

ASP.NET Core MVC

Introduzione alle applicazioni web MVC su ASP.NET Core

ASP.NET Core 2.0

Anno 2017/2018

Indice generale

1	Introduzione.....	5
1.1	ASP.NET Core.....	5
1.1.1	Accesso a una pagina HTML.....	5
1.1.2	Accesso a una “pagina server”	5
1.2	MVC.....	6
1.3	ASP.NET Core e MVC.....	6
2	Architettura dei progetti ASP.NET Core MVC.....	7
2.1	Creazione di un progetto.....	7
2.2	Struttura generale del progetto.....	8
2.3	View, layout page e pagine web.....	8
2.3.1	Usare automaticamente la layout page: _ViewStart.cshtml.....	10
2.3.2	Impostare il titolo della pagina nelle view: dizionario ViewData.....	10
2.4	Controller e View.....	11
2.4.1	Percorso di ricerca delle view.....	11
2.4.2	Tipo restituito dai metodi action.....	11
2.4.3	Passare dati alla view: uso di ViewData.....	11
2.4.4	Passare dati (strong typed) alla view: uso di @model.....	12
2.5	Eseguire i metodi <i>action</i> dei controller: routing.....	12
2.5.1	Uso di Tag Helper.....	13
2.6	Importare namespace e Tag Helpers nelle view: _ViewImports.....	13
2.7	Razor.....	14
2.8	Riepilogo.....	14
3	Applicazione MVC di esempio: MotoGP.....	15
3.1	Descrizione dell’applicazione.....	15
3.2	Model.....	15
3.3	View e controller.....	16
3.4	Layout page.....	16
3.5	Home page.....	17
3.6	Elenco moto.....	17
3.6.1	Utilizzare il Model nel controller.....	17
3.6.2	Accedere al Model nella view.....	18
3.6.3	Generazione degli hyperlink.....	18

3.7	Elenco dei piloti.....	19
3.7.1	Caricamento dei piloti.....	20
3.7.2	Binding della richiesta con i parametri del metodo.....	20
3.8	Informazioni sul pilota.....	21
4	Inserimento di nuovi dati.....	22
4.1	Nuovo pilota.....	22
4.1.1	Implementazione base del form.....	23
4.1.2	Processare i dati inseriti.....	24
4.2	Selezionare la moto da un elenco.....	24
4.3	Upload del file con la foto del pilota.....	25
4.3.1	Salvare il file su disco.....	26
5	Validare i dati.....	28
5.1	Validazione dei singoli campi del pilota.....	28
5.2	Visualizzazione degli errori: uso dei tag helper.....	29
5.2.1	Messaggi di errore riepilogativi: aggiungere errori al modello.....	30
5.2.2	Personalizzare i messaggi di errore.....	30
5.3	Validazione generale del modello.....	31
6	Usare Entity Framework.....	32
6.1	Aggiungere EF a un progetto ASP.NET Core.....	32
6.2	Configurare e utilizzare l'oggetto context.....	32
6.2.1	Configurazione del model.....	33
6.2.2	Uso del context.....	33
6.3	Definire un ViewModel.....	34
6.4	Visualizzare le immagini memorizzate nel database.....	35
6.4.1	Implementare un metodo action che restituisce l'immagine.....	36
6.5	Usare un database in memoria.....	37
6.6	Configurare il context in modo che usi l'InMemory provider.....	37
6.7	Inserire i dati nel database.....	37
7	Configurare l'applicazione.....	39
7.1	Classe Startup.....	39
7.2	Impostare i servizi utilizzati.....	40
7.3	Configurare i servizi.....	40

7.4	Configurare il <i>context</i> in Startup.....	40
7.4.1	Memorizzare la stringa di connessione in appsettings.json.....	41
7.4.2	Conclusioni.....	42

1 Introduzione

Il tutorial introduce gli elementi essenziali delle applicazioni web realizzate mediante il framework ASP.NET Core e che utilizzano il *design pattern* **Model View Controller**.

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.

