# 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.

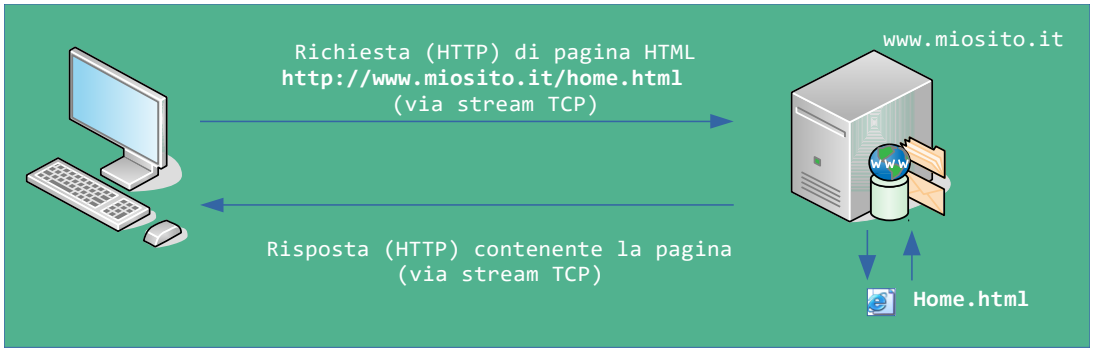
## 1.1 ASP.NET Core

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.

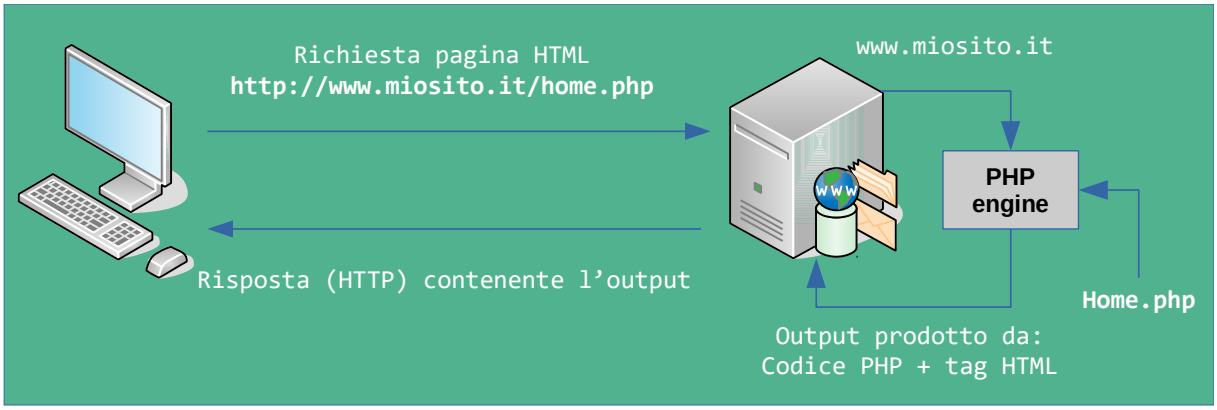
### *1.1.1 Accesso a una pagina HTML*

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. La richiesta e la 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:

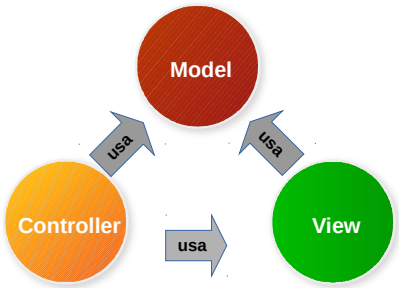



La pagina viene processata da un modulo software in grado di eseguire il codice PHP. Il risultato finale viene inviato al server (il quale riceve dunque del codice HTML), che lo ritrasmette al client.

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'importante 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, nella applicazione "Library", rientrano nel Model le classi entità: `Book`, `Author`, `Genre`, etc, le eventuali classi *repository* e qualsiasi classe e modulo utilizzato per rispondere ai requisiti (ad esempio, per implementare il prestito di un libro).
2. Nella categoria **View** rientrano i componenti dell'interfaccia utente; questi dovrebbero contenere il codice minimo necessario per presentare i dati. In una 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 mostra che il Model è completamente indipendente da View e Controller: l'intero 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.

