ASP.NET Core MVC

Introduzione alle applicazioni web MVC su ASP.NET Core

Anno 2020/2021

Indice generale

1 Introduzione

Il tutorial introduce i fondamenti sullo sviluppo di applicazioni web mediante il framework ASP.NET Core e utilizzando il *design pattern* **M**odel **V**iew **C**ontroller.

Prima di procedere a esaminare l'architettura il funzionamento delle applicazioni di ASP.NET Core, intendo fare un riepilogo sul funzionamento delle applicazioni web in generale.

1.1 Applicazioni web

Il termine *applicazione web* designa un'applicazione *distribuita*, nella quale un processo, in esecuzione su un *server*, offre dei servizi a dei *client*, processi in esecuzione sulle macchine degli utenti finali.[1] I client comunicano con il processo server utilizzando il protocollo HTTP.

La parte server dell'applicazione è "ospitata" all'interno di un processo chiamato *web server*[2], il quale fornisce i servizi fondamentali – connettività, *storage*, etc – ed è in grado di ospitare più applicazioni nello stesso momento. Tra i web server più famosi vi sono **Apache** e **Internet Information Services**.

Di seguito, e per tutto il tutorial, mi occuperò soltanto della parte server.

1.2 Siti web

Un *sito web* è un tipo di applicazione web nel quale i client sono dei *browser* – Chrome, Opera, Safari, Edge, etc. Questi consentono all'utente di fruire dei contenuti e dei servizi forniti dal server, di solito sotto forma di pagine HTML.

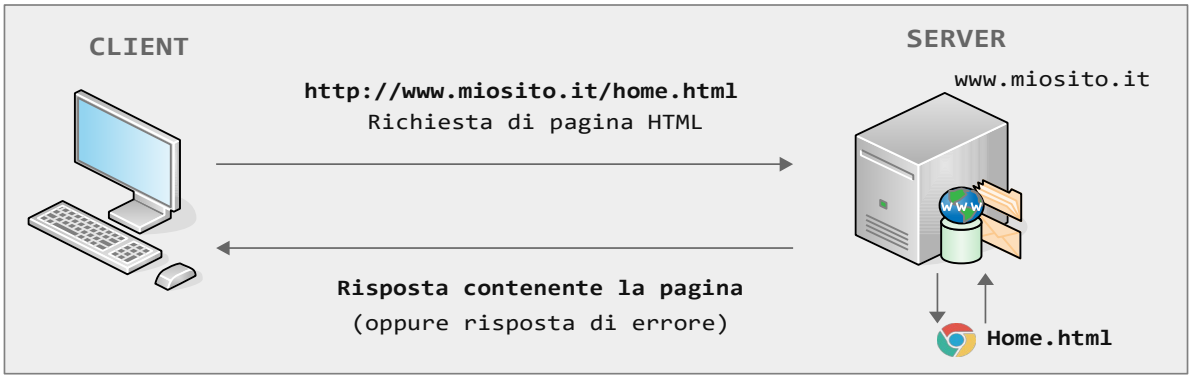
1.2.1 Siti web statici

Un *sito web statico* fornisce un servizio di natura "documentale". Il client – il browser – richiede un documento specificandone il percorso e il nome - l'**URL** -, il server lo invia al client. Nella maggior parte dei casi i documenti richiesi sono pagine HTML, le quali rendono possibile la navigazione tra i contenuti del sito mediante gli *hyperlink*.

A pagina successiva è mostrata una tipica interazione client-server. Il client esegue una richiesta HTTP che, nell'URL, specifica la pagina **home.html**. Il server cerca la pagina nel proprio spazio e ne invia il contenuto al client mediante una risposta HTTP. Se la pagina non viene trovata, il server invia una risposta di errore.

1 In realtà un client è qualunque processo richieda i servizi dell'applicazione web. Può accedere che un processo server sia a sua volta un client di un altro processo server.

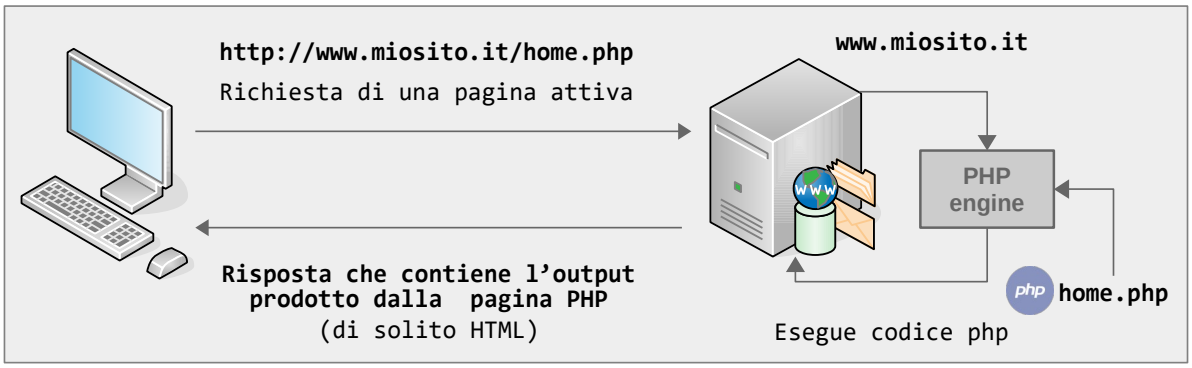
2 Spesso, si usa il termine *web server* anche per indicare la macchina fisica che ospita l'applicazione.



1.2.2 Siti web dinamici

Nei siti web dinamici i contenuti inviati al client sono in parte o in tutto prodotti mediante l'esecuzione di codice. Esistono varie tecnologie, ma la più comune è l'uso di cosiddette *active server pages*. Queste sono dei file che integrano al proprio interno sia codice HTML che codice esecutivo, scritto in un linguaggio che il server può processare.

Nello schema seguente, il client richiede una pagina PHP. Il server, dopo averla riconosciuta come una *pagina attiva*, non si limita a caricarla e a inviarla al client, ma la processa, eseguendo il codice PHP. Il risultato finale, tipicamente semplice HTML, viene inviato al client.

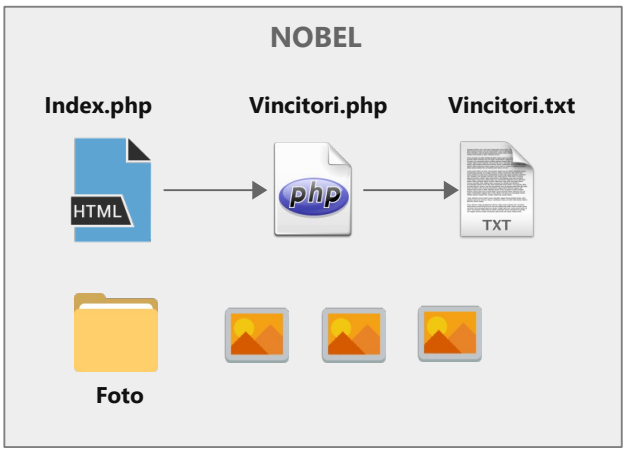


È fondamentale sottolineare che il processo di esecuzione della pagina – **home.php**, nell'esempio – avviene interamente nel server. Il client riceve soltanto il risultato prodotto dalla sua esecuzione.

1.3 Un semplice sito web di esempio

Di seguito propongo un semplicissimo sito composto da due pagine, una HTML, l'altra PHP. Il mio obiettivo è mostrare un sito con contenuti statici e dinamici e usarlo come punto di riferimento quando introdurrò il funzionamento di ASP.NET Core.

La home page del sito, **Index.html**, contiene un *hyperlink* alla pagina PHP, **Vincitori.php**, la cui funzione è visualizzare un elenco di vincitori di premio Nobel. Le informazioni sui vincitori sono memorizzate nel file, in formato CSV, **Vincitori.txt**. (Esiste anche una cartella, **Foto**, contenente le le foto dei vincitori.)



La pagina **Index.html** è una risorsa statica: il server ne invia il contenuto al client senza eseguire alcuna elaborazione

```
<html>
<head>
...
</head>
<body>
    <h1>NOBEL PRIZE</h1>
    <hr />
    <p><a href='Vincitori.php'>Elenco vincitori</a></p>
</body>
</html>
```

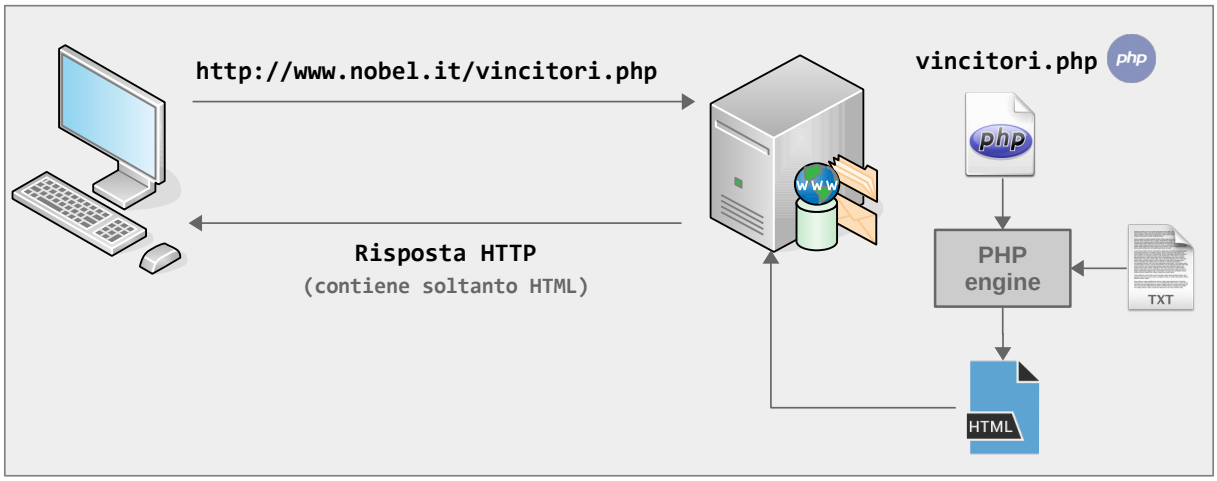
Quando l'utente clicca il link, il client richiede al server la pagina **Vincitori.php**:

```
<html>
<head>
...
</head>
<body>
    <?php

    $righe = file("Vincitori.txt");
    foreach ($righe as $value)
    {
        $items = str_getcsv($value, ";");
        echo '<p>' . $items[0] . ', ' . $items[1] . '</p>';
    }
    ?>
</body>
</html>
```