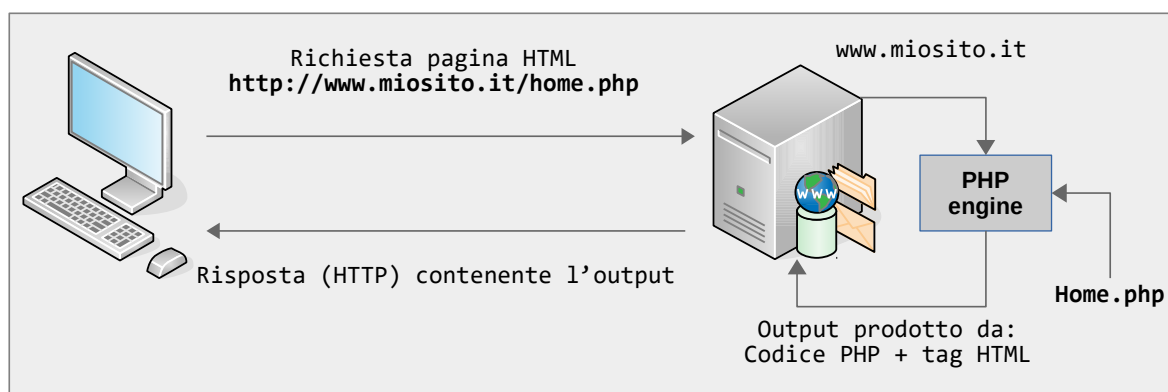
1.1.1 Accesso a una pagina HTML

L'accesso a una 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:

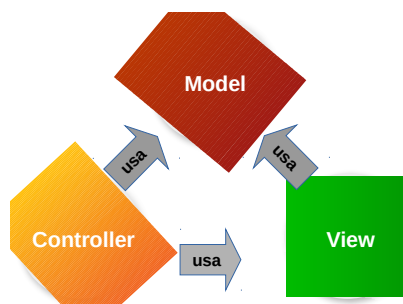


La pagina viene processata da un modulo software in grado di eseguire il codice PHP. Il risultato finale viene inviato al server, che lo ritrasmette al client (il quale riceve dunque del codice HTML).

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

1.2 MVC

Il *design pattern* **Model View Controller** 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, considerando l'applicazione **Library** fanno parte del Model le classi `Book`, `Author`, `Genre`, etc, eventuali 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. 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 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 indipendente 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.

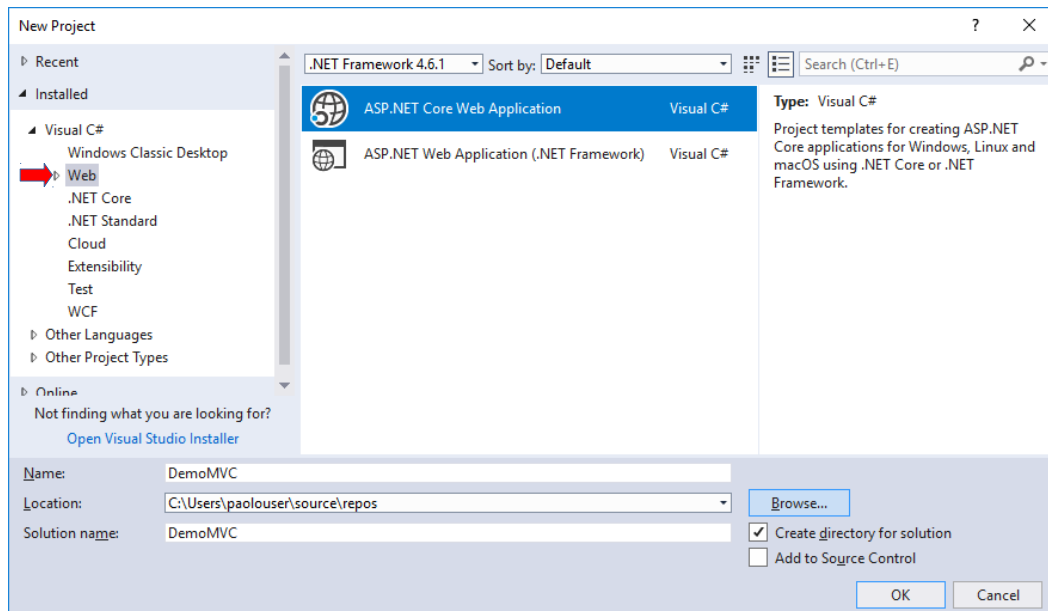
1.3 ASP.NET Core e MVC

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

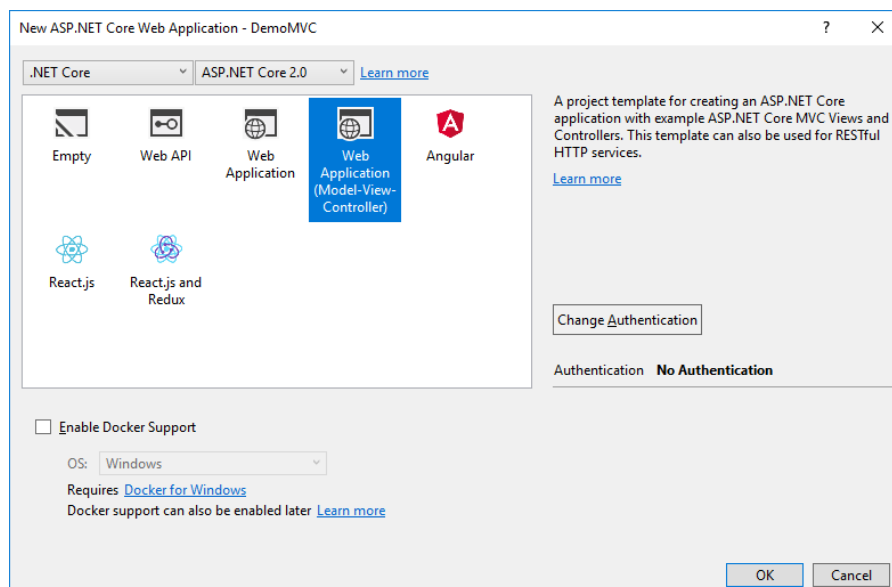
2 Architettura dei progetti ASP.NET Core MVC