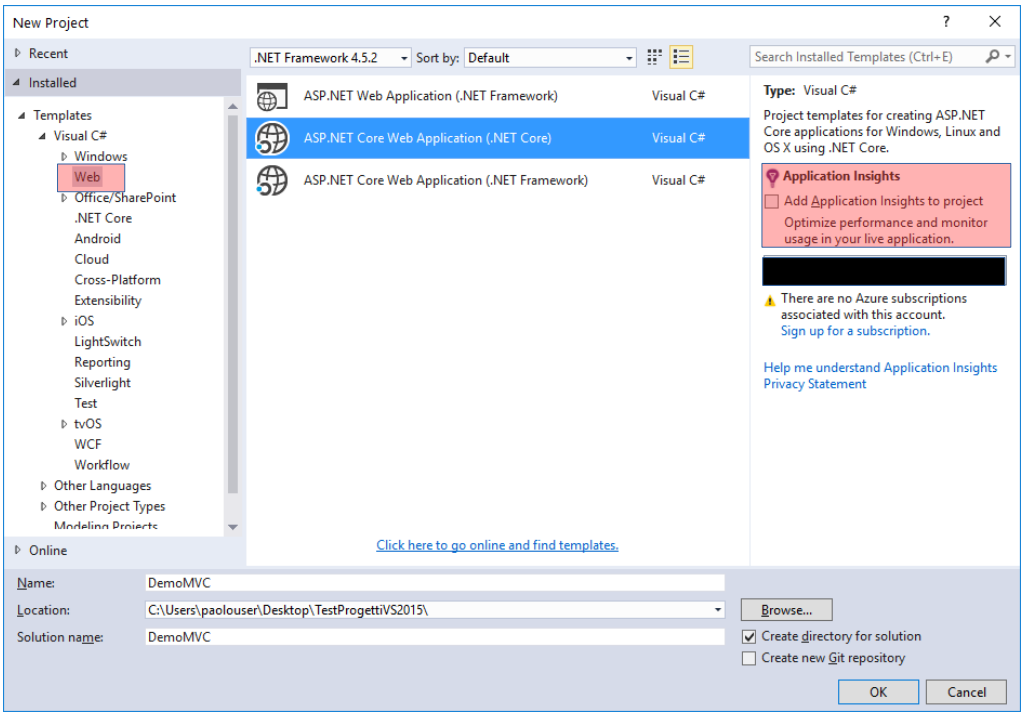
## 1.3 ASP.NET Core e MVC

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

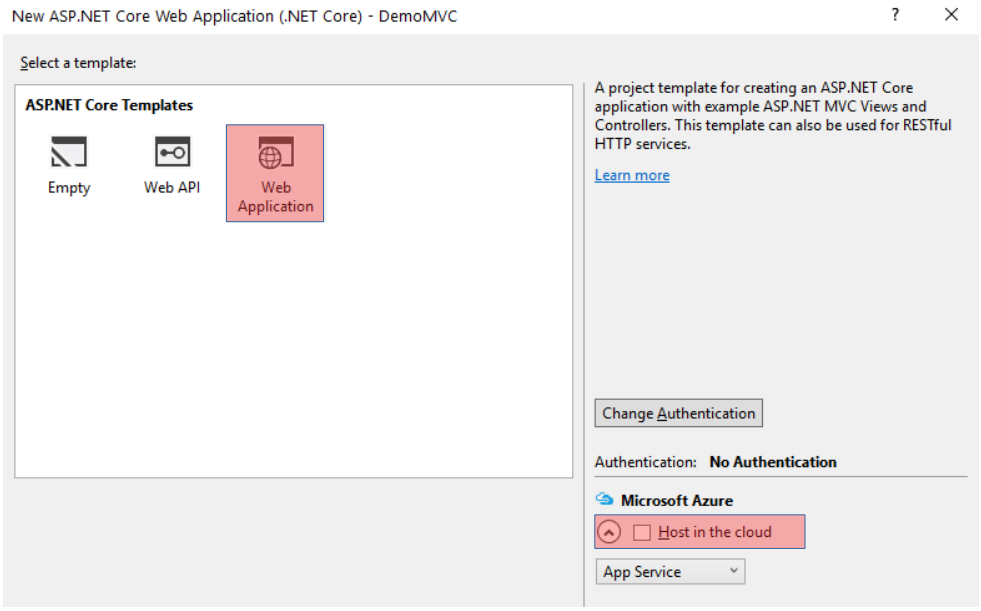
# 2 Architettura dei progetti ASP.NET Core MVC