Questa è una *pagina attiva*: il server la processa ed esegue il codice PHP, il cui output viene integrato con la parte HTML della pagina. L'output finale viene inviato al client:

```
<html>
<head>
    ...
</head>
<body>
    <p>Einstein,  Albert </p>
    <p>Fermi,  Enrico </p>
    <p>Feynman,  Richard </p>
</body>
</html>
```

Lo schema riassume il processo che conduce, dalla richiesta della pagina **Vincitori.php**, alla risposta HTTP contenente l'HTML inviato al browser:



1.4 Conclusioni

L'uso di *pagine attive* consente di fornire al client dei contenuti esterni alle pagine, prelevati da un file, un database, etc, nonché di personalizzare tali contenuti in base alle azioni e/o all'identità dell'utente.

D'altra parte, il codice esecutivo che produce i contenuti di una pagina è integrato all'interno del codice HTML che stabilisce la loro presentazione. In sostanza, *non c'è alcuna separazione tra la logica che produce i contenuti e quella che li deve presentare*.

ASP.NET Core consente di superare brillantemente questo problema.

2 Introduzione ad ASP.NET Core

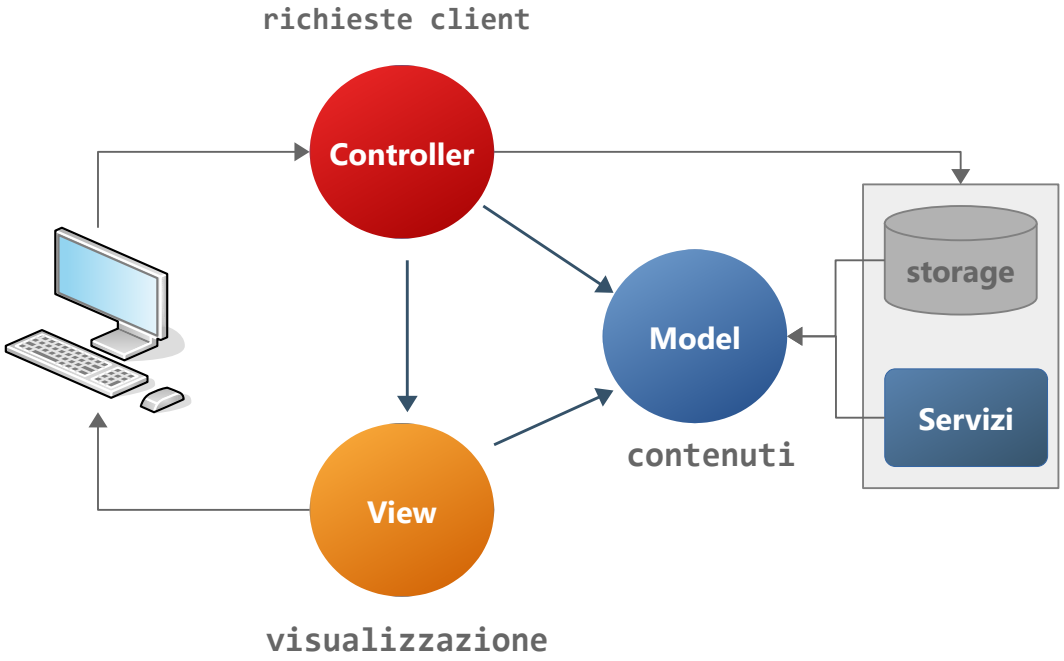
ASP.NET Core (***A**ctive **S**erver **P**ages over .**NET Core***) rappresenta un *framework cross-platform* per la realizzazione di applicazioni web. Il *framework* è basato su .NET Core, la versione multi piattaforma del .NET Framework.

ASP.NET consente di realizzare vari tipi di applicazione e di integrare varie tecnologie, alcune delle quali agiscono specificatamente sul client (e delle quali, dunque, non ci interesseremo). In questo tutorial mi limiterò a trattare la realizzazione di siti web mediante l'uso del *pattern* architetturale ***Model View Controller***.

2.1 Model View Controller (MVC)

Il *design pattern* MVC viene incontro all'esigenza, soprattutto nelle applicazioni distribuite, di isolare la logica che fornisce i contenuti da quella che li presenta. A questo scopo il cuore dell'applicazione è suddiviso in tre tipi di componenti:

- *model*: definiscono i contenuti. Considerando il programma **Library**, fanno parte del *model* le *entity class* `Book`, `Author`, `Genre`, etc.
- *view*: presentano i contenuti; rappresentano dunque l'interfaccia utente dell'applicazione.
- *controller*: processano le richieste dell'utente, ottengono il *model* che soddisfa tali richiese e lo passano alle *view*, che lo presenterà all'utente.



Di norma, nelle applicazioni realistiche, esistono altri componenti: *business logic*, servizi, accesso dati, configurazione, etc.

2.1.1 Relazione tra i componenti Model, View e Controller

Le frecce blu nello schema evidenziano le relazioni che intercorrono tra *model*, *view* e *controller*, e che caratterizzano questo *design pattern*.

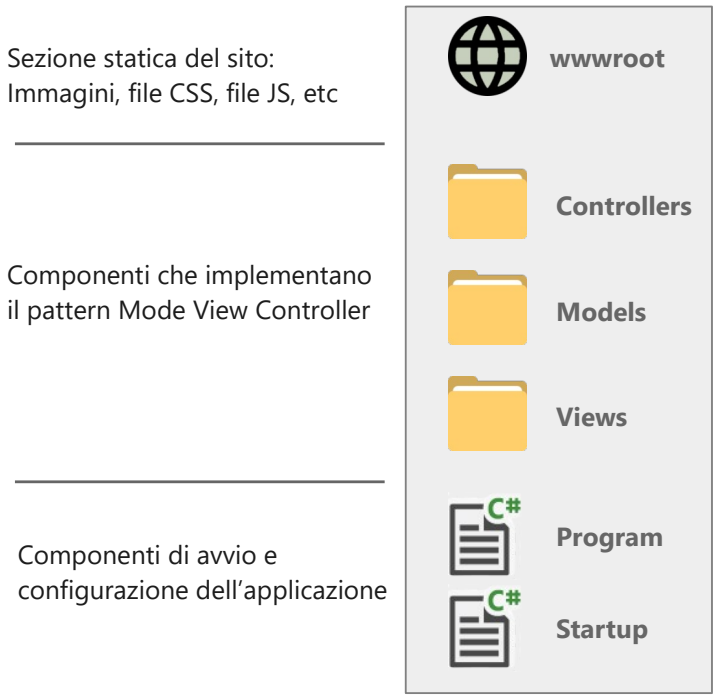
Il *model* è indipendente da *view* e *controller*: non dipende dunque dal tipo applicazione, dalla sua architettura o interfaccia utente. In generale, lo stesso *model* dovrebbe poter essere utilizzato in applicazioni desktop, *mobile*, web, etc.

Il *view* dipende dal *model*, poiché ha la funzione di presentarlo, ma non dipende dal *controller*. Semplicemente: il *view* riceve dei dati e li presenta all'utente.

Infine, il *controller* usa sia *view* che *model*: in risposta alle richieste dell'utente, ottiene il *model* da classi *data access*, servizi, etc, e lo passa al *view*.

