Nel seguente esempio, il metodo `Index()` memorizza in `ViewData` un messaggio di benvenuto, che sarà visualizzato nella *view*:

HomeController

```
public IActionResult Index()
{
    ViewData["messaggio"] = "Hello, World";
    return View();
}
```

Index

```
@{ViewData["Title"] = "Home";}
<div>
    HOME (view)
    <p><b>@ViewData["messaggio"]</b></p>
</div>
```

2.4.4 Passare dati (strong typed) alla view: uso di @model

In molti scenari occorre gestire oggetti complessi ed è opportuno, nella *view*, poter usufruire dell'Intellisense. A questo scopo esiste la direttiva `@model`, che consente definire il tipo di oggetto da visualizzare nella *view*

Vedremo in azione questa modalità più avanti.

2.5 Eseguire i metodi *action* dei controller: routing

L'ultimo tassello del pattern MVC riguarda il collegamento tra le azioni dell'utente e l'esecuzione dei metodi *action*; si parla in questo caso di *routing* (istradamento).

Il meccanimo di *routing* stabilisce il formato (*route*) degli indirizzi URL gestiti dall'applicazione e li "mappa" con i metodi *action* dei *controller*. Per ogni richiesta (l'utente digita un URL, clicca su un *hyperlink*, invia un form HTML, etc), ASP.NET analizza l'URL, verifica la sua corrispondenza con un *controller* e un metodo *action* e lo esegue.

Nella classe **Startup.cs** (5.1), ASP.NET configura automaticamente la *route* predefinita:

`/[<controller>=Home]/[<action>=Index]/[<parametri>]`

e cioè:

- l'URL viene suddiviso in tre campi (separati da `/`), tutti e tre opzionali.
- Il primo campo specifica il *controller*; se omesso, viene utilizzato l'`HomeController`.
- Il secondo campo specifica il metodo *action*; se omesso viene utilizzato `Index()`.
- Segue, opzionalmente, il parametro o i parametri. Questi ultimi possono essere specificati anche come *querystring*:

```
/[<controller>=Home]/[<action>=Index][?<nome>=<valore>&...]
```

Seguono alcuni URL e i relativi metodi *action* che vengono eseguiti:

URL	Metdo action
/home/index	`Index()` di `HomeController`
/	`Index()` di `HomeController`
/home/GetUser/12	`GetUser(int id)` di `HomeController`
/Catalog/Index	`Index()` di `CatalogController`
/Catalog/GetBooks?genre="Fantascienza"&auth=1	`GetBooks(string genre, int auth)` …

A titolo di esempio modifico la *view* **Index**, collocandovi un *hyperlink* che richiama la *view* **About**:

Index

```
@{ViewData["Title"] = "Home";}
<div>
   HOME (view)
   <p><a href="/home/about">About</a></p>
</div>
```

HomeController

```
public class HomeController:Controller
{
    ...
    public IActionResult About()
    {
        return View();
    }
```

Lo stesso risultato si otterrebbe digitando l'URL (ovviamente comprensivo del sito o, nell'esempio: "http://localhost:5000/home/about").

Nota bene: ASP.NET Core non fa differenza tra lettere minuscole e maiuscole quando mappa l'URL al metodo *action* da eseguire.

2.5.1 Uso di Tag Helper

I *tag helper* consentono di decorare i tag HTML con degli attributi *server-side*. Quest'ultimi non vengono inviati al client, ma elaborati da ASP.NET; hanno la funzione di semplificare la definizione del codice HTML.

Ad esempio, per definire un *hyperlink* è possibile definire separatamente *controller* e *action*:

```
@{ViewData["Title"] = "Home";}
<div>
    HOME (view)
    <p>
        <a href="/home/about">About</a>
        <a asp-controller="home" asp-action="about">About</a>
    </p>
</div>
```

Sarà ASP.NET a elaborare i due tag e produrre un *hyperlink* che rispetta la sintassi HTML.

2.6 Importare namespace e Tag Helpers nelle view: _ViewImports

Le *view* vengono tradotte in classi e compilate; come tali devono definire i *namespace* necessari, come accade in qualsiasi tipo di progetto. Per evitare di dover dichiarare i *namespace* in ogni *view*, è possibile farlo una volta per tutte nel file **_ViewImports.cshtml**:

```
@using DemoMVC
```

Nello stesso file è collocata anche la direttiva:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

che rende disponibili i *tag helpers* in tutte le *view*.

2.7 Razor

Razor identifica la tecnologia che consente di utilizzare C# nelle *view*. Al link di seguito si trova un tutorial che ne spiega le basi:

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor

2.8 Riepilogo

Segue un riepilogo del processo eseguito durante una richiesta del client. Supponiamo che l'utente clicchi sul link **About** della home page:

1. Il browser crea una richiesta HTTP contenente l'URL cliccato: **"/Home/About"**.
2. Il server web passa la richiesta al modulo ASP.NET, che analizza l'URL e, utilizzando la *route* predefinita, verifica la sua corrispondenza con la classe `HomeController` e il metodo `About()`.
3. Crea un oggetto della suddetta classe e ne esegue il metodo.
4. `About()` esegue il metodo `View()`, il quale cerca una *view* di nome **About** nella cartella **"Views/Home"**.
5. Prima di caricare la *view*, vengono eseguite le istruzioni contenute in **_ViewImports** e **_ViewStart**, le quali dichiarano la *layout page* da utilizzare, importano i *tag helper* e dichiarano i *namespace*.
6. La view **About** viene caricata e integrata nella *layout page* **_Layout**.
7. Codice e *tag helper* presenti in entrambi i file vengono processati. Il risultato è puro HTML (più javascript e CSS, se definiti nella *layout page*).
8. Il codice HTML così ottenuto viene scritto nella risposta HTTP, che viene inviata al server web e quindi al client.

3 Applicazione MVC di esempio: MotoGP

L'applicazione seguente mostra in azione gli elementi di base di ASP.NET e del pattern MVC; sarà utilizzata anche per approfondire alcuni concetti introdotti nei paragrafi precedenti.

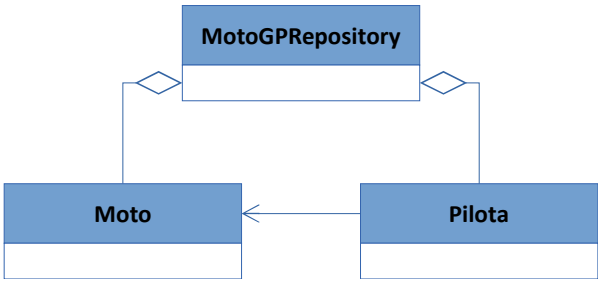
3.1 Descrizione dell'applicazione

L'obiettivo è realizzare un sito web per la gestione del campionato di Moto GP. Il sito dovrà consentire di:

- Visualizzare l'elenco dei piloti e delle moto.
- Visualizzare i piloti che corrono con una determinata moto.
- Visualizzare tutte le informazioni relative a un singolo pilota, foto compresa.
- Inserire un nuovo pilota.