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:¹



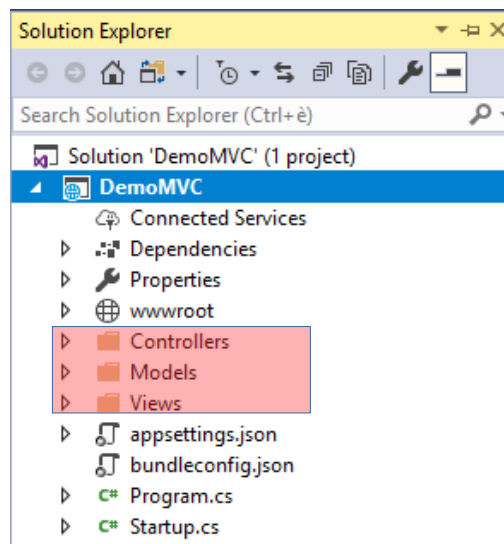
Nella schermata successiva occorre selezionare **Web Application (Model-View-Controller)**; ciò produce la creazione di un progetto dotato degli elementi principali e già funzionante.²



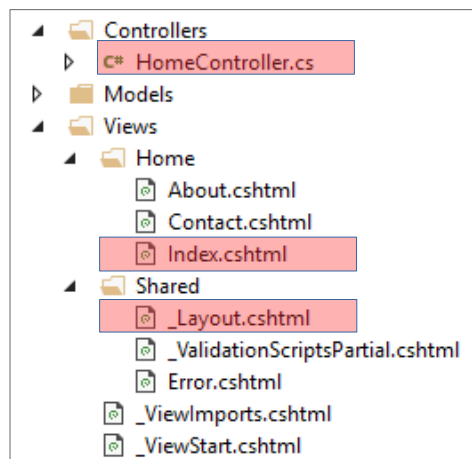
- 1 Se è stata installata la funzionalità **Application Insights**, conviene disabilitarla; ciò riduce le dimensioni del progetto e ne aumenta le performance.
- 2 Se sono state installate le funzionalità di Azure, è opportuno disabilitare la voce **Host in the cloud**.

2.2 Struttura generale del progetto

In *Solution Explorer* appare la seguente schermata:



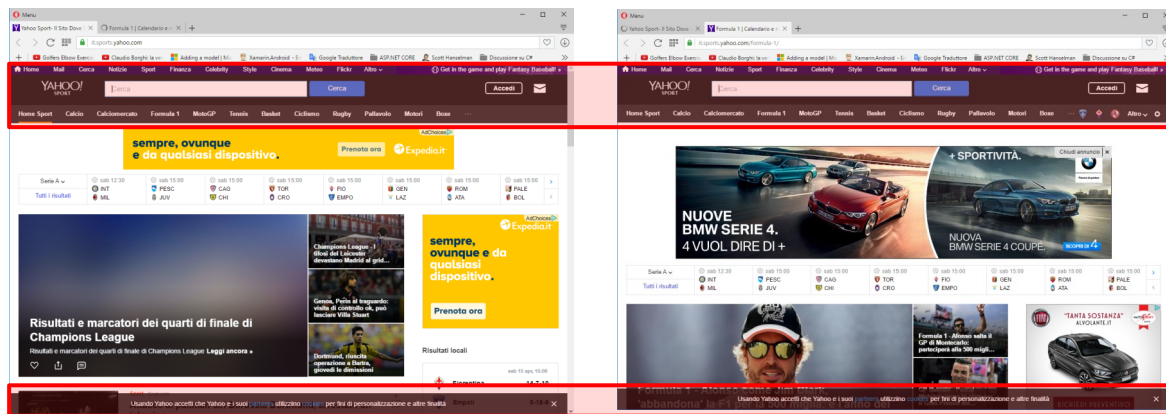
Il progetto definisce un prototipo di applicazione funzionante, comprensivo di *view*, *controller*, librerie javascript, fogli di stile e immagini. La maggior parte dei file può essere ignorata; ciò che interessa si trova nelle cartelle *Controllers* e *Views*:



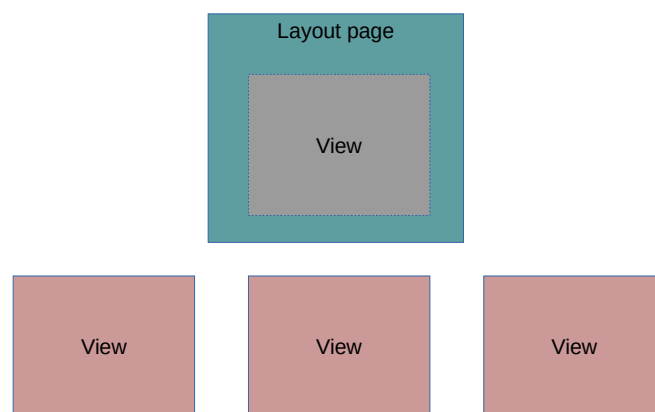
Di seguito mi focalizzerò sui file **HomeController**, **Index** e **_Layout**, i quali rappresentano l'ossatura di una applicazione MVC.

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 caricate da codice. Inoltre, 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 definisce i contenuti comuni e dunque è 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";}

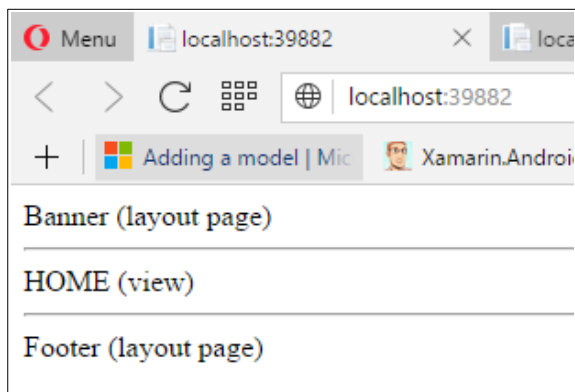
<div>
    HOME (view)
</div>
```

_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`

Nelle view non è necessario 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";}
```

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, in HTML, viene impostato mediante il tag **title**, contenuto in **head**. Ciò pone un problema: la pagina visualizzata è rappresentata dalla *view*, ma il tag **head** è specificato 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>
```

Come vedremo più avanti, `ViewData` 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'*home controller* (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()` e ne restituisce il risultato. 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 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	Index
<pre>public IActionResult Index() { ViewData["messaggio"] = "Hello, World"; return View(); }</pre>	<pre>@{ViewData["Title"] = "Home";} <div> HOME (view) <p>@ViewData["messaggio"]</p> </div></pre>

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

Nella classe **Startup.cs** (7.1), 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 omissso, viene utilizzato l'`HomeController`.
- Il secondo campo specifica il metodo *action*; se omissso 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	<code>Index()</code> di <code>HomeController</code>
/	<code>Index()</code> di <code>HomeController</code>
/home/GetUser/12	<code>GetUser(int id)</code> di <code>HomeController</code>
/Catalog/Index	<code>Index()</code> di <code>CatalogController</code>
/Catalog/GetBooks?genre="Fantascienza"&auth=1	<code>GetBooks(string genre, int auth)</code> ...

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):

["http://localhost:5000/home/about"](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 successivamente compilate; come tali devono definire i *namespace* necessari. Per evitare di dover dichiarare i *namespace* in ogni *view*, è possibile farlo una volta per tutte nel file **_ViewImports.cshtml**; di default ha il seguente contenuto (supposto **DemoMVC** il nome del progetto):

```
@using DemoMVC  
@using DemoMVC.Models  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Nota bene: la direttiva `addTagHelper` rende disponibili i *tag helpers* per tutte le *view*.

2.7 Razor

Razor identifica la tecnologia che consente di utilizzare C# nelle *view*. Il link seguente contiene 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 ed processata; il risultato viene integrato nella *layout page* **_Layout**, precisamente nella posizione specificata da `RenderBody()`.
7. Anche la *layout page* viene processata; il risultato finale è puro HTML (più javascript e CSS, se definiti), che viene inviato al client.

3 Applicazione MVC di esempio: MotoGP

L'applicazione 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. Analizzerò il funzionamento dell'applicazione considerando separatamente la visualizzazione, l'inserimento e la validazione.

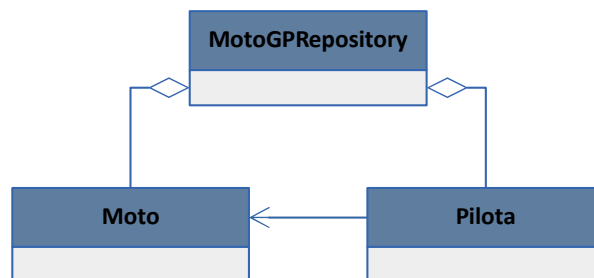
3.1 Descrizione dell'applicazione

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.