2.2 Architettura di un'applicazione ASP.NET Core MVC

Lo schema sottostante riassume la struttura generale di un'applicazione ASP.NET Core MVC[3]:



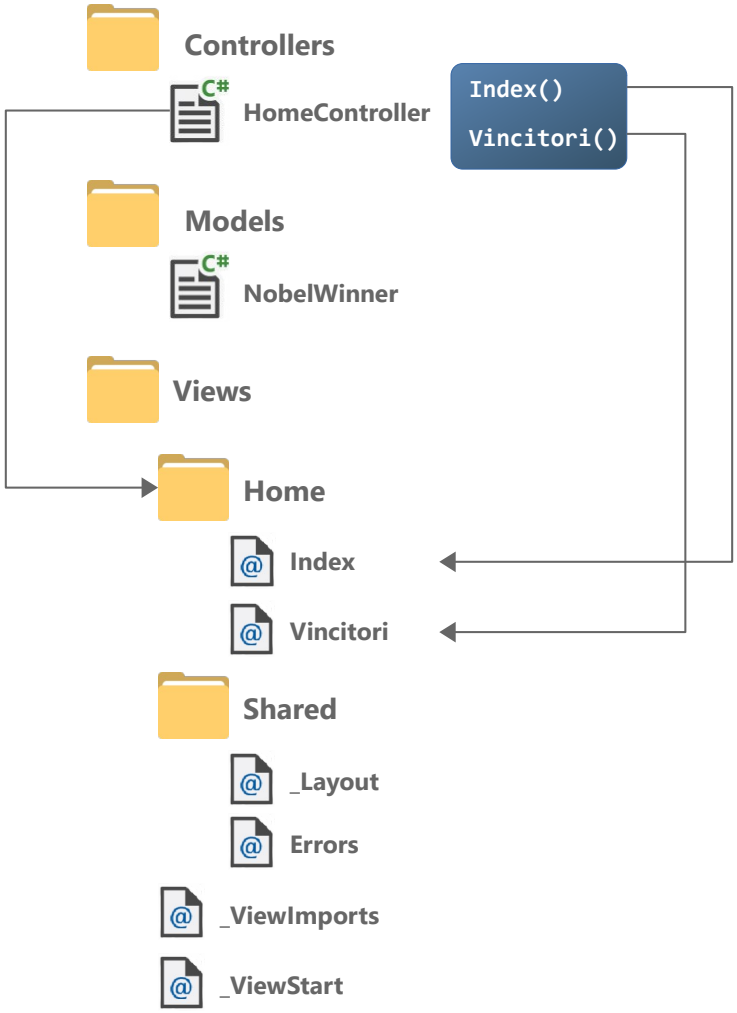
Gli elementi importanti, che caratterizzano questo tipo di applicazione, si trovano nelle tre cartelle centrali. Per quanto riguarda gli altri:

- la cartella **wwwroot** definisce la parte statica del sito, nella quale saranno collocate le immagini, i file CSS, le librerie javascript.
- **Program** produce l'avvio dell'applicazione. **Startup** consente di configurarla e di stabilire i servizi e componenti utilizzati, permettendo di aggiungerne di nuovi, come ad esempio Entity Framework.

3 Il progetto contiene anche altri file, che svolgono varie funzioni, tra le quali la configurazione del protocollo impiegato – HTTPS vs HTTP – e il numero di porta al quale il server starà in ascolto delle richieste del client.

2.2.1 Architettura del pattern MVC

Lo schema sottostante riassume la struttura dei componenti che, nell'applicazione, implementano il pattern MVC. Nota bene: la separazione di componenti non è soltanto logica, ma fisica, poiché risiedono in file separati; addirittura in cartelle separate.[4]



Nello schema ho supposto l'esistenza, come *model*, della classe `NobelWinner`.

Di default, l'applicazione definisce un solo *controller*, `HomeController`. Il *controller* definisce un metodo – *action* – per ogni *view* che deve visualizzare. (`Index()` e `Vincitori()`, nell'esempio)

Le *view*, file con estensione **cshtml**, sono collocate in sotto cartelle. Ad ogni *controller* corrisponde una cartella contenente le *view* che dovrà eseguire (nell'esempio, la cartella **Home** contiene le *view* **Index** e **Vincitori**).

La cartella **Shared** contiene le *view* condivise da tutta l'applicazione, tra queste **_Layout**, che stabilisce la struttura HTML comune a tutte le pagine del sito.

Infine, **_ViewImports** e **_ViewStarts** non hanno la funzione di presentare contenuti ma, come vedremo, di semplificare la realizzazione delle altre *view*.

4 In realtà la strutturazione nelle cartelle **Models**, **Controllers** e **Views** è l'impostazione di default e può essere modificata mediante un'opportuna configurazione.

2.3 Funzionamento di un'applicazione ASP.NET Core MVC

L'elaborazione di una richiesta da parte di un'applicazione ASP.NET Core MVC segue un percorso molto diverso da quello schematizzato nel paragrafo 1.3.

Le differenza comincia fin dalla codifica degli URL, che stabiliscono le risorse richieste dal client.

2.3.1 Codifica degli URL: route

Nei siti "normali" il client specifica nell'URL il nome della risorsa richiesta (file HTML, PHP, o altro), come mostra il frammento della pagina **Index.php**:

```
...
    <p><a href='Vincitori.php'>Elenco vincitori</a></p>
...
```

Il caricamento della pagina **Index.html** implica una richiesta analoga, con la differenza che, essendo la pagina di default, può essere omessa. Dunque:

`http://www.nobel.it`

equivale a:

`http://www.nobel.it/index.html`

In ASP.NET Core gli URL hanno una struttura diversa, definita *route*: indicano il *controller* e l'*action* che devono rispondere alla richiesta. In sintesi, dunque, *una route specifica un metodo da eseguire*.

Una *route* rispecchia la seguente struttura:

***/controller*=Home / *action*=Index / [id]**

dove **home** e **index** sono i valori predefiniti nel caso non vengano specificato *controller* e/o *action*.

Nel caso del sito Nobel, le due *route* per ottenere la home page e i vincitori sarebbero:

`/home/index` e `/home/vincitori`

Nel primo caso, la *route* può ridursi al carattere `/`, dato che *home* e *index* sono il *controller* e l'*action* predefiniti.

2.3.2 Route con parametro

Una *route* può specificare un parametro facoltativo (campo **id** nella struttura della *route*). Tratterò le *route* con parametri più avanti.

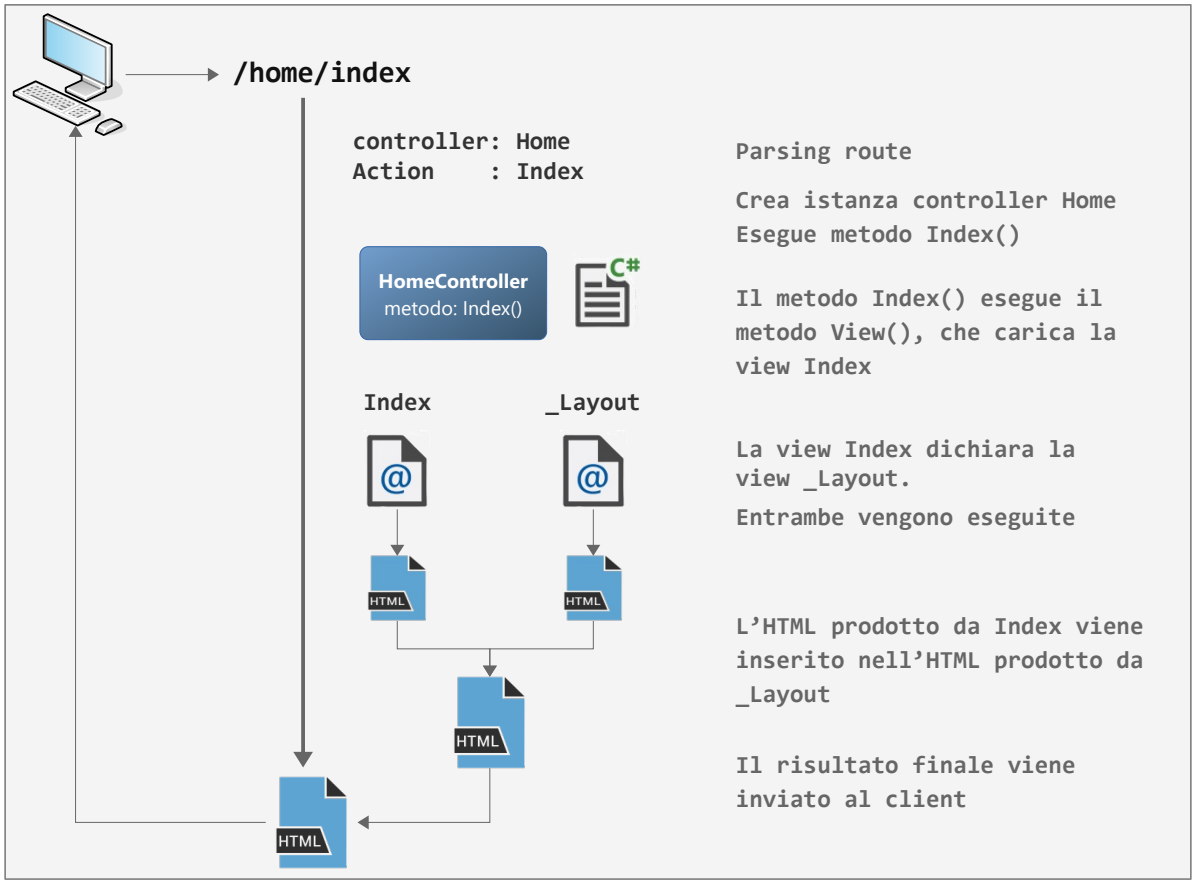
2.3.3 Elaborazione della richiesta

Una richiesta del client viene sempre processata da un metodo definito in un *controller*. La *route* della richiesta serve appunto a stabilire il metodo da eseguire.

È opportuno ribadirlo ancora una volta: il client non chiede una pagina o qualunque altro tipo di risorsa, statica o dinamica (come una pagina PHP); alla richiesta del client corrisponde sempre l'esecuzione di un metodo di un *controller*.

Naturalmente, il client "ignora" completamente i concetti di *controller* e *action*; a seguito della sua richiesta otterrà una pagina HTML o altro tipo di contenuto.