## 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*:

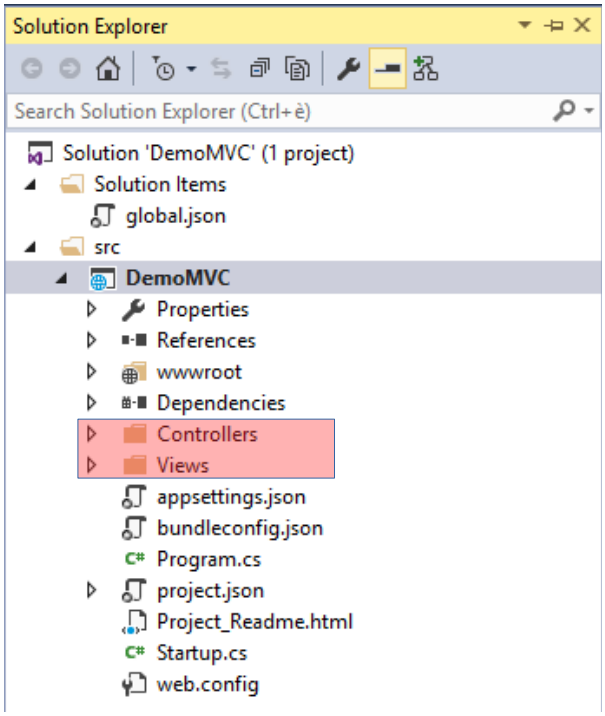



Nella successiva schermata occorre selezionare "Web Application"; ciò fa sì che venga creato un progetto dotato degli elementi principali e già funzionante. È opportuno disabilitare la voce *Host in the cloud*.

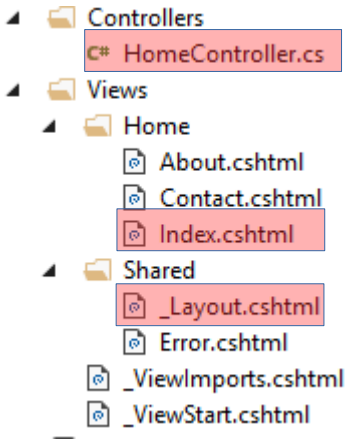


## 2.2 Struttura generale del progetto

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 questi file può essere ignorata; ciò che è realmente importante si trova nelle cartelle *Controllers* e *Views:*

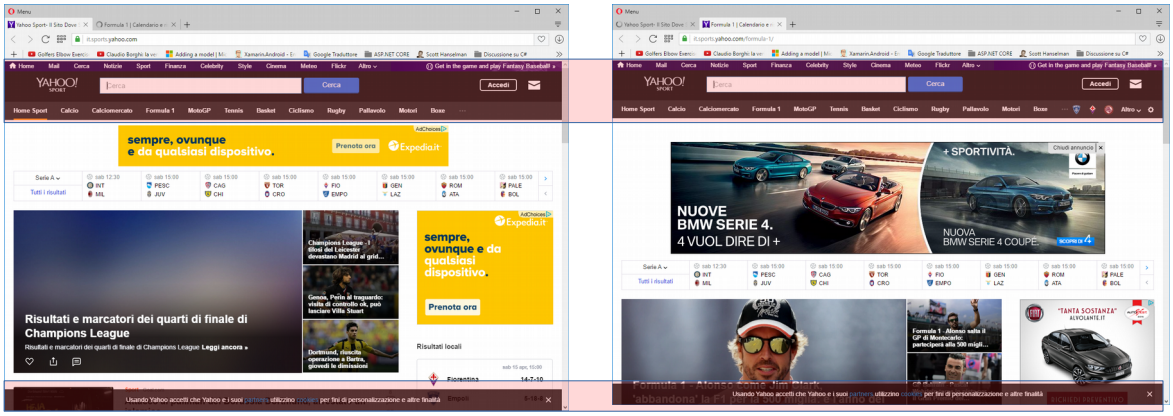



Di seguito mi focalizzerò sui file **HomeController**, **Index** e **_Layout**, i quali rappresentano l'ossatura di una applicazione MVC.