3.2 Model

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



```
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.

3.3 View e controller

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

- **Index**: è la home page e mostra semplicemente una foto.
- **ElencoMoto**: visualizza l'elenco delle marche iscritte al mondiale; consente di cliccare su una moto e ottenere l'elenco dei piloti che corrono con essa. (*view* **ElencoPilota**)
- **ElencoPilota**: 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.

3.4 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>
            
        </header>
        <menu>
            <ul>

```

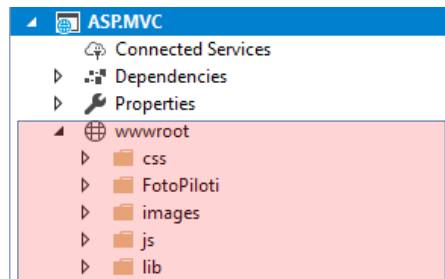


```

        <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 referencia 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.



3.5 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">
    
</div>

```

Index()

```

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

```

3.6 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.

3.6.1 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()`.

3.6.2 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.

3.6.3 Generazione degli hyperlink

Di seguito 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 referencia 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 possiamo scrivere:

```

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

```

3.7 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.

3.7.1 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();
    //risponde alla richiesta: /Home/ElencoPiloti
    public IActionResult ElencoPiloti()
    {
        return View(repo.GetPiloti());
    }

    //risponde alla 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()`.

3.7.2 Binding della richiesta con i parametri del metodo

Considerando la *route* predefinita:

/Controller=Home/Action=Index/Id?

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

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**.

3.8 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, carica il pilota corrispondente 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">
            
        </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.

4 Inserimento di nuovi dati

In questo scenario, l'unica modifica prevista è l'inserimento di un nuovo pilota.

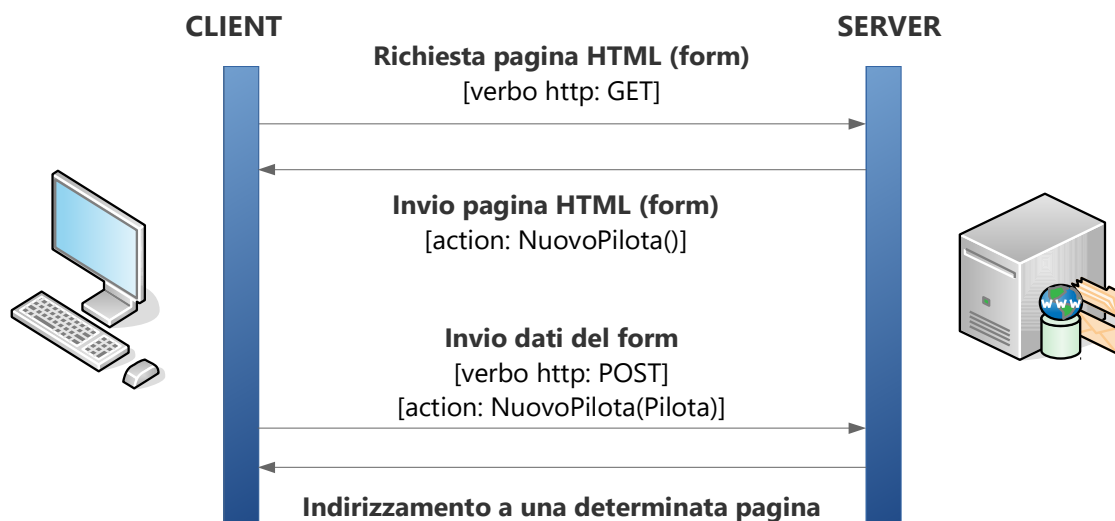
4.1 Nuovo pilota

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 gestione di un form HTML avviene in due fasi:

- Il browser chiede la pagina contenente il form. Il server risponde con un form vuoto.
- Il browser invia al server i dati inseriti dall'utente (invio del form). Il server processa i dati, li verifica e, dopo l'inserimento, reindirizza il browser a una nuova pagina.



Il form è gestito mediante due metodi *action*. Il primo metodo carica la *view* contenente il form; il secondo riceve i dati inseriti nel form 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, qualifica il metodo come un' *action* che risponde a una richiesta del form da parte del browser. Corrisponde al verbo HTTP **GET**.

4.1.1 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 l'impiego del verbo **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:

- 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**.

4.1.2 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 inseriti nel form.
- Sulla base del *model* dichiarato nella *view*, ASP.NET è in grado di "bindare" i dati ricevuti alle corrispondenti proprietà del parametro `pilota`.
- Dopo aver inserito il pilota nel *repository*, il metodo reindirizza l'utente alla pagina elenco piloti utilizzando `RedirectToAction()`, 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.)