Lo schema seguente mostra a grandi linee come avvenga tutto ciò. Supponi che il client richieda la home page del sito, specificando dunque l'URL **http://www.nobel.it/home/index**:



Lo schema nasconde molti dettagli, ma mette in evidenza la questione principale: è il metodo `Index()` a stabilire il contenuto da restituire al client. Lo schema, quindi, mostra il comportamento predefinito:

1. Il metodo `Index()`, mediante l'esecuzione del metodo `View()`, carica la *view* **Index**.
2. La *view* **Index**, che definisce i contenuti da visualizzare, dichiara la *layout view* **_Layout**, la quale definisce la struttura generale delle pagine del sito.
3. Entrambe le *view* sono eseguite (allo stesso modo di una pagina PHP).
4. Il risultato prodotto dalla *view* **Index** viene inserito nell'HTML prodotto dalla *layout view* **_Layout**.
5. Il risultato finale, una pagina HTML, viene inviato in risposta al client.

2.4 Implementazione delle *view: C# + HTML = razor*

Le *view* sono le equivalenti delle *pagine attive*, possono contene codice HTML e codice esecutivo. ASP.NET Core fornisce una tecnologia di parsing, chiamata *razor*, che consente di integrare in molto semplice naturale codice C# e HTML.

Mediante il carattere `@` è possibile inserire un costrutto C# in qualsiasi punto della *view*. Seguono alcuni esempi (per un approfondimento vedi: Guida completa Razor [5])

2.4.1 Blocchi di codice

Ad esempio:

```
@{
    ViewData["Title"] = "Home";
    string[] elencoCittà = {"Roma", "Milano", "Firenze" };
   // da qui in poi "elencoCittà" è utilizzabile ovunque
   // (non può essere dichiarata un'altra variabile con lo stesso nome)
}
```

Nota bene: le variabili dichiarate in blocco sono accessibili ovunque nella *view*, a partire dalla linea successiva al blocco. (Si comportano come le variabili locali di un metodo)

2.4.2 Costrutti di controllo

`if`, `for`, `foreach`, `while`, etc. Il costrutto non deve terminare con il carattere `;`. Ad esempio:

```
@for (int i = 0; i < 10; i++)
{
    <p>HELLO!</p>
}
```

2.4.3 Espressioni implicite

Costrutti che producono un valore, comprese le chiamate a metodi. Ad esempio:

```
@DateTime.Now

@foreach (var città in elencoCittà)
{
    <p>@città.ToLower()</p>

    <a href="home/dettagli/@città">@città</a>
}
```

Nota bene: le *espressioni implicite* non devono contenere spazi, a meno che l'istruzione non abbia una terminazione non ambigua. (Ad esempio, la lista degli argomenti passata a un metodo può avere degli spazi, dato che la parentesi finale termina esplicitamente l'espressione.)

2.4.4 Espressioni esplicite

Sono rappresentate da una coppia di `(` `)` contenenti l'espressione. Ad esempio:

```
<p>L'altra settimana: @( DateTime.Now - TimeSpan.FromDays(7) )</p>
```

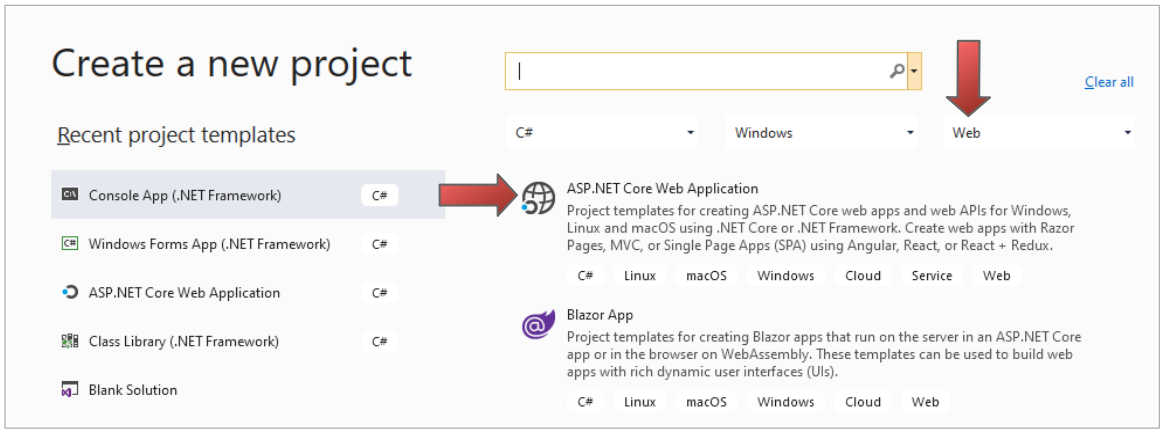
5 https://docs.microsoft.com/it-it/aspnet/core/mvc/views/razor?view=aspnetcore-3.1

3 Creare applicazioni ASP.NET Core MVC

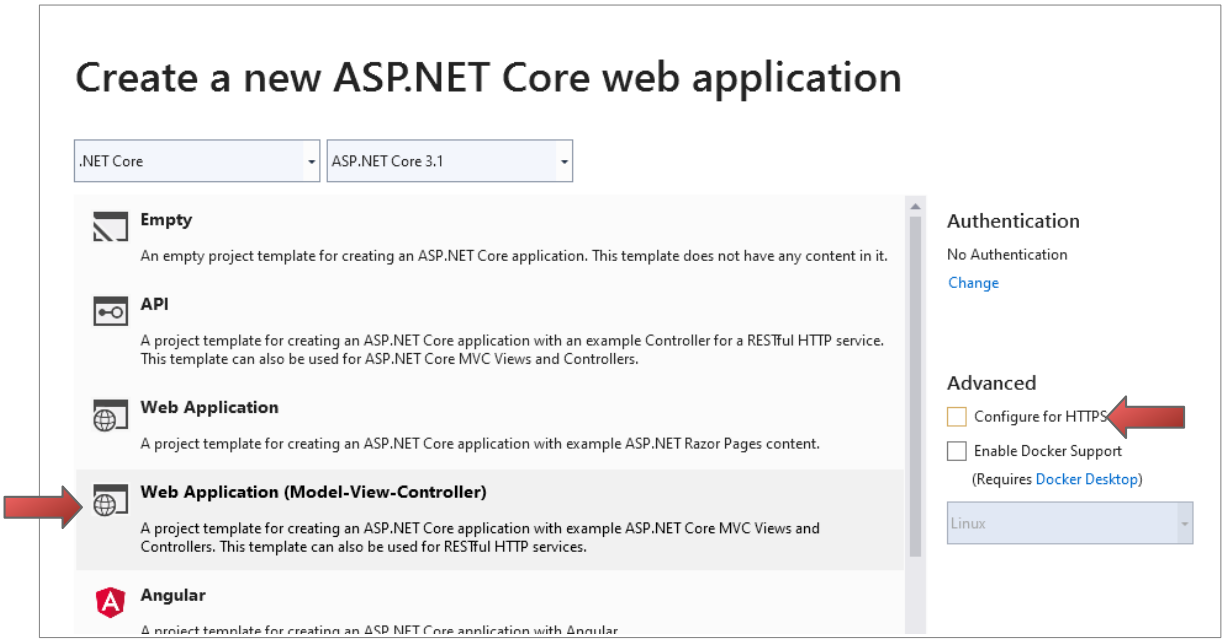
Di seguito mostrerò come realizzare un sito ASP.NET Core MVC che emuli il comportamento di quello realizzato in PHP. Nel mentre aggiungerò alcuni dettagli sul funzionamento di ASP.NET Core sui quali ho finora sorvolato.

3.1 Creazione di un progetto

Si crea un progetto selezionando il tipo di applicazione **ASP.NET Core Web Application**, nella la categoria **web**:



Successivamente si stabilisce il nome e la collocazione della *solution*. Infine si sceglie il *template* da utilizzare per la creazione del progetto (**Model-View-Controller**):



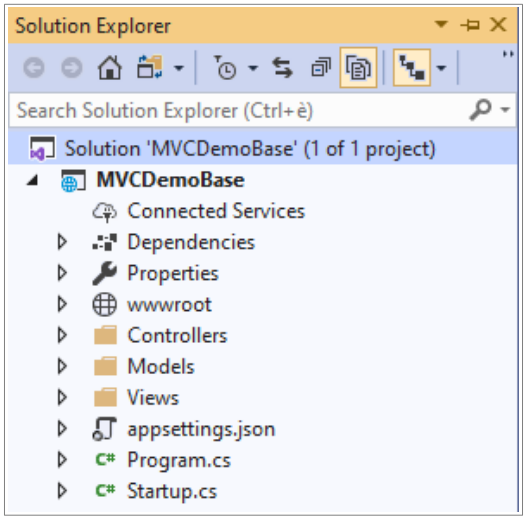
Per semplificare lo sviluppo è opportuno disabilitare la voce **Configure for HTTPS**.

3.2 Struttura generale del progetto

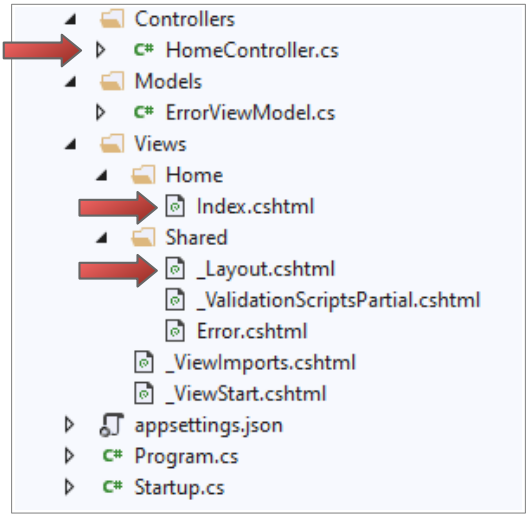
ASP.NET Core crea un'applicazione già funzionante, dotata di due *view* e una *layout view* che forniscono dei contenuti dimostrativi. Poiché intendo partire praticamente da zero, elimino quasi tutto il codice generato, lasciando le *view* **Index** e **_Layout** ridotte al minimo.