3.1.1 Model

Il Model è rappresentato da tre classi: `Pilota`, `Moto` e `MotoGPRepository`, collocate nella cartella **Model**.




```
public class Pilota
{
    public int PilotaId { get; set; }
    public string Nome { get; set; }
    public string Cognome { get; set; }
    public string Nominativo { get {...}  }
    public int MotoId { get; set; }
    public Moto Moto { get; set; }
    public int Punti { get; set; }
    public int Vittorie { get; set; }
    public string FileFoto { get; set; }
}
```

```
public class Moto
{
    public int MotoId { get; set; }
    public string Nome { get; set; }
}
```

```
public class MotoGPRepository
{
    private static List<Pilota> piloti = new List<Pilota>();
    private static List<Moto> moto = new List<Moto>();
    static int pilotId = 0;
    static int motoId = 0;
    static MotoGPRepository()
```

```
    {
        CreaMoto(new Moto { Nome = "Yamaha" });
        ...

        CreaPilota(new Pilota {...});
        CreaPilota(new Pilota {...});
        ...
    }
}
```

`MotoGPRepository` gestisce le liste di piloti e di moto (soltanto alcuni), simulando l'esistenza di un database. Le due liste sono statiche e dunque "vivono" per l'intera durata dell'applicazione.

3.1.2 *View e controller*

Intendo definire un solo *controller* e le seguenti *view:*

- **Index**: rappresenta la home page e mostra semplicemente una foto.
- **ElencoMoto**: visualizza l'elenco delle marche iscritte al mondiale; consente cliccare su una moto e ottenere l'elenco dei piloti che corrono con essa. (*view* **ElencoPiloti**)
- **ElencoPiloti**: visualizza l'elenco dei piloti; tutti, o soltanto i piloti di una determinata moto. Consente di cliccare sul nominativo di un pilota e ottenere le informazioni disponibili (*view* **InfoPilota**)
- **InfoPilota**: visualizza la informazioni sul singolo pilota, foto compresa.
- **NuovoPilota**: consente l'inserimento di un nuovo pilota.

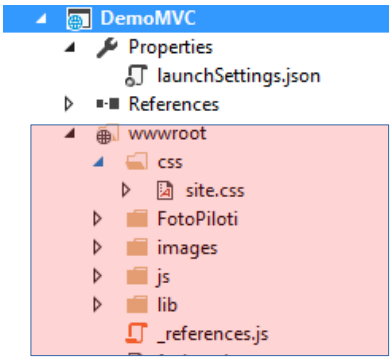
3.2 Layout page

La *layout page* definisce un banner e un menù*:*

```
<!DOCTYPE html>
<html>
    <head>
        <title>@ViewData["Title"]</title>
        <link rel="stylesheet" href="~/css/site.css" />
    </head>
<body>
    <header>
        <img src="~/images/Banner.png"/>
    </header>
    <menu>
        <ul>
            <li><a asp-controller="home" asp-action="Index">Home</a></li>
            <li><a asp-controller="home" asp-action="ElencoMoto">Moto</a></li>
            <li><a asp-controller="Home" asp-action="ElencoPiloti">Piloti</a></li>
        </ul>
    </menu>
    <div>
        @RenderBody()
    </div>
```

```
</body>
</html>
```

La pagina referenzia due elementi statici, il foglio di stile e l'immagine del banner. Entrambi sono memorizzati nella cartella **wwwroot**, all'interno della quale sono presenti anche le foto dei piloti.



3.3 Home page

Sia la *view* **Index** che il metodo `Index()` sono minimali, poiché non devono eseguire alcuna elaborazione:

Index

```
@{ViewData["Title"] = "Home";}

<div class="textcenter">
        <img src="~/images/Homefoto.png"/>
</div>
```

Index()

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

3.4 Elenco moto

L'implementazione di questa funzionalità richiede alcuni approfondimenti:

- Accedere al Model nel *controller* e passarlo alla *view*.
- Accedere ai dati in modalità "strong type" nella *view*.
- Generare un elenco di *hyperlink* il cui attributo **href** dipenda dalla moto visualizzata.

3.4.1 Utilizzare il Model nel controller

Un modo semplice per utilizzare il Model è creare un `MotoGPRepository` nel *controller* e utilizzarlo per ottenere l'elenco delle moto:

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();

    public IActionResult ElencoMoto()
    {
        return View(repo.GetMoto());  // passa l'elenco delle moto alla view
    }
}
```

Nota bene: invece di passare i dati alla *view* mediante il dizionario `ViewData`, ho usato la modalità "*strong type*", passando l'elenco delle moto al metodo `View()`.

3.4.2 Accedere al Model nella view

Nella *view* **ElencoMoto** occorre dichiarare il tipo dei dati mediante la direttiva `@model`, Ciò produce la definizione automatica di una variabile di nome `Model`, che rappresenta l'oggetto ricevuto dal *controller*.

```
@{ViewData["Title"] = "Elenco moto";}

@model IEnumerable<Moto>  //Dichiara il tipo dell'oggetto ricevuto dal controller
...
```

Nota bene: con questa modalità, Visual Studio fornisce l'*Intellisense* e l'analisi del codice durante scrittura.

3.4.3 Generazione degli hyperlink

Per meglio capire cosa generare, mostro il risultato da ottenere:

```
...
<p class="moto"> <a href="/Home/ElencoPilotiMoto/1">Yamaha</a></p>
<p class="moto"> <a href="/Home/ElencoPilotiMoto/2">Honda</a></p>
<p class="moto"> <a href="/Home/ElencoPilotiMoto/3">Ducati</a></p>
<p class="moto"> <a href="/Home/ElencoPilotiMoto/4">Aprilia</a></p>
...
```

Ogni link referenzia l'*action* `ElencoPilotiMoto()` e specifica l'id della moto visualizzata; inoltre, visualizza il nome della moto. Per specificare l'id nel link, si può usare il tag helper **asp-route-...** , al quale sarà assegnato il campo `MotoId` della moto:

```
@{ViewData["Title"] = "Elenco moto";}

@model IEnumerable<Moto>  // Dichiara il tipo dell'oggetto ricevuto dal controller
<div class="textcenter">
    @foreach (var m in Model)  // Model è di tipo IEnumerable<Moto>; "m" è di tipo Moto
    {
        <p class="moto">
           <a asp-controller="Home"
              asp-action="ElencoPilotiMoto" asp-route-id="@m.MotoId">@m.Nome</a>
        </p>
    }
</div>
```

Alternativamente potremmo scrivere:

```
<a href="/Home/ElencoPilotiMoto/@m.MotoId">@m.Nome</a>
```

3.5 Elenco dei piloti

La visualizzazione dell'elenco dei piloti rappresenta un chiaro esempio di come il pattern MVC semplifichi lo sviluppo permettendo di separare la UI dalla elaborazione dati. L'elenco dei piloti può essere ottenuto sia cliccando sulla voce "**Piloti**" del menù, sia sul nome di una moto nella *view* **ElencoMoto**. In entrambi i casi è compito della *view* **ElencoPiloti** visualizzare l'elenco:

```
@{ViewData["Title"] = "Elenco piloti";}

@model IEnumerable<Pilota>
<div>

    <table class="center grid">
        <thead>
            <tr>
                <th>Nominativo</th>
                <th>Moto</th>
                <th>N°</th>
                <th class="textright">Vittorie</th>
                <th class="textright">Punti</th>
            </tr>
        </thead>
        @foreach (var p in Model) //Model è di tipo IEnumerable<Pilota>,
        {                         "p" è di tipo Pilota
            <tr>
                <td><a asp-controller="Home" asp-action="InfoPilota"
                       asp-route-id="@p.PilotaId">@p.Nominativo</a></td>
                <td>@p.Moto.Nome</td>
                <td>@p.Numero</td>
                <td class="textright">@p.Vittorie</td>
                <td class="textright">@p.Punti</td>
            </tr>
        }
    </table>
</div>
```

Nota bene: il nome del pilota viene visualizzato mediante un *hyperlink* che l'utente può cliccare per accedere alla pagina di informazioni sul pilota.

3.5.1 Caricamento dei piloti

In base all'azione dell'utente, viene eseguito uno di due metodi che restituiscono l'elenco dei piloti da visualizzare:

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();

    //Mappa la richiesta: /Home/ElencoPiloti
    public IActionResult ElencoPiloti()
    {
        return View(repo.GetPiloti());
    }
```

```
    //Mappa la richiesta: /Home/ElencoPilotiMoto/<id>  (tag helper asp-route-id)
    public IActionResult ElencoPilotiMoto(int id)
    {
        return View("ElencoPiloti", repo.PilotiMoto(id)); //Specifica la view da caricare
    }
}
```

Interessante è il metodo `ElencoPilotiMoto()`; questo viene chiamato quando l'utente, nella *view* **ElencoMoto**, clicca su uno dei link generati da:

```
<a asp-controller="Home" asp-action="ElencoPilotiMoto"
       asp-route-id="@m.MotoId">@m.Nome</a>
```

Nell'invocare il metodo, ASP.NET esegue il cosiddetto "binding" tra il parametro dell'URL, e cioè `MotoId`, e il parametro del metodo `ElencoPilotiMoto()`.

Il metodo presenta un secondo aspetto interessante: dichiara esplicitamente la *view* da caricare, e cioè la stessa del metodo `ElencoPiloti()`.

Binding tra i parametri della richiesta e i parametri del metodo

Un'analisi del processo di *binding* richiederebbe diverse considerazioni, sulle quali mi limito a sorvolare. Facendo riferimento alla *route* predefinita:

/Controller=Home/Action=Index/Id?

posso dire che la parte finale del *tag helper* **asp-route-...** deve coincidere con il nome del parametro del metodo. Normalmente si usa **Id**.

Usare un singolo metodo con parametro opzionale

In teoria sarebbe stato possibile risolvere il tutto con un singolo metodo:

```
public IActionResult ElencoPiloti(int? id)
{
    if (id == null)
        return View(repo.GetPiloti());
    return View(repo.PilotiMoto(id));
}
```

Ma, in questo scenario particolare, esiste un bug nell'implementazione dei *tag helper* che obbliga a impiegare un *workaround* semplice da implementare, ma meno semplice da comprendere. Pertanto, ho preferito definire due metodi separati per il caricamento della *view* **ElencoPiloti**.

3.6 Informazioni sul pilota

La pagina contenente le informazioni sul pilota viene caricata in risposta al click sul pilota nella *view* **ElencoPiloti**:

```
...
     <td><a asp-controller="Home" asp-action="InfoPilota"
            asp-route-id="@p.PilotaId">@p.Nominativo</a></td>
...
```

Il metodo *action* `InfoPilota()` riceve l'id del pilota e lo passa alla *view* omonima.

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();
    ...
    public IActionResult InfoPilota(int id)
    {
        return View(repo.GetPilota(id));
    }
}
```

La *view* dichiara il tipo `Pilota` come Model e ne visualizza le proprietà:

```
@{ViewData["Title"] = "Pilota";}
@model Pilota
@{
    string statistiche = string.Format("Vittorie: {0} --- Punti: {1}", Model.Vittorie,
                                                                       Model.Punti);
    string moto = string.Format("{0} {1}", Model.Moto.Nome, Model.Numero);
}
<table class="content">

    <tr>
        <th colspan="2"><h1>@Model.Nominativo</h1></th>
    </tr>
    <tr>
        <th colspan="2" class="textcenter">
            <img src="~/FotoPiloti/@Model.FileFoto" />
        </th>
    </tr>
    <tr>
        <td class="textcenter"><h3>@moto</h3></td>
    </tr>
    <tr>
        <td class="textcenter">@statistiche</td>
    </tr>
</table>
```

In questa *view* c'è una novità rispetto alle precedenti: in un blocco C# vengono impostate due variabili stringa, utilizzate successivamente nel codice HTML. Ciò mostra una caratteristica di *razor*: le variabili definite fuori dai metodi sono considerate globali e dunque accessibili ovunque nella pagina.

3.7 Nuovo pilota

La creazione di un nuovo pilota richiede di implementare le seguenti funzionalità:

- Un form HTML per l'inserimento dei dati (Nome, Cognome, etc).
- L'uso di un *combobox* per selezionare la moto guidata dal pilota. Questo deve essere popolato con l'elenco delle moto.
- La possibilità di "uploadare" la foto del pilota.

La funzionalità di inserimento è suddivisa in due parti, corrispondenti ad altrettanti metodi *action*. Il primo metodo carica la *view* contenente il form HTML; il secondo riceve i dati inseriti nel form e e procede all'inserimento. Segue il primo dei due metodi:

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();

    [HttpGet]
    public IActionResult NuovoPilota()
    {
        return View();
    }
}
```

L'attributo `[HttpGet]`, benché non obbligatorio, stabilisce che il metodo viene chiamato per il caricamento del form e non per processare i suoi dati.

3.7.1 Implementazione base del form

Segue la *view* **NuovoPilota**, che, nell'attuale versione, non considera l'input della moto e del file immagine. Nota bene: l'attributo **method** del form specifica che sarà impiegato un **post** per l'invio dei dati.

```
@{ViewData["Title"] = "Nuovo pilota";}
@model Pilota
<div>
    <form asp-controller="Home" asp-action="NuovoPilota" method="post">
        <table class="center">
            <tr>
                <td><label asp-for="Nome"></label></td>
                <td><input asp-for="Nome"/></td>
            </tr>
            <tr>
                <td><label asp-for="Cognome"></label></td>
                <td><input asp-for="Cognome"/></td>
            </tr>
            <tr>
                <td><label asp-for="Numero"></label> </td>
                <td><input asp-for="Numero" /></td>
            </tr>
            <tr>
                <td><label asp-for="Punti"></label> </td>
```

```
                <td><input asp-for="Punti" value="0"/></td>
            </tr>
            <tr>
                <td><label asp-for="Vittorie"></label></td>
                <td><input asp-for="Vittorie" value="0"/></td>
            </tr>
            <tr>
                <td colspan="2" class="textcenter">
                    <input type="submit" value="Crea" />
                </td>
            </tr>
        </table>
    </form>
</div>
```

Ci sono due questioni interessanti, correlate tra loro:

- Come le altre, anche questa *view* dichiara il tipo del `Model`.
- L'uso dei *tag helper* consente di generare automaticamente il tipo appropriato dei tag di **input**, di inserire i dati nelle proprietà specificate e di impostare automaticamente le **label**.