4.2 Selezionare la moto da un elenco

Il modo corretto per l'inserimento della moto del 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 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; 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** permette 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>
```

4.3 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 del 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.

4.3.1 Salvare il file su disco

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

5 Validare i dati

I controller dovrebbero sempre validare i dati prima di elaborarli. In caso di errore, dovrebbe essere riproposto il form di inserimento, in modo che l'utente possa correggere l'input. A questo proposito, ASP.NET fornisce un meccanismo di validazione che consente:

- Di validare il contenuto dei singoli campi di input, in accordo a determinati criteri.
- Di validare l'input nel suo insieme, verificando che i dati inseriti siano coerenti tra loro e con lo stato dell'applicazione.

In entrambi i casi è possibile stabilire il contenuto e la modalità di visualizzazione dei messaggi di errore.

5.1 Validazione dei singoli campi del pilota

È possibile utilizzare il meccanismo di validazione automatica specificando nel *model* i criteri che i dati devono 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; }
}
```

Gli attributi stabiliscono i criteri utilizzati per stabilire la validità dei dati inseriti. (Esistono altri tipi di attributi, che consentono un elevato livello di personalizzazione nella validazione dei campi.)

Nel metodo *action* che elabora il form, prima di procedere all'elaborazione dell'input, 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);
    }

    // ... procede all'inserimento

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

5.2 Visualizzazione degli errori: uso dei tag helper

In risposta a un input errato, è possibile visualizzare un messaggio di errore riepilogativo e/o dei messaggi corrispondenti ai campi non validi. ASP.NET 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 del processo di validazione. L'approccio standard è quello di utilizzare dei tag **span**, adiacenti ai campi di input, per mostrare i singoli errori, e un tag **div** che riepiloghi gli errori e/o visualizzi messaggi che riguardano la validità del modello in generale.

```
@{
    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="Nome"></label></td>
                <td><input asp-for="Nome"/>
                    <span asp-validation-for="Nome" class="error-text"></span>
                </td>
            </tr>
            <tr>
                <td><label asp-for="Cognome"></label></td>
```

```

        <td><input asp-for="Cognome"/>
        <span asp-validation-for="Cognome" class="error-text"></span>
    </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**. 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.

5.2.1 Messaggi di errore riepilogativi: aggiungere errori al modello

Se impostato al valore **ModelOnly**, il tag helper **asp-validation-summary** non produce alcun output, a meno che non siano 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);
    }
    ...
}

```

Nota bene: il primo parametro del metodo `AddModelError()` è vuoto; ciò contraddistingue un errore di riepilogo ad uno che riguarda uno specifico campo.

5.2.2 Personalizzare i messaggi di errore

I messaggi di errori prodotti dagli attributi che decorano il modello sono stabiliti da ASP.NET e sono in inglese. È possibile personalizzarli, oppure semplicemente rimpiazzarli con un simbolo, quando la natura dell'errore è evidente.

```

public class Pilota
{
    ...
    [Required(ErrorMessage="*")]
    public string Nome { get; set; }
}

```

```

[Required(ErrorMessage="*")]
public string Cognome { get; set; }

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

...
}

```

5.3 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);
    }

    ...
}

```

Segue uno *screen shot* che mostra il risultato del processo di validazione. Il form è stato inviato senza aver inserito il nome e con un numero della moto non valido:

Uno o più campi non contengono dati corretti

Nome *

Cognome

Numero
Il numero deve essere compreso tra 1 e 99

Moto

Punti

Vittorie

File foto

6 Usare Entity Framework

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 alcune funzionalità, come ad esempio il *lazy loading*.³ (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.

6.1 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**.

Dalla versione ASP.NET Core 2.0, questo pacchetto è automaticamente incluso nel progetto e dunque non è necessario installarlo.

6.2 Configurare e utilizzare l'oggetto context

La classe `MotoGPContext` fornisce l'accesso al database **PilotiMotoGP**, che memorizza le tabelle **Piloti** e **Moto**.

Innanzitutto occorre istruire il *context* sulla stringa di connessione da usare; un modo è stabilire la stringa di connessione nel metodo `OnConfiguring()`:

```
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; }
}
```

3 Questa funzionalità è in fase di implementazione e potrebbe essere rilasciata nella prossima versione.

6.2.1 Configurazione del model

La configurazione del model non richiede niente di particolare, se non la mappatura delle *entità* mediante attributi, poiché la convenzione sui nomi le assocerebbe alle tabelle **Motos** e **Pilotis**.

```
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; }
}
```

6.2.2 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*, senza considerare nell'ambito delle applicazioni web, il *lazy loading* è normalmente inutilizzabile.

6.3 Definire un ViewModel

La nuova versione di `Pilota` incorpora l'immagine corrispondente, adesso memorizzata nel database e non più in un file separato⁴. 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 definire una nuova classe 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 `PilotaInfo`:

```
// 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
    });
}

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

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

4 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 le performance.