Dopo queste modifiche, *Solution Explorer* mostra la seguente struttura, della quale interessano soprattutto i tre elementi evidenziati:

Struttura generale



Struttura pattern MVC



3.3 Contenuto predefinito delle *view*

Dopo il mio intervento, le *view* **_Layout** e **Index** hanno il seguente contenuto:

_Layout

(ho omesso parte del tag `<head>`)

```
<!DOCTYPE html>
<html lang="en">
<head>
    ...
    <title>@ViewData["Title"]</title>
</head>
<body>
    <h2>NOBEL PRIZE</h2>
    <hr />
    @RenderBody()
</body>
</html>
```

Index

```
@{
    ViewData["Title"] = "Home";
}
<div>
   <a href='/home/vincitori'>Vincitori</a>
</div>
```

Le due linee evidenziano il legame tra **Index** e **_Layout**: il risultato prodotto da **Index** viene inserito nel risultato prodotto da **_Layout**, in corrispondenza del metodo `RenderBody()`.

Esistono altre due *view*, **_ViewStart** e **_ViewImports**, che non hanno la funzione di presentare contenuti, ma di definire una volta per tutte il codice da eseguire al caricamento di ogni *view*, evitando che sia necessario specificarlo ogni volta:

- **_ViewStart** stabilisce la *layout view* che presenta la struttura generale delle pagine del sito (nei siti reali possono esistere più *layout view*). Di default è **_Layout**.
- **_ViewImports** importa i *namespace* necessari ad ogni *view*.

_ViewStart

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

_ViewImports

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

3.4 Controller

Di default viene generato lo scheletro dell'`HomeController` (anche in questo caso ho semplificato il codice):

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

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

Il metodo `View()` restituisce la *view* corrispondente al nome del metodo `Index()`. Infatti, internamente il metodo `View()` esegue un codice equivalente a quello mostrato di seguito:

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

Una volta stabilito il nome della *view*, ASP.NET Core la localizza all'interno della struttura del sito e la esegue (usando il pattern *controller* **home** → cartella **Home**; action **Index** → *view* **Index**).

3.5 Esecuzione dell'applicazione

L'applicazione è configurata per essere eseguita sul web server **IIExpress**, all'indirizzo **localhost**. Il numero di porta è generato casualmente, ma può essere modificato aprendo le proprietà del progetto, selezionando la scheda **Debug** e modificando l'**App URL**. (Ho impostato la porta **5000**).

L'esecuzione del progetto implica:

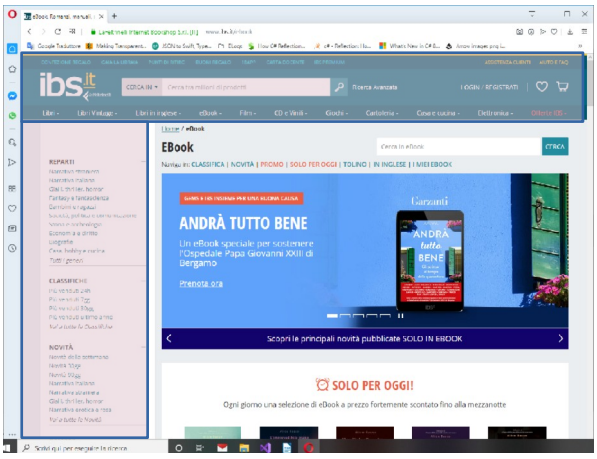
1. L'avvio del *web server* IIExpress (nell'area di notifica appare l'icona corrispondente), se non è già in esecuzione.
2. L'avvio dell'applicazione.
3. L'apertura del browser e la richiesta all'URL: **localhost:5000**, che corrisponde alla *route*: **localhost:5000/home/index**

3.6 *View*, *layout view* e pagine web

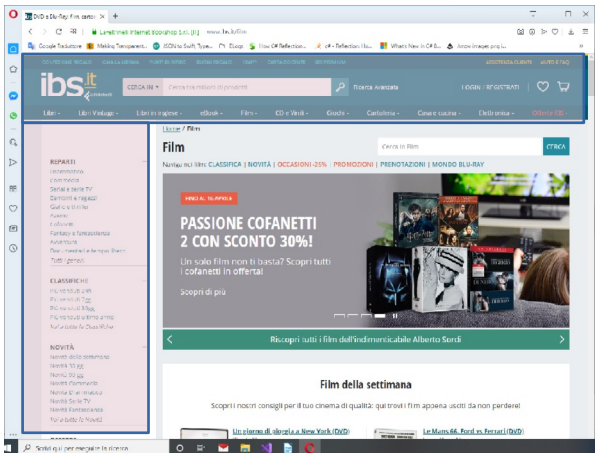
Il modello di presentazione dei contenuti adottato da ASP.NET Core viene incontro a un problema tipico nella realizzazione di siti web: evitare la duplicazione del codice comune nelle pagine del sito.

Ad esempio, lo *screen shot* sottostante mostra due pagine del sito IBS; le sezioni in alto e a sinistra sono identiche, cambia il contenuto mostrato nella sezione centrale:

IBS – Contenuto EBook



IBS – Contenuto Film

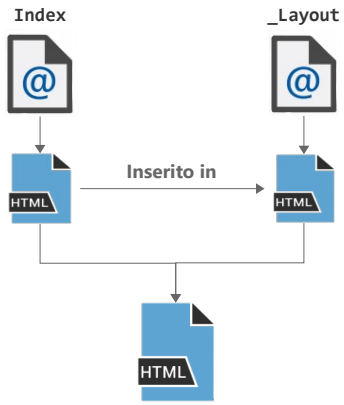


Nella maggior parte dei siti, infatti, le pagine condividono una struttura comune, dei contenuti comuni e, infine, l'uso di risorse comuni, come fogli di stile e librerie javascript.

A questo scopo, tutti i *framework* per lo sviluppo di siti web implementano il concetto di *master page*, o *layout page*, e cioè l'idea di collocare la struttura comune in un componente speciale, in modo che le pagine del sito possano limitarsi a presentare i contenuti specifici (Ebook e Film, nello *screen shot*).

In ASP.NET Core questa soluzione è implementata mediante l'uso delle *layout view*, e cioè *view* che definiscono struttura, contenuti e risorse comuni alle pagine del sito.

Durante l'elaborazione di una richiesta, l'HTML della *view* che presenta i contenuti, caricata dal metodo *action* del *controller*, viene inserito nell'HTML prodotto dalla *layout view*.



Nell'esempio considerato, **Index** e **_Layout** producono il seguente risultato; a sinistra, l'HTML in grigio chiaro è prodotto da **_Layout**, quello in grigio scuro da **Index**:

HTML inviata al browser

```
<!DOCTYPE html>
<html>
<body>
    <h2>NOBEL PRIZE</h2>
    <hr />
    <div>
     <a href='/home/vincitori'>vincitori</a>
    </div>
</body>
</html>
```

Output prodotto dal browser

3.7 Impostare il titolo della pagina: uso di ViewData

Il titolo delle pagine web viene visualizzato sulla barra del titolo del browser e, in HTML, viene impostato mediante il tag **title**, contenuto a sua volta nel tag **head**.

In ASP.NET Core ciò pone un problema, poiché i contenuti HTML inviati al browser sono definiti nelle singole *view*, ma il tag **head** è specificato una volta per tutte nella *layout view* **_Layout**.

La soluzione è quella di memorizzare il titolo nel dizionario `ViewData` all'interno delle singole *view*. **_Layout** utilizzerà il valore nel tag **title** della pagina:

Index

```
@{ViewData["Title"] = "Home";}
<div>
    HOME
</div>
```

_Layout

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

Come vedremo, `ViewData` può essere impiegato anche per altri scopi.

3.8 Visualizzazione dei vincitori: passare il *model* alla *view*

Nella versione PHP del sito, la pagina **Vincitori.php** carica i nomi dei vincitori dal disco e li presenta all'utente. In ASP.NET Core MVC è il *controller* che risponde alle richieste del client, dunque è il *controller* che deve ottenere i dati e passarli alla *view* che dovrà presentarli.