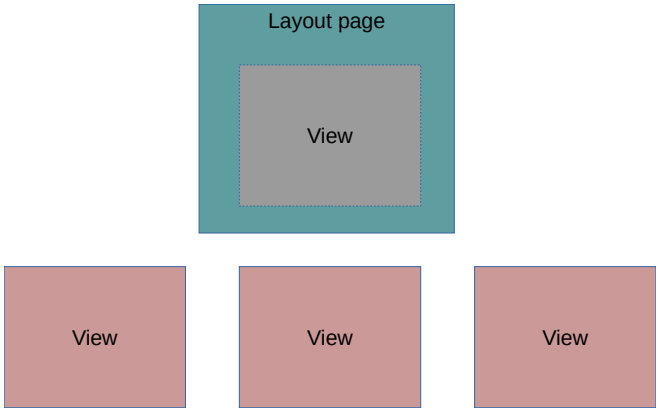
## 2.3 View, layout page e pagine web

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 dai metodi definiti nei *controller*. 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**

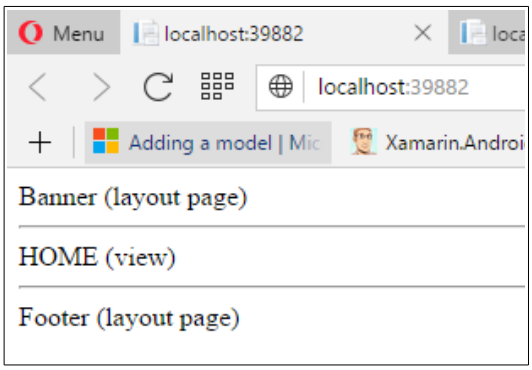
```
@{Layout = "_Layout";}

<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*: ad ogni metodo corrisponde una *view*.

Segue l'`HomeController`, generato durante la creazione del progetto (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()`: ed è questo metodo 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 dei 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 si può usare 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, 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 query string:

```
/[<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").

### 2.5.1 Uso di Tag Helper

I *tag helpers* 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* (e parametro, se presente):

```
@{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* possono essere considerate delle classi e come tali è necessario che definiscano 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 seguente link 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 Helpers** e dichiarano i *namespace*.
6. La view **About** viene caricata e integrata nella *layout page* **_Layout**.
7. Il codice e i *tag helpers* presenti in entrambi i file vengono processati. Il risultato è rappresentato da puro HTML (più javascript e CSS, se definiti)
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.

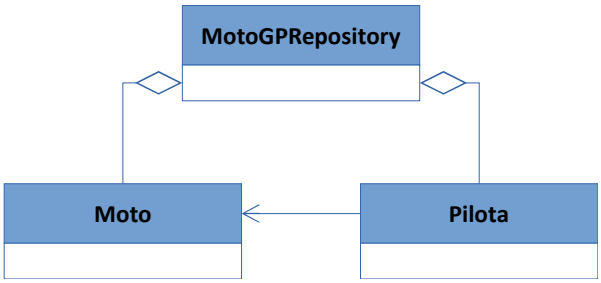
## 3.1 Descrizione dell'applicazione

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 collezioni di piloti e di moto (soltanto alcuni), simulando l'esistenza di un database, ma gestendo i dati in memoria. Le due collezioni 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.

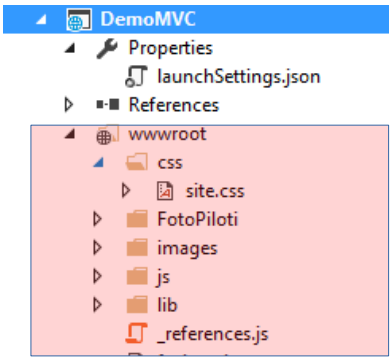
## 3.2 Layout page

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 solo memorizzati nella cartella "**wwwroot**", all'interno della quale sono presenti anche le foto dei piloti e i file javascript (che non uso).




## 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: se si usa 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();

    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
    }
}
```

Particolarmente interessante è il metodo `ElencoPilotiMoto()`. Il metodo 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.

### Usare un singolo metodo con parametro opzionale

In teoria sarebbe stato possibile risolvere il tutto in modo più elegante 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. Ho preferito pertanto 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, che saranno utilizzate successivamente nel codice HTML. Ciò mostra una caratteristica di *razor*: le variabili definite fuori dai metodi sono sempre 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 la selezione della 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"/></td>
            </tr>
            <tr>
                <td><label asp-for="Vittorie"></label></td>
                <td><input asp-for="Vittorie"/></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 nome del 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**". Per farlo è necessario 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 completo 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 ci scrive il file.
  - Imposta in `pilota` il nome del file.

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