3.7.2 Processare i dati inseriti

Nel form, il *tag helper* **asp-action** specifica il metodo `NuovoPilota()` come il destinatario dei dati inseriti; ovviamente non si tratta dello stesso metodo che carica la *view*:

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota)
{
    repo.NuovoPilota(pilota);
    return RedirectToAction("ElencoPiloti");
}
```

Vi sono tre elementi degni di nota:

- Il metodo è decorato con l'attributo `[HttpPost]`; questo lo identifica come un metodo che riceve i dati da un form.
- Sulla base del model dichiarato nella *view*, ASP.NET è in grado di "bindare" i dati del form nelle corrispondenti proprietà del parametro `pilota`.
- Dopo aver inserito il pilota nel *repository*, il metodo "ridireziona" l'utente alla pagina elenco piloti, non caricando direttamente la *view*, ma utilizzando `RedirectTiAction()`, il quale esegue il metodo *action* corrispondente alla *view* specificata.

(Nota bene: nell'attuale versione non viene presa in considerazione l'eventualità di errori, nei dati come nel processo di inserimento.)

3.7.3 Selezionare la moto da un elenco

Il modo corretto per l'inserimento della moto guidata dal pilota è quello di consentire all'utente di selezionarla da un elenco, implementato mediante un tag **select**. Qui sorge un problema, perché il Model utilizzato nella *view* è di tipo `Pilota`, e la suddetta classe non definisce l'elenco delle moto.

Una soluzione è quella di passare l'elenco alla *view* mediante `ViewData`:

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();

    [HttpGet]
    public IActionResult NuovoPilota()
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        return View();
    }

    [HttpPost]
    public IActionResult NuovoPilota(Pilota pilota)
    {
        ... // resta invariato
    }
}
```

Nella *view* si memorizza innanzitutto l'elenco in una variabile e successivamente si usa il *tag helper* **asp-items** per generare il tag **select**:

```
@{
    ViewData["Title"] = "Nuovo pilota";
    var listaMoto = ViewData["listaMoto"] as IEnumerable<Moto>;
}
@model Pilota
<div>
    <form asp-controller="Home" ...>
        <table class="center">
             ...
            <tr>
                <td><label asp-for="MotoId">Moto</label></td>
                <td>
                    <select asp-for="MotoId"
                            asp-items="@(new SelectList(listaMoto,"MotoId","Nome"))">
                    </select>
                </td>
            </tr>
            ...
         </table>
    </form>
</div>
```

Nota bene, l'uso di **asp-items** permettere di generare dinamicamente il seguente tag:

```
<select id="MotoId" name="MotoId">
    <option value="1">Yamaha</option>
    <option value="2">Honda</option>
    <option value="3">Ducati</option>
    <option value="4">Aprilia</option>
</select>
```

3.7.4 Upload del file con la foto del pilota

Per implementare la funzionalità di upload della foto del pilota occorre:

- Utilizzare un tag **input** di tipo "file".
- Nel metodo `NuovoPilota()` accedere al file caricato.
- Ottenere il percorso della cartella **FotoPiloti**, collocata in **wwwroot** (la cartella radice per risorse statiche del sito), e copiare il file.

Alla *view* **NuovoPilota** occorre aggiungere il tag **input** per la selezione del file; inoltre, perché sia possibile l'upload, occorre aggiungere l'attributo **enctype** al form:

```
@{
    ViewData["Title"] = "Nuovo pilota";
    var listaMoto = ViewData["listaMoto"] as IEnumerable<Moto>;
}
@model Pilota
<div>
    <form asp-controller="Home" enctype="multipart/form-data" ...>
        <table class="center">
             ...
            <tr>
                <td><label asp-for="FileFoto">File foto</label></td>
                <td><input type="file" name="file" /></td>
            </tr>
            ...
         </table>
    </form>
</div>
```

Nota bene: al tag **input** deve essere dato un nome ben definito, poiché dovrà essere lo stesso utilizzato nel secondo parametro nel metodo `NuovoPilota()`:

```
using Microsoft.AspNetCore.Http;   // definisce il tipo IFormFile

public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();

    [HttpGet]
    public IActionResult NuovoPilota()
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        return View();
    }

    [HttpPost]
    public IActionResult NuovoPilota(Pilota p, IFormFile file)
    {
        //... crea pilota e salva file
    }
}
```

Il tipo `IFormFile` memorizza le informazioni relative al file caricato e consente di salvarlo su disco.

Salvare il file su disco

Occorre innanzitutto conoscere il percorso della cartella **wwwroot**. Un modo per farlo è quello di accedere all'*hosting environment*, e cioè l'oggetto che memorizza le informazioni sull'ambiente di esecuzione dell'applicazione. L'oggetto viene creato automaticamente da ASP.NET e viene passato al *controller*, purché dichiari il costruttore appropriato:

```
using Microsoft.AspNetCore.Hosting;  // definisce il tipo IHostingEnvironment

public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();
    IHostingEnvironment hostEnv;

    public HomeController(IHostingEnvironment hostEnv)
    {
        this.hostEnv = hostEnv;
    }
    ...
    [HttpPost]
    public IActionResult NuovoPilota(Pilota pilota, IFormFile file)
    {
        if (file != null) // è stato selezionato un file?
        {
            var ext = Path.GetExtension(file.FileName);
            var filePath = string.Format("{0}/FotoPiloti/{1}{2}.{3}", hostEnv.WebRootPath,
                                         pilota.Cognome, pilota.Nome, ext);

            var fs = new FileStream(filePath, FileMode.Create))
            file.CopyTo(fs);  // salva il file sul filestream e dunque su disco
            fs.Close();

            pilota.FileFoto = Path.GetFileName(filePath);
        }

        pilota.Moto = repo.GetMoto(pilota.MotoId);
        repo.NuovoPilota(pilota);
        return RedirectToAction("ElencoPiloti");
    }
}
```

Il codice del metodo `NuovoPilota()`:

- Verifica se è stato caricato un file. In caso positivo:
  - Ottiene l'estensione del file.
  - Genera il percorso di destinazione, utilizzando la proprietà `WebRootPath` per accedere al percorso di **wwwroot**.
  - Crea un `FileStream` e lo usa per salvare il file.
  - Imposta il nome del file nell'oggetto `pilota`.

- Usando la proprietà `MotoId`, ottiene un *reference* alla moto corrispondente.
- Inserisce il pilota nel *repository*.