3.8.1 Caricare i nomi dei vincitori

Dopo aver aggiunto una cartella **Data** al progetto e avervi collocato il file **Vincitori.txt**, aggiungo alla cartella **Models** la classe che rappresenta il singolo vincitore:

```
public class NobelWinner
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string FullName => LastName + ", " + FirstName;
}
```

In `HomeController` colloco il metodo che carica i dati (si tratta di una soluzione temporanea):

```
public class HomeController : Controller
{

   ...
   List<NobelWinner> LoadNobelWinners()
   {
      var winners = new List<NobelWinner>();
      foreach (var riga in System.IO.File.ReadAllLines("Data/Vincitori.txt"))
      {
          if (riga.StartsWith("#"))
              continue;

          var items = riga.Split(';');

          var nw = new NobelWinner
          {
              LastName = items[0].Trim(),
              FirstName = items[1].Trim(),
          };
          winners.Add(nw);
      }
      return winners;
  }
}
```

3.8.2 Passaggio dei contenuti alla view Vincitori: uso di ViewData

Occorre un metodo, `Vincitori()`, che gestisca la richiesta **home/vincitori**. Il metodo ha responsabilità di caricare la *view* omonima, passandogli i contenuti da visualizzare:

Esistono due tecniche; una prevede di collocare i contenuti nel dizionario `ViewData`:

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

    public IActionResult Vincitori()
    {
        ViewData["vincitori"] = LoadNobelWinners();
        return View();
    }
    ...
}
```

3.8.3 Visualizzazione dei vincitori

Occorre creare la *view* **Vincitori**; Visual Studio fornisce la voce di menù necessaria, ma risulta comodo eseguire un semplice copia incolla di un'altra *view* (**Index** in questo caso).

Nella *view* **Vincitori**, prima si accede ai dati memorizzati in `ViewData`, poi si scorrono in modo del tutto simile a quanto faremmo in PHP.

```
@{
    ViewData["Title"] = "Vincitori";
    var winners = ViewData["vincitori"] as List<NobelWinner>;
}

<div>
    @foreach (var wi in winners)
    {
        <p>@wi.FullName</p>
    }
</div>
```

3.9 Usare il *model*: passare alla *view* dati *tipizzati*

L'uso di `ViewData` per passare i contenuti alla *view* è funzionale, ma non ottimale. ASP.NET Core fornisce un modo migliore, nel quale la *view* definisce il *model* utilizzato come se dichiarasse una variabile. Si usa la direttiva `@model`:

```
@model <tipo variabile>
```

Nella *view* **Vincitori** il *model* è rappresentato da una lista di `NobelWinner`:

```
@{
    ViewData["Title"] = "Vincitori";
}
@model List<NobelWinner>  // variabile implicita di nome Model

<div>

    @foreach (var nw in Model)
    {
        <p>@nw.FullName</p>
    }
</div>
```

Nota bene: è come se fosse dichiarata una variabile di nome `Model` di tipo `List<NobelWinner>`.

3.9.1 Passare il model

Nel *controller* il passaggio dei dati è estremamente semplice. Il metodo `View()` ha una versione che accetta come argomento il *model* da passare alla *view*:

```
public IActionResult Vincitori()
{
    var winners = LoadNobelWinners();
    return View(winners);

}
```

3.10 Passare dati in una richiesta: databinding

Negli esempi considerati finora, sia in PHP che in ASP.NET Core, le richieste del client non contenevano informazioni nell'URL se non la risorsa da visualizzare. In molti casi, però, l'URL definisce uno o più valori necessari a qualificare la richiesta.

Ad esempio, un sito di prodotti elettronici potrebbe visualizzare nella home un elenco di articoli in offerta; cliccando su uno degli articoli si apre una pagina che ne mostra i dettagli. In questo caso la richiesta dovrà specificare sia la pagina che visualizza i dettagli, sia il codice dell'articolo da visualizzare.

In ASP.NET Core si possono usare tre tecniche:

1. Passare l'informazione nella *route;* modalità che consente di passare un solo parametro.
2. Usare una *query string*, che consente di passare più parametri.
3. Usare entrambe le modalità nello stesso URL.

Qualunque sia la modalità utilizzata, ASP.NET Core applica un meccanismo molto potente e versatile, il ***databinding***, che associa automaticamente il valore/i presenti nell'URL al parametro/i del metodo che dovrà elaborare la richiesta.

3.11 Passare un parametro nella *route*

La struttura di una *route* prevede la possibilità di specificare, oltre a *controller* e *action*, un valore aggiuntivo.

/controller=Home / *action*=Index / **[id]**

Nel sito Nobel è possibile utilizzare questa possibilità per ottenere la visualizzazione "dettagli" di un premo Nobel selezionato dall'elenco dei vincitori. Per farlo occorre innanzitutto modificare la *view* **Vincitori**, in modo che visualizzi l'elenco mediante degli *hyperlink*:

```
@{
    ViewData["Title"] = "Vincitori";
}
@model List<NobelWinner>

<div>
    @foreach (var nw in Model)
    {
        <p><a href="/home/vincitore/@nw.FullName">@nw.FullName</a> </p>
    }
</div>
```

Ad esempio, se l'utente clicca sul primo link, il client invia la richiesta:

/home/vincitore/`Einstein, Albert`

dove quello evidenziato è il valore associato al campo **id** della route.

3.11.1 Gestire una richiesta con parametro nella route

Il metodo *action* che gestisce la richiesta deve semplicemente specificare un parametro stringa di nome `id`:

```
public IActionResult Vincitore(string id)  //id -> FullName del vincitore scelto
{
    ...
}
```

Nota bene: essendo il valore specificato nella *route*, il parametro deve chiamarsi `id`. (Non esistono distinzioni tra maiuscole/minuscole nel nome del parametro.)

3.12 Usare una *query string*

Una *query string* è una stringa che contiene un elenco di coppie *chiave=valore*, separate dal carattere `&`. La stringa è preceduta dal carattere `?` e segue l'URL della richiesta.

La seguente *query string* specifica un solo valore, di chiave *fullname*:

`http://www.Nobel.it/home/vincitore?fullname=Einstein, Albert`

Ecco la sua applicazione nella *view* **Vincitori**:

```
...
<div>
    @foreach (var nw in Model)
    {
        <p><a href="/home/vincitore?fullname=@nw.FullName">@nw.FullName</a> </p>
    }
<div>
```

3.12.1 Gestire una richiesta con query string

Il metodo *action* che gestisce la richiesta deve specificare un parametro stringa con lo stesso nome della chiave usata nella *query string*:

```
public IActionResult Vincitore(string fullName)
{
    ...
}
```

Il processo di *databinding* si occuperà di trovare la corrispondenza tra la chiave della *query string* e il parametro del metodo.

(Ancora una volta: non esiste distinzione tra maiuscole/minuscole.)

3.12.2 Gestire due parametri nella query string

L'uso di una *query string* è utile soprattutto quando occorre passare due o più parametri:

```
<div>
    @foreach (var nw in Model)
    {
        <p><a href="/home/vincitore?firstname=@nw.FirstName&lastname=@nw.LastName">
           @nw.FullName</a> </p>
    }
<div>
```

Il metodo *action* dovrà specificare due parametri con lo stesso nome delle chiavi:

```
public IActionResult Vincitore(string firstName, string lastName)
{
    ...
}
```

Nota bene: l'ordine di dichiarazione dei parametri non ha importanza, il processo di *databinding* basa la propria corrispondenza sui nomi.

3.13 Definire gli *hyperlink* mediante i *tag helper*

I *tag helper* sono attributi *server-side*, poiché vengono processati dal server; sono utili per personalizzare la definizione dei tag HTML. Sono usati in molti scenari; qui li utilizzo per definire l'*hyperlink* della *view* **Vincitori** e semplificare il passaggio dei parametri *firstname* e *lastname*.

Ricordo che l'URL usato è il seguente:

/home/vincitore?firstname=Albert&lastname=Einstein

il quale specifica: *controller*, *action*, *firstname* e *lastname*. Ebbene, è possibile definire questi valori separatamente e lasciare ad ASP.NET Core il compito di costruire l'URL:

```
<div>
   @foreach (var nw in Model)
   {
       <p><a asp-controller ="home"
             asp-action="vincitore"
             asp-route-firstname="@nw.FirstName"
             asp-route-lastname="@nw.LastName">@nw.FullName</a></p>
   }
</div>
```

Nota bene: mediante i *tag helper* `asp-route-chiave` è possibile stabilire il nome della chiave da associare al valore.

Questa tecnica è valida anche se vogliamo passare un parametro nella *route*. Dunque, l'*hyperlink*:

```
<a href="/home/vincitore/@nw.FullName">@nw.FullName</a>
```

può essere scritto:

```
<a asp-controller ="home" asp-action="vincitore"
   asp-route-id="@nw.FullName">="@nw.FullName</a>
```

Sarà ASP.NET Core, nell'eseguire la *view*, a tradurre la seconda forma nella prima.

4 Un esempio completo di applicazione: MotoGP

Di seguito svilupperò una nuova applicazione allo scopo di mostrare altre funzioni delle applicazioni web e la loro implementazione mediante di ASP.NET Core.

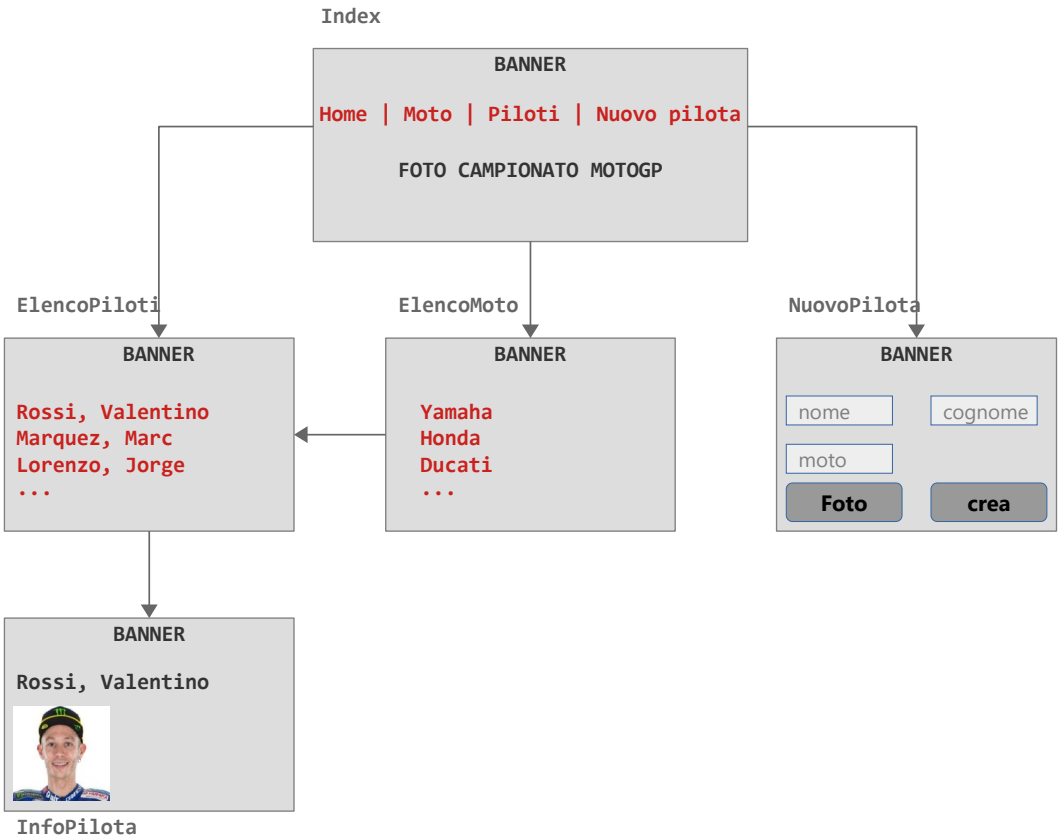
4.1 Descrizione generale dell'applicazione

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.

Segue lo schema delle pagine del sito, ad ognuna delle quali corrisponde una *view*. In tutte le pagina comparirà un menù per accedere alle funzioni principali.

Le frecce nello schema indicano la navigazione tra le pagine. Ad esempio, selezionando una moto in **ElencoMoto**, il sito dovrà visualizzare i piloti che corrono con quella moto.



4.1.1 View e controller

Riepilogando, intendo definire le seguenti *view:*

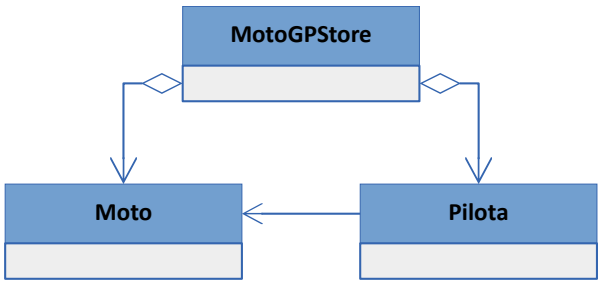
- **Index**: è la home page e mostra semplicemente una foto e definisce il menù.
- **ElencoMoto**: visualizza l'elenco delle marche iscritte al mondiale; consente di cliccare su una moto e ottenere l'elenco dei piloti che corrono con essa. (*view* **ElencoPiloti**)
- **ElencoPiloti**: visualizza l'elenco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

Tutte le richieste saranno gestire dall'`HomeController`.

4.2 Model e *data access*

Rispetto alla soluzione adottata per il sito Nobel, intendo usare un approccio "più professionale" nella gestione dei contenuti.

Il *model* è rappresentato da due classi, `Pilota` e `Moto`, collocate entrambe nella cartella **Models**. L'accesso ai dati è fornito dalla classe `MotoGPStore`. Questa genera in memoria le liste dei piloti e delle moto; sono entrambe liste statiche, dunque il loro ciclo di vita coincide con quello dell'applicazione.



Segue il codice delle classi:

```
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 MotoGPStore
{
    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 {...});
        ...
    }
}
```

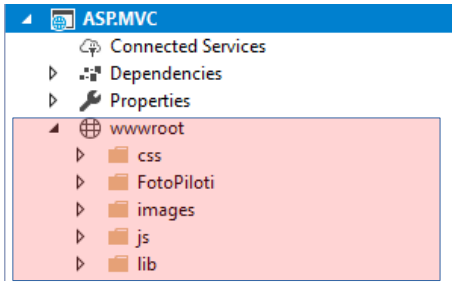
Per semplicità, anche `MotoGPStore` è collocata nella cartella **Models**.

4.3 Layout view

La *layout view* **_Layout** definisce una foto che funge banner e il menù, poiché entrambi gli elemento devono comparire in tutte le pagine:

```
<!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>
            <li><a asp-controller="home" asp-action="NuovoPilota">Nuovo pilota</a></li>
        </ul>
    </menu>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

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



4.4 Home page

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

Index

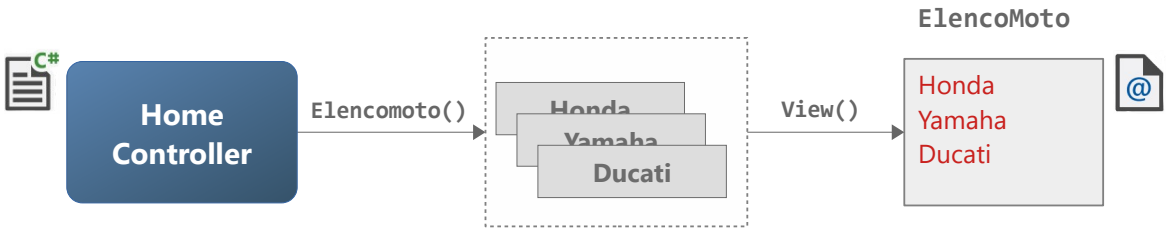
```
@{ViewData["Title"] = "Home";}

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

Index()

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

4.5 Visualizzare l'elenco delle moto - ElencoMoto

Schematizziamo questa funzione secondo il *pattern* MVC:



- La *view* **ElencoMoto** ha la funzione di visualizzare le moto iscritte al campionato mediante un elenco di *hyperlink*.
- Il *model* utilizzato dalla *view* è pertanto rappresentato da una lista di `Moto`.
- Il *controller* ottiene questo elenco da `MotoGPStore` e lo passa alla *view*.

4.5.1 Implementazione della view

Prima di implementare la *view*, intendo mostrare l'HTML che dovrà essere prodotto:

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

Ogni link referenzia l'*action* `ElencoPilotiMoto()` e specifica nella *route* l'id della moto visualizzata.

Segue l'implementazione:

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

@model IEnumerable<Moto>


<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="ElencoPiloti"
              asp-route-id="@m.MotoId">@m.Nome</a>
        </p>
    }
</div>
```

Nota bene: alternativamente avrei potuto scrivere:

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

4.5.2 Controller: ottenere il model e passarlo alla view

Il compito del *controller* è quello di usare il *repository* per ottenere i dati richiesti e quindi passarli alla *view*:

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();
    ...
    public IActionResult ElencoMoto()
    {
        return View(repo.GetMoto());  // passa l'elenco delle moto alla view
    }
}
```

4.6 Visualizzazione dell'elenco dei piloti - ElencoPiloti

La visualizzazione dell'elenco dei piloti rappresenta un esempio di come il pattern MVC faciliti la separazione tra elaborazione delle richieste e presentazione dei dati. L'elenco dei piloti può essere visualizzato in due circostanze, che producono elenchi diversi:

- Cliccando sulla voce **Piloti** del menù: elenco completo dei piloti.
- Nella *view* **ElencoMoto** cliccando sul nome di una moto: elenco dei piloti che corrono con la moto selezionata.

Ma dal punto di vista della *view* **ElencoPiloti**, ciò non ha importanza, poiché si limita a ricevere un elenco e a visualizzarlo.

```
@{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*, in modo che l'utente possa selezionarlo per accedere alla pagina di informazioni sul pilota.

4.7 Caricamento dei piloti: *databinding* con parametro facoltativo

Poiché esistono due richieste distinte che devono produrre un elenco di piloti:

`home/ElencoPiloti>` (tutti i piloti)

e

`home/ElencoPiloti/<id>` (piloti della moto <id>)

occorre stabilire come gestirle sul *controller*. La prima idea potrebbe essere quella di definire due metodi con lo stesso nome, uno dei quali dichiara un parametro di nome `id`:

```
public class HomeController : Controller
{
    ...
    //Home/ElencoPiloti
    public IActionResult ElencoPiloti()
    {
        ...
    }
```

```
    //Home/ElencoPiloti/<id>
    public IActionResult ElencoPiloti(int id)
    {
        ...
    }
}
```

Purtroppo questa modalità non funziona: essendo il parametro **id** della *route* facoltativo, il *databinding* non fa distinzione tra i due metodi, i quali sarebbero entrambi eleggibili a rispondere ad ognuna delle due richieste.

4.7.1 Definire un metodo action con parametro id nullable

La soluzione è quella di definire un solo metodo, che dichiari un parametro `id` nullabile.

```
public class HomeController : Controller
{
    MotoGPRepository repo = new MotoGPRepository();
    ...
    public IActionResult ElencoPiloti(int? id)
    {
        if (id == null)
            return View(repo.GetPiloti());       //Home/ElencoPiloti/
        return View(repo.PilotiMoto(id.Value));  //Home/ElencoPiloti/<id>
    }
}
```

Nel processare la richiesta **home/ElencoPiloti** il meccanismo *databinding* assegnerà `null` al parametro id (non essendo specificato alcun valore nella richiesta).