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>
```

6.4 Visualizzare le immagini memorizzate nel database

La view **InfoPilota** visualizza la foto del pilota mediante un tag **img** che riferenzia il file contenente l'immagine, collocato nella cartella **FotoPiloti**; il nome del file è memorizzato nella proprietà **FileFoto** di **Pilota**. Ma se le immagini sono memorizzate nel database e accessibili mediante la proprietà **Foto**, occorre adottare una tecnica diversa per 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 **base64string**.

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">
            
            
        </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)
            {
                
            }
            else
            {
                
            }
        </th>
        ...
    </table>

```

6.4.1 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: viene restituito un file incorporato in un oggetto di tipo `FileContentResult`; per farlo uso il metodo `File()`.

Il metodo *action* viene richiamato direttamente dal tag **img**, specificando l'URL opportuno e passando l'id del pilota:

```

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

    <tr>
        <th colspan="2" class="textcenter"><h1>@Model.Nominativo</h1></th>
    </tr>
    <tr>
        <th colspan="2" class="textcenter">
            
        </th>
        ...
    </table>

```

6.5 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 provider SQLite: **Microsoft.EntityFrameworkCore.Sqlite**.

6.6 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.UseInMemoryDatabase("MotoGP"));

    services.AddMvc();
}
```

(Il nome fornito al database, `"MotoGP"`, non è significativo.)

6.7 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 si 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();
}
```

7 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.

7.1 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)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
}
```

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

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

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

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

7.2 Impostare i servizi utilizzati

La modularità di ASP.NET Core è dimostrata dal fatto che, di default, l'applicazione non fornisce alcun servizio⁵; questi devono essere definiti nel metodo `ConfigureServices()`:

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

7.3 Configurare i servizi

Il metodo `Configure()` configura i servizi specificati nel metodo precedente.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    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.

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

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)

7.4 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()`.

Nell'esempio seguente configuro il *context*, facendo in modo che usi una determinata stringa di connessione:

⁵ Questo vale per il progetto "empty". Il progetto "web application" prevede appunto di aggiungere il servizio che implementa il pattern MVC.


```

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

    services.AddMvc();
}

```

Perché il *context* possa utilizzare questa modalità di creazione, è necessario che definisca un costruttore appropriato, 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 necessario 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;
    }
    ...
}

```

7.4.1 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();
}

```

7.4.2 Conclusioni

Questo approccio ha il 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.

8 Autenticazione dell'utente

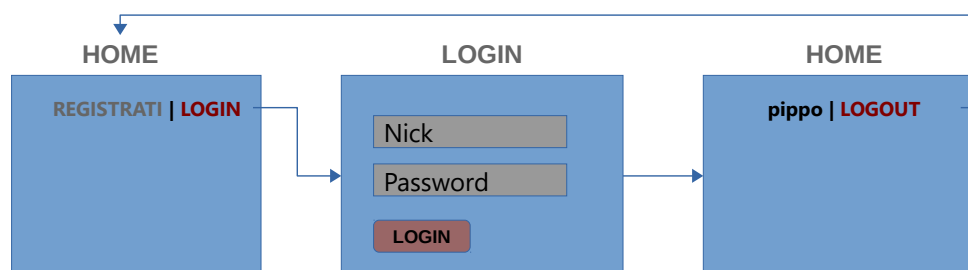
Il termine *autenticazione* disegna il processo con il quale un'applicazione stabilisce l'identità dell'utente, allo scopo di fornire servizi specifici e/o autorizzare (o negare) l'accesso a determinate risorse. Tale processo si basa su alcune premesse:

- L'utente deve essere stato precedentemente registrato; in alternativa.⁶
- L'utente, inizialmente anonimo, deve fornire le proprie credenziali per essere autenticato (*login*). È possibile usare e credenziali di un servizio esterno (Google, Facebook, Microsoft, etc)
- L'applicazione deve memorizzare lo stato dell'utente per tutta la durata della sessione.
- L'applicazione deve fornire la possibilità all'utente di ritornare anonimo (*logout*)

ASP.NET Core fornisce tutti i servizi necessari al processo di autenticazione; di seguito introduco le basi minime per implementare i processi di *login*, *logout*, e per conoscere lo stato dell'utente: anonimo/autenticato.

8.1 Implementazione dei processi di login/logout

In generale, i processi di *login* e *logout* possono essere schematizzati nel seguente modo:



La *home* consente agli utenti anonimi di autenticarsi e a quelli già autenticati di eseguire il *logout*. *login* è un form HTML che chiede le credenziali dell'utente. L'invio del form produce l'esecuzione del metodo `Login()` dell'*home controller*.

8.1.1 Login

Come ogni form, anche quello di *login* è gestito mediante due metodi; il primo che carica il form, il secondo che ne elabora i dati:

```
using Microsoft.AspNetCore.Authentication.Cookies;
using System.Security.Claims;
using Microsoft.AspNetCore.Http;
...

public class HomeController : Controller
{
    public IActionResult Login()
    {
    }
```

⁶ ASP.NET Core fornisce anche un'infrastruttura per la registrazione e la memorizzazione del profilo utente.