3.8 Validare dei dati

Un compito che dovrebbe essere implementato nel *controller* è quello di verificare che i dati siano validi prima di salvarli nel *repository*; in caso contrario, dovrebbe essere riproposto il form di inserimento, insieme a un messaggio di errore. A questo proposito, ASP.NET fornisce un meccanismo di validazione automatica che, utilizzando dei criteri basati su attributi, consente di verificare automaticamente se il *model* inviato dal form è valido.

Innanzitutto occorre decorare il *model*, specificando i vincoli da soddisfare:

```
using System.ComponentModel.DataAnnotations;

public class Pilota
{
    public int PilotaId { get; set; }

    [Required]
    public string Nome { get; set; }

    [Required]
    public string Cognome { get; set; }

    public string Nominativo { get { return Cognome + ", " + Nome; } }
    public int MotoId { get; set; }
    public Moto Moto { get; set; }

    [Range(1, 99)]
    public int Numero { get; set; }

    [Range(0, 450)]
    public int Punti { get; set; }

    [Range(0, 18)]
    public int Vittorie { get; set; }

    public string FileFoto { get; set; }
}
```

Quindi, nel metodo *action* che elabora il form, occorre verificare che il *model* sia valido:

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota, IFormFile file)
{
    if (!ModelState.IsValid)  // se non è valido, carica nuovamente il form
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        return View(pilota);
    }
    ...
```

```
    return RedirectToAction("ElencoPiloti");
}
```

Infine, nella *view* occorre visualizzare gli errori, in modo che l'utente possa correggerli.

```
@{
    ViewData["Title"] = "Nuovo pilota";
    var listaMoto = ViewData["listaMoto"] as IEnumerable<Moto>;
}
@model Pilota
<div>
    <form asp-controller="Home" enctype="multipart/form-data" ...>
        <table class="center">
             ...
            <tr>
                <td><label asp-for="FileFoto">File foto</label></td>
                <td><input type="file" name="file" /></td>
            </tr>
            ...
            <tr>
                <td colspan="2" class="error-text">
                    @foreach (var ms in ViewData.ModelState.Values)
                    {
                        foreach (var e in ms.Errors)
                        {
                            <p>@e.ErrorMessage</p>
                        }
                    }
                </td>
            </tr>
         </table>
    </form>
</div>
```

Il riquadro mostra come accedere allo "stato del modello" nella *view*. Per ognuno dei dati inseriti, vengono visualizzati gli errori (poiché potrebbero essere più di uno). Dopo aver confermato il form, questo viene validato nel *controller* ed eventualmente ricaricato con i relativi messaggi di errore, stabiliti automaticamente da ASP.NET.[1]

3.8.1 Personalizzare i messaggi di errore

Sempre mediante gli attributi è possibile definire i propri messaggi di errore. Di seguito mostro come personalizzare il messaggio relativo al numero della moto:

```
public class Pilota
{
    ...
    [Range(1, 99, ErrorMessage ="Il numero deve essere compreso tra 1 e 99")]
    public int Numero { get; set; }
    ...
}
```

1 Come vedremo in 3.8.2, esiste un approccio migliore.

Segue uno screen shot, dove il form è stato caricato dopo aver violato tre vincoli sui dati:

3.8.2 Usare i tag helper per visualizzare gli errori

ASP.NET core definisce dei *tag helper* per la visualizzazione degli errori. Ne esistono di due tipi:

- `asp-validation-for` è utilizzato per visualizzare errori di validazione dei singoli campi.
- `asp-validation-summary` è utilizzato per visualizzare un riepilogo degli errori e/o dei messaggi personalizzati.

Entrambi i *tag helper*, insieme agli attributi e all'oggetto `ModelState`, consentono un elevato livello di personalizzazione. L'approccio standard, comunque, è quello di utilizzare un tag di validazione accanto a ogni tag di **input**, e un tag generale che riepiloghi gli errori e/o visualizzi messaggi che riguardano la validità del modello in generale.

Di seguito mostro un esempio d'uso di entrambi i tag:

```
@{
    ViewData["Title"] = "Nuovo pilota";
    var listaMoto = ViewData["listaMoto"] as IEnumerable<Moto>;
}
@model Pilota
<div>
    <form asp-controller="Home" enctype="multipart/form-data" ...>
        <table class="center">
            <tr>
                <td colspan="2">
                    <div asp-validation-summary="ModelOnly" class="error-text"></div>
                </td>
            </tr>
             ...
            <tr>
                <td><label asp-for="Numero"></label> </td>
                <td><input asp-for="Numero" />
                    <span asp-validation-for="Numero" class="error-text"></span>
                </td>
```

```
            </tr>
            ...
        </table>
    </form>
</div>
```

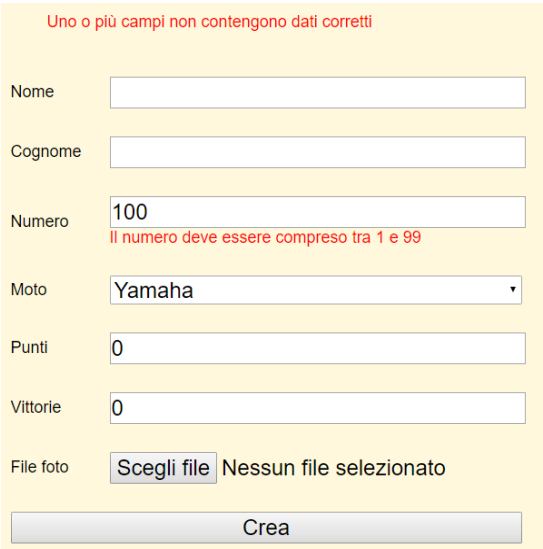
Il **div** posto all'inizio ha la funzione di riepilogo. Il valore **ModelOnly** dell'attributo indica che non saranno visualizzati i singoli errori relativi ai campi. Questi vengono visualizzati attraverso dei tag **span**, posizionati accanto ai tag di **input** corrispondenti. Alternativamente, si può decidere di usare soltanto il **div** di riepilogo, specificando il valore **All** per l'attributo, in modo che vengano visualizzati automaticamente tutti gli errori.

Aggiungere errori al modello

Se impostato al valore **ModelOnly**, il *tag helper* `asp-validation-summary` non produce alcun output, a meno che non siano esplicitamente aggiunti uno o più errori nel *controller*.

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota, IFormFile file)
{
    if (!ModelState.IsValid)  // se non è valido, carica nuovamente il form
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        ModelState.AddModelError("", "Uno o più campi sono contengono dati corretti");
        return View(pilota);
    }
    ...
}
```

Queste modifiche producono il seguente risultato:



3.8.3 Validazione generale del modello

Può accadere che i singoli campi del modello siano validi ma che non lo sia il modello nel suo insieme. È compito del metodo *action* verificare questa eventualità ed eventualmente aggiungere un errore all'oggetto `ModelState`:

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota, IFormFile file)
{
    if (!ModelState.IsValid)
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        ModelState.AddModelError("", "Uno o più campi non contengono dati corretti");
        return View(pilota);
    }

    if (pilota.Vittorie > 0 && pilota.Punti == 0)
    {
        ViewData["listaMoto"] = repo.GetElencoMoto();
        ModelState.AddModelError("", "Valori incoerenti tra 'Punti' e 'Vittorie'");
        return View(pilota);
    }
    ...
}
```

Nota bene: in entrambi i casi, il primo parametro del metodo `AddModelError()` è vuoto; è questo a contraddistinguere un errore generale, che non riguarda uno specifico campo.

3.8.4 Validazione “lato client”

4 Usare Entity Framework in ASP.NET Core MVC

Diversamente dagli altri tipi di applicazione, ASP.NET Core MVC non è compatibile con Entity Framework versione 6; pertanto, occorre usare Entity Framework Core. Questo presenta alcune differenze rispetto a EF 6:

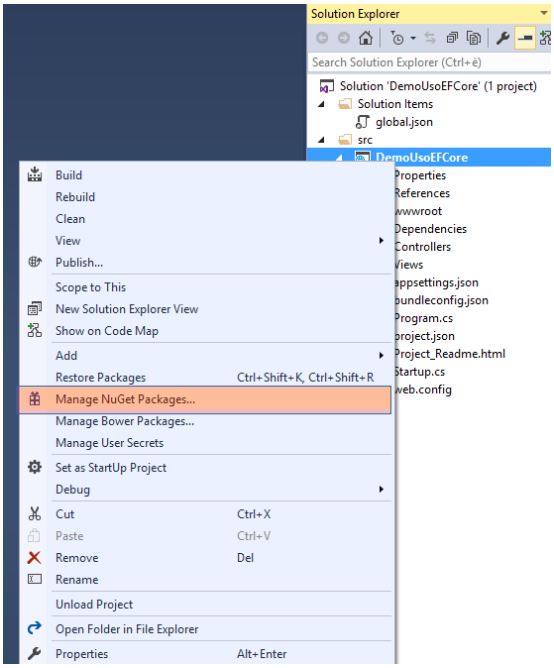
- Non implementa (ancora) alcune funzionalità, come ad esempio il *lazy loading*. (Quindi non ha senso dichiarare virtuali le *reference property* e le *collection property.*
- Implementa un sistema di configurazione dell'*entity model* leggermente diverso.
- Implementa un diverso meccanismo per gestire la stringa di connessione (non è in grado di ottenerla dai file **App.Config** e **Web.Config**).
- Ha un'architettura modulare, basata sul concetto di provider, che consente selezionare il modulo necessario per dialogare con un determinato DBMS.

Detto questo, i concetti appresi su EF 6 restano validi anche per EF Core.

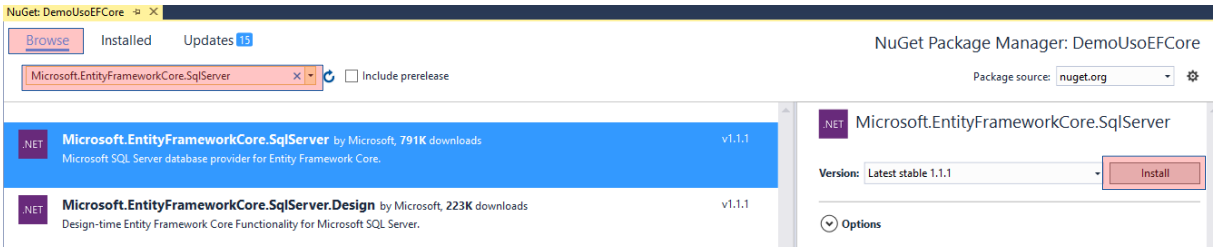
4.1 Aggiungere EF a un progetto ASP.NET Core

Per aggiungere EF a un progetto occorre innanzitutto stabilire il provider che si desidera utilizzare, e dunque il DBMS che ospita il database. Ad esempio, se dobbiamo interfacciarci con SQL Server, occorre aggiungere il pacchetto NuGet: **Microsoft.EntityFrameworkCore.SqlServer**.

Per farlo, aprire il menù contestuale del progetto e selezionare il comando **Menage NuGet Packages**:



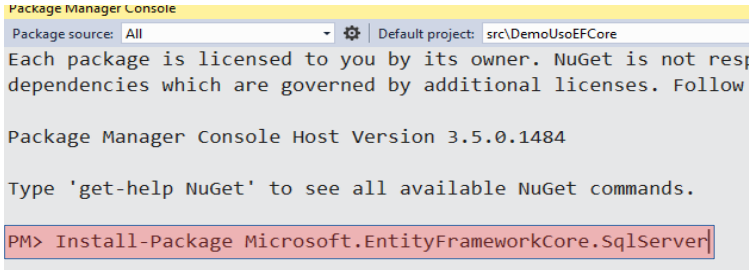
Sulla finestra successiva selezionare la pagina **Browse** e digitare il nome del pacchetto, quindi installarlo.



Alternativamente, aprire la Package Manager Console attraverso il menù:

Tools | Nuget Package Manager | Package Manager Console

e, al prompt, digitare il comando **Install-Package** seguito dal nome del pacchetto:



Dipendenze e compatibilità tra i moduli di ASP.NET Core

ASP.NET Core è un framework modulare in rapida evoluzione; può accadere che la versione di un modulo non sia compatibile con la versione precedente di un altro. È ciò che accade quando si installa EF Core in un progetto ASP.NET Core in Visual Studio 15. In questo caso, il modulo **Microsoft.Extensions.Logging** ha la versione 1.0.0 e non è compatibile con EF Core 1.1.1 (la versione corrente mentre sto scrivendo). (Il file **project.json** contiene l'elenco dei moduli referenziati nel progetto e mostra questo problema mediante un avvertimento.)

Occorre aggiornare il modulo di logging utilizzando la stessa procedure impiegata per installare EF Core. Ad esempio, aprendo il Console Package Manager e digitando:

PM> Install-Package Microsoft.Extensions.Logging

Visual Studio aggiornerà il modulo all'ultima versione disponibile.

4.2 Configurare e utilizzare l'oggetto context

Vi sono due modi per configurare l'oggetto *context* perché che utilizzi un determinato database. In quello più semplice, il *context* definisce internamente la stringa di connessione al database.

Segue la classe `MotoGPContext`; fornisce l'accesso al database **PilotiMotoGP**, il quale memorizza le tabelle **Piloti** e **Moto**.

```
public class MotoGPContext: DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb...");
```

```
    }

    public DbSet<Pilota> Piloti { get; set; }
    public DbSet<Moto> Moto { get; set; }
}
```

Mediante il parametro `optionsBuilder` si può configurare il *context*, ad esempio per stabilire la stringa di connessione invocando il metodo di estensione `UseSqlServer()`.

4.2.1 Configurazione del model

Nel nostro caso, la configurazione del model non richiede niente di particolare:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
...
[Table("Moto")]
public class Moto
{
    public int MotoId { get; set; }
    public string Nome { get; set; }
}

[Table("Piloti")]
public class Pilota
{
    public int PilotaId { get; set; }
    ...
    public byte[] Foto { get; set; }

}
```

È necessaria la mappatura delle *entità* mediante attributi, poiché la convenzione sui nomi le assocerebbe alle tabelle **Motos** e **Pilotis**.

4.2.2 Uso del context

Consideriamo il metodo *action* `ElencoPiloti()`; l'uso di EF ricalca il pattern già conosciuto:

```
public class HomeController : Controller
{
    IHostingEnvironment hostEnv;
    MotoGPRepository repo = new MotoGPRepository();
    MotoGPContext db = new MotoGPContext();
    ...
    public IActionResult ElencoPiloti()
    {
        return View(db.Piloti.Include(p => p.Moto));
        return View(repo.GetPiloti());
    }
    ...
}
```

Due cose degne di nota:

- il *context* viene dichiarato e creato globalmente, così può essere utilizzato in tutti i metodi del *controller*.
- Mediante il metodo `Include()`, per ogni pilota viene inclusa la moto corrispondente. Si tratta della tecnica di *eager loading* (che esiste anche in EF 6), ed è necessaria, poiché in EF Core non si applica il *lazy loading*.[2]

4.3 Definire un ViewModel

La nuova versione di `Pilota` incorpora l'immagine corrispondente, adesso memorizzata nel database e non più in un file separato[3]. Ciò diminuisce notevolmente le prestazioni della *view* **ElencoPiloti**, poiché ogni pilota memorizza anche l'immagine, nonostante la *view* non la visualizzi.

In questo caso non è conveniente passare direttamente il model alla view; è opportuno progettare un nuovo tipo che definisca soltanto le informazioni rilevanti. Si parla in questo caso di *viewmodel*, poiché si tratta di un tipo che definisce dati del *model*, ma è progettato allo scopo di favorire l'implementazione della *view*.

```
public class PilotaInfo
{
    public int PilotaId { get; set; }
    public string Nominativo { get; set; }
    public string NomeMoto { get; set; }
    public int Numero { get; set; }
    public int Punti { get; set; }
    public int Vittorie { get; set; }
}
```

Naturalmente, occorre modificare il *controller*, il quale dovrà restituire un elenco di `PilotiInfo`:

```
// usato dai metodi action ElencoPiloti() e ElencoPilotiMoto()
private IEnumerable<PilotaInfo> GetPilotiInfo(int id = 0) // id->0: tutti i piloti
{
    var piloti = id == 0 ? db.Piloti.Include(p => p.Moto)
        : db.Piloti.Include(p => p.Moto).Where(p => p.MotoId == id);

    return piloti.Select(p => new PilotaInfo
    {
        PilotaId = p.PilotaId,
        Nominativo = p.Nominativo,
        NomeMoto = p.Moto.Nome,
        Numero = p.Numero,
        Vittorie = p.Vittorie,
        Punti = p.Punti
    });
}
```

2 Sorvolo sul fatto che, nell'ambito di applicazioni web, dovrebbe essere usato l'*eager loading* in ogni caso.

3 Qui non discuto sull'opportunità di memorizzare l'immagine nel database e nella stessa **Piloti**. In realtà si tratta di una scelta errata, che diminuisce notevolmente le performance.

```
public IActionResult ElencoPilotiMoto(int id)
{
    return View("ElencoPiloti", GetPilotiInfo(id));
}

public IActionResult ElencoPiloti()
{
    return View(GetPilotiInfo());
}
```

Infine, occorre modificare la *view* **ElencoPiloti**:

```
@{ViewData["Title"] = "Elenco piloti";}

@model IEnumerable<PilotaInfo>
<div>

    <table class="center grid">
        ...
        @foreach (var p in Model)
        {
            ...
                <td>@p.NomeMoto</td>
            ...
        }
    </table>
</div>
```

4.4 Visualizzare le immagini memorizzate nel database

La *view* **InfoPilota** visualizza la foto del pilota mediante un tag **img** che referenzia il file contenente l'immagine, collocato nella cartella **FotoPiloti**; il nome del file è memorizzato nella proprietà `FileFoto` di `Pilota`. Con l'uso di SQL Server, le immagini sono memorizzate nel database e accessibili mediante la proprietà `Foto`: occorre adottare una tecnica diversa perché il browser possa visualizzarle.

Vi sono due possibilità, la prima delle quali, molto semplice, sfrutta una caratteristica del tag **img**: visualizzare un'immagine "incorporata" nella pagina e codificata in base64.

Dunque, è sufficiente modificare il tag **img** della *view* **InfoPilota**:

```
@{ViewData["Title"] = "Pilota";}
...
<table class="content">

    <tr>
        <th colspan="2" class="textcenter"><h1>@Model.Nominativo</h1></th>
    </tr>
    <tr>
        <th colspan="2" class="textcenter">
            <img src="~/FotoPiloti/@Model.FileFoto" />
            <img src="data:image;base64,@System.Convert.ToBase64String(Model.Foto)" />
        </th>
```

```
    ...
</table>
```

I byte dell'immagine vengono trasferiti insieme alla pagina e codificati in base64. Questa tecnica, di per sé, non gestisce il caso in cui la proprietà `Foto` sia `null`. Una soluzione consiste nel verificare questa condizione:

```
@{ViewData["Title"] = "Pilota";}
...
<table class="content">

    <tr>
        <th colspan="2" class="textcenter"><h1>@Model.Nominativo</h1></th>
    </tr>
    <tr>
        <th colspan="2" class="textcenter">
            @if (Model.Foto != null)
            {
                <img src="data:image;base64,@System.Convert.ToBase64String(Model.Foto)" />
            }
            else
            {
                <img src="#" />
            }
        </th>
    ...
</table>
```

4.4.1 Implementare un metodo action che restituisce l'immagine

Un'alternativa è implementare un metodo *action* che restituisca l'immagine da visualizzare.

```
public FileContentResult GetFotoPilota(int id)
{
    var p = db.Piloti.Find(id);
    return File(p.Foto, "image/jpg");
}
```

Nota bene: il metodo restituisce un file incorporato in un oggetto di tipo `FileContentResult`; per farlo il metodo `File()`.

Il metodo viene richiamato direttamente dal tag **img**, specificando l'URL opportuno:

```
@{ViewData["Title"] = "Pilota";}
...
<table class="content">

    <tr>
        <th colspan="2" class="textcenter"><h1>@Model.Nominativo</h1></th>
    </tr>
    <tr>
        <th colspan="2" class="textcenter">
            <img src="/Home/GetFotoPilota/@Model.PilotaId" />
```

```
        </th>
    ...
</table>
```

4.5 Usare un database in memoria

La modularità di EF Core consente di utilizzare diversi *provider*, e dunque diversi DBMS, nella stessa applicazione. Particolarmente utile è la possibilità di gestire un database completamente in memoria, allo scopo di semplificare e velocizzare le fasi di sviluppo e test dell'applicazione.