```

        return View();
    }

    [HttpPost]
    public IActionResult Login(User user)
    {
        ...
    }
}

```

Il metodo `Login(User)` ha la funzione di:

1. Validare i dati inseriti.
2. Verificare che le credenziali corrispondano a un utente registrato.
3. Eseguire il **sign-in**: l'utente viene riconosciuto come autenticato.

Di seguito mostro il codice che esegue l'ultima fase:

```

[HttpPost]
public IActionResult Login(User user)
{
    //... valida i dati
    //... verifica esistenza utente
    SignIn(user);
    return RedirectToAction("Index");
}

```

```

public void SignIn(User user)
{
    string scheme = CookieAuthenticationDefaults.AuthenticationScheme; //-> "Cookies"

    var identity = new ClaimsIdentity(scheme);
    identity.AddClaim(new Claim(ClaimTypes.Name, user.FullName));
    var principal = new ClaimsPrincipal(identity);

    HttpContext.SignInAsync(scheme, principal).Wait();
}

```

Il metodo `SignIn()`:

- Stabilisce il tipo di autenticazione: basata su *cookie*.
- Crea un'*identità* per l'utente da autenticare. A questa identità associa il nome completo dell'utente, memorizzato nel record `user`.
- Eseguire il *sign-in*: ogni successiva richiesta dell'utente (nella stessa sessione) viene associata all'*identità* suddetta.

(Nota bene: il metodo `SignInAsync()` è asincrono e restituisce un *task*; la chiamata al metodo `Wait()` sospende l'esecuzione fintantoché il task non è completato. Non è l'approccio corretto per eseguire un metodo asincrono; qui lo uso soltanto per semplicità.)

8.1.2 Logout

Il *logout* si riduce all'invocazione di un unico metodo:

```
public class HomeController : Controller
{
    ...

    public IActionResult Logout() // chiamato dal link Logout della home page
    {
        string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
        HttpContext.SignOutAsync(scheme).Wait();
        return RedirectToAction("Index");
    }
}
```

Dopo l'esecuzione di `SignOutAsync()`, le successive richieste dell'utente non sono più associate all'*identità* precedentemente creata: l'utente è ritornato ad essere anonimo.

HttpContext

`HttpContext` è una proprietà dell'*home controller* che fornisce l'accesso a tutte le informazioni e i servizi relativi al *contesto* della richiesta in corso e, in generale, della sessione dell'utente. Come vedremo, questo oggetto è accessibile (sotto altro nome) anche nelle *view*.

8.2 Visualizzazione dello stato dell'utente

Nell'esempio schematizzato, le informazioni visualizzate in *home* dipendono dal fatto che l'utente sia autenticato oppure no. A questo scopo occorre ottenere l'identità associata all'utente e quindi verificare se è autenticata o anonima.

```
@using Microsoft.AspNetCore.Http

@{
    var identity = Context.User.Identity;
}

<div>
    @if (identity.IsAuthenticated)
    {
        <span>@identity.Name | </span>
        <a asp-controller="Home" asp-action="Logout">Logout</a>
    }
    else
    {
        <a asp-controller="Home" asp-action="Login">Login</a>
        <a asp-controller="Home" asp-action="Register">Register</a>
    }
}
```

</div>

Alcune considerazioni:

- `Context`, pur con nome diverso, riferenzia lo stesso oggetto della proprietà `HttpContext` usata nell'*home controller*.
- `Name` (proprietà di `User.Identity`) memorizza il nome dell'utente, impostato nel metodo `SignIn()` dall'istruzione:

```
identity.AddClaim(new Claim(ClaimTypes.Name, user.FullName));
```

8.3 Configurazione del servizio di autenticazione

Il servizio di autenticazione, come qualsiasi altro servizio di ASP.NET Core, deve essere configurato nella classe `Startup`.

```
public class Startup
{
    ...
    public void ConfigureServices(IServiceCollection services)
    {
        string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
        services.AddAuthentication(scheme).AddCookie();
        ...
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        ...
        app.UseStaticFiles();

        // questa istruzione deve precedere UseMvc()
        app.UseAuthentication();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

Nota bene: il metodo `AddCookie()` esiste in più versioni, e consente di personalizzare il servizio di autenticazione.