A questo scopo è innanzitutto necessario installare il giusto provider, ad esempio mediante la **Package Manager Console**:

```
PM>Install-Package Microsoft.EntityFrameworkCore.InMemory
```

> ### InMemory database e modello relazionale
>
> Il provider **InMemory** non implementa un database relazionale e dunque, ad esempio, non può imporre il rispetto dei vincoli di integrità referenziale. Se si desidera gestire un database in memoria senza rinunciare al modello relazionale, occorre utilizzare il provdier SQLite: **Microsoft.EntityFrameworkCore.Sqlite**.

4.5.1 Configurare il context in modo che usi l'InMemory provider

Nel metodo `ConfigureServices()` della classe `Startup` scrivere:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MotoGPContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("MotoGPConnection")));

    services.AddDbContext<MotoGPContext>(options =>options.UseInMemoryDatabase());

    services.AddMvc();
}
```

4.5.2 Inserire i dati nel database

Poiché il database è gestito in memoria, quando parte l'applicazione le tabelle sono vuote; per simulare l'esistenza dei dati, questi devono essere inseriti automaticamente all'avvio. A questo scopo di può implementare un metodo che inserisca i dati; questo sarà eseguito in `Configure()` della classe `Startup`:

```
public void Configure(IApplicationBuilder app, ...)
{
    ...
    var db = app.ApplicationServices.GetService<MotoGPContext>();
    GeneraDatabase(db);
}
```

L'istruzione evidenziata ottiene un oggetto *context*, sulla base della configurazione effettuata nel metodo `ConfigureServices()`:

```
private static void GeneraDatabase(MotoGPContext db)
{
    if (db.Moto.Any()) //se ci sono già i dati, termina metodo
        return;

    //... inserisce moto e piloti
    db.SaveChanges();
}
```

5 Configurare l'applicazione

ASP.NET Core implementa un meccanismo di configurazione modulare che consente di stabilire i servizi utilizzati dall'applicazione. Di seguito ne introduco la struttura generale e fornisco un esempio di configurazione dell'oggetto *context* usato per accedere al database.

5.1 Classe Startup

Il codice di avvio di una applicazione ASP.NET è collocato nel metodo `Main()` dei file **Program.cs**: questo costruisce ed esegue l'oggetto che rappresenta il server web (e/o che dialoga con esso, se viene utilizzato un server web esterno, come IIS o Apache.)

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()  // stabilisce la classe di configurazione
        .Build();

    host.Run();  // esegue il server web
}
```

Viene anche stabilita la classe che conterrà il codice di configurazione: `Startup`.

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        // ... definisce l'origine delle opzioni di configurazione: file,
        //      variabili di ambiente, etc
    }

    public IConfigurationRoot Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        // ... definisce i servizi usati dall'applicazione
    }

    public void Configure(
        IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
    {
        // ... configura i servizi usati dall'applicazione
    }
}
```

5.1.1 Stabilire l'origine delle opzioni di configurazione

Nel costruttore viene creato l'oggetto che memorizza le opzioni di configurazione. Queste possono provenire da più fonti, come mostra il codice generato automaticamente da ASP.NET:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

Interessante è la riga evidenziata, che carica le opzioni memorizzate nel file **appsettings.json**. Più avanti utilizzerò questo file per memorizzare la stringa di connessione.

5.1.2 Stabilire i servizi utilizzati

La modularità di ASP.NET Core è dimostrata dal fatto che, di default, l'applicazione non fornisce alcun servizio fruibile dall'utente[4]; questi devono essere definiti nel metodo `ConfigureServices()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(); // aggiunge la gestione del pattern MVC
}
```

5.1.3 Configurare i servizi

Il metodo `Configure()` definisce il codice di configurazione dei servizi aggiunti nel metodo precedente (più quelli predefiniti di ASP.NET Core):

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())  // stabilisce la pagina di errore da mostrare in base al
    {                         // fatto che l'applicazione nello stato "sviluppo" o meno.
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles(); // configura MVC perché possa restituire i file statici (pagine
                          // HTML, css, immagini, etc.
```

4 Questo vale per il progetto "empty". Il progetto "web application" prevede appunto di aggiungere il servizio che implementa il pattern MVC.

```
app.UseMvc(routes =>      // stabilisce la route predefinita utilizzata da MVC
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Interessante l'ultima istruzione: definisce la *route* predefinita utilizzata da MVC per stabilire i *controller* e i metodi *action* da eseguire in risposta alle richieste dell'utente. (2.5)

5.2 Configurare il *context* in Startup

Intervenendo sulla classe `Startup` è possibile istruire ASP.NET Core a creare automaticamente il *context* e a passarlo ai *controller* che ne richiedono l'uso. Per farlo occorre aggiungere un nuovo servizio all'applicazione, nel metodo `ConfigureServices()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MotoGPContext>(
        options =>options.UseSqlServer("Server = (localdb)\\mssqllocaldb;..."));

    services.AddMvc();
}
```

Perché il *context* possa supportare questa modalità di creazione, è necessario che definisca l'opportuno costruttore, in grado di ricevere dall'esterno le opzioni di configurazione:

```
public class MotoGPContext: DbContext
{
    public MotoGPContext(DbContextOptions options):base(options) {}
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        ...
    }
    public DbSet<Pilota> Piloti { get; set; }
    public DbSet<Moto> Moto { get; set; }
}
```

Infine, nei *controller*, è sufficiente aggiungere un parametro al costruttore: sarà ASP.NET, quando crea il *controller*, a costruire il *context* e a passarlo come argomento:

```
public class HomeController : Controller
{
    IHostingEnvironment hostEnv;
    MotoGPContext db = new MotoGPContext();
    MotoGPContext db;

    public HomeController(IHostingEnvironment hostEnv, MotoGPContext db)
    {
        this.db = db;
        this.hostEnv = hostEnv;
```

```
    }
    ...
}
```

5.2.1 Memorizzare la stringa di connessione in appsettings.json

Un secondo vantaggio della creazione del *context* in `Startup` è quello di poter ottenere la stringa di connessione da una qualsiasi delle sorgenti utilizzate per la configurazione dell'applicazione. Di norma la stringa viene memorizzata nel file **appsettings.json**:

```
{
  "ConnectionStrings": {
    "MotoGPConnection": "Server=(localdb)\\mssqllocaldb;..."
  },

  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Per accedervi è necessario eseguire il metodo `GetConnectionString()` dell'oggetto `Configuration`, passando come argomento la chiave associata alla stringa:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MotoGPContext>(
        opt => opt.UseSqlServer(Configuration.GetConnectionString("MotoGPConnection")));

    services.AddMvc();
}
```

5.2.2 Conclusioni

Questo approccio, benché apparentemente più complicato, ha il grande vantaggio di centralizzare il codice di creazione e configurazione del *context*. Semplicemente modificando `Startup`, e senza intervenire nella classe *context*, è possibile cambiare la configurazione utilizzata, compresa l'origine del database.