4.8 Informazioni sul pilota - InfoPilota

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

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

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

Nota bene: in questa *view* imposto innanzitutto due variabili stringa contenenti le statistiche sul pilota, che utilizzerò successivamente nel codice HTML. .

5 Inserimento dei dati

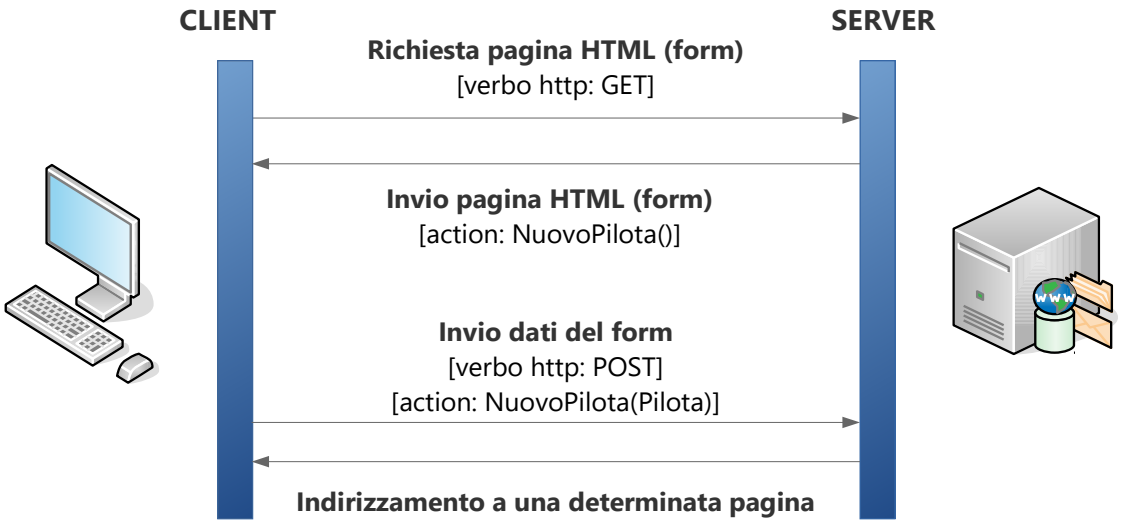
La creazione di un nuovo pilota richiede di implementare:

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

5.1 Gestione di un form HTML

IL funzionamento di un form HTML si suddivide 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**.

5.1.1 Implementazione del form HTML in ASP.NET

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>
```

Nota bene: anche questa *view* dichiara il tipo del *model;* ciò, unito all'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**.

5.1.2 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.
- ASP.NET è in grado di "bindare" i dati ricevuti, assegnandoli alle corrispondenti proprietà del parametro `pilota`.
- Dopo aver inserito il pilota nel *repository*, il metodo reindirizza l'utente alla *view* **ElencoPiloti** utilizzando `RedirectToAction()`.

5.2 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 il *record* 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** permettere di generare dinamicamente il seguente codice HTML:

```
<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>
```

5.3 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.

5.3.1 Salvare il file su disco

Si suppone di voler salvare il file in una sotto cartella del sito. Occorre innanzitutto conoscere il percorso della cartella **wwwroot**; un modo è quello di accedere all'oggetto *host environment*, 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 IWebHostEnvironment

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

```
    public HomeController(IWebHostEnvironment 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*.

6 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.

6.1 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");
}
```

6.2 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 sulla 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 da visualizzare automaticamente tutti gli errori.

6.2.1 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 contengono dati corretti");
        return View(pilota);
    }
    ...
}
```

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

6.2.2 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
{
```

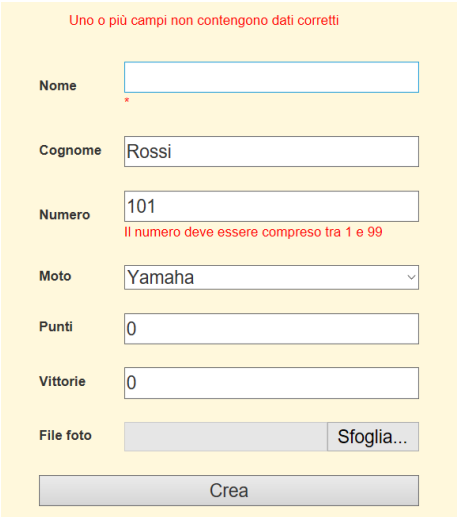
```
    ...
    [Required(ErrorMessage="*")]
    public string Nome { get; set; }

    [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; }
    ...
}
```

6.3 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:



7 Usare Entity Framework

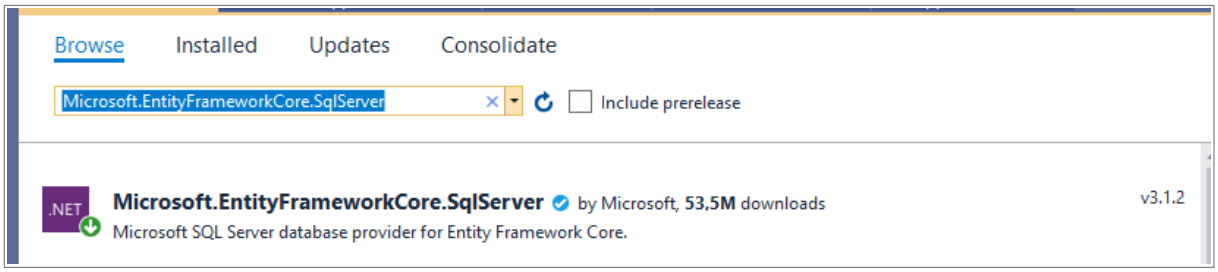
Diversamente dagli altri tipi di applicazione, ASP.NET Core MVC non è compatibile con Entity Framework versione 6; occorre usare **Entity Framework Core**. Questo presenta alcune differenze rispetto a EF 6:

- Implementa diversamente alcune funzionalità, come ad esempio il *lazy loading*.
- Implementa un sistema di configurazione dell'*entity model* leggermente diverso.
- Implementa un diverso meccanismo per gestire la stringa di connessione.
- 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 di base appresi su EF 6 restano validi anche per EF Core.

7.1 Aggiungere EF a un progetto ASP.NET Core

Dalla versione ASP.NET Core 3.0, il pacchetto deve essere aggiunto all'applicazione (come è stato fatto con EF 6.0). Si esegue il Nuget Package Manager e, nella scheda **Browse**, si seleziona il provider corrispondente al tipo di DBMS. Nello *screen shot* seguente mostro il pacchetto da installare per interfacciarsi con un database SQL Server:



7.2 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()`:

```
using Microsoft.EntityFrameworkCore;
...

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; }
}
```

7.2.1 Configurazione del model

La configurazione del model richiede la mappatura delle *entità* mediante attributi, poiché la convenzione sui nomi, anglosassone, 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; }

}
```

7.2.2 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* (usata anche in EF 6), ed è necessaria, poiché, nell'ambito delle applicazioni web, il *lazy loading* è normalmente inutilizzabile.

7.3 Definire un ViewModel

La nuova versione di `Pilota` incorpora l'immagine corrispondente, adesso memorizzata nel database e non più in un file separato[6]. 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 sì i dati del *model*, ma è progettato allo scopo di favorire l'implementazione della *view*.

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

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

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

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

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

6 Qui non discuto sull'opportunità di memorizzare l'immagine nel database. In realtà si tratta di una scelta errata, che diminuisce le performance.

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

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

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

@model IEnumerable<PilotaInfo>

<div>

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

7.4 Visualizzare le immagini memorizzate nel database

La *view* **InfoPilota** visualizza la foto del pilota mediante un tag **img** che referenzia il file contenente l'immagine, collocato nella cartella **FotoPiloti**; il nome del file è memorizzato nella proprietà `FileFoto` di `Pilota`. 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">
            <img src="~/FotoPiloti/@Model.FileFoto" />
            <img src="data:image;base64,@System.Convert.ToBase64String(Model.Foto)" />
        </th>
    ...
</table>
```

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

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

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

7.4.1 Implementare un metodo action che restituisce l'immagine

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

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

Nota bene: il metodo `File()` restituisce un file incorporato in un oggetto `FileContentResult`.

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">
            <img src="/Home/GetFotoPilota/@Model.PilotaId" />

        </th>
```

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

7.5 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, mediante il Nuget Packege Manager, oppure la **Package Manager Console**, mediante il comando:

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

> ### InMemory database e modello relazionale
>
> Il provider **InMemory** non implementa un database relazionale e dunque 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**.

7.6 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.)

7.7 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 di può implementare un metodo che inserisca i dati; questo sarà eseguito in `Configure()` della classe `Startup`:

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

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

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

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

8 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.

8.1 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 class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

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
    }
}
```

8.2 Impostare i servizi utilizzati

La modularità di ASP.NET Core è dimostrata dal fatto che, di default, l'applicazione non fornisce alcun servizio[7]; questi devono essere definiti nel metodo `ConfigureServices()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

8.3 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();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

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

    app.UseRouting();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>   // stabilisce la route predefinita utilizzata da MVC
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{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. (Errore: sorgente del riferimento non trovata)

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

8.4 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:

```
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
{
    IHostEnvironment hostEnv;
    MotoGPContext db = new MotoGPContext();
    MotoGPContext db;

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

8.4.1 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();
}
```

8.4.2 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.

9 Autenticazione dell'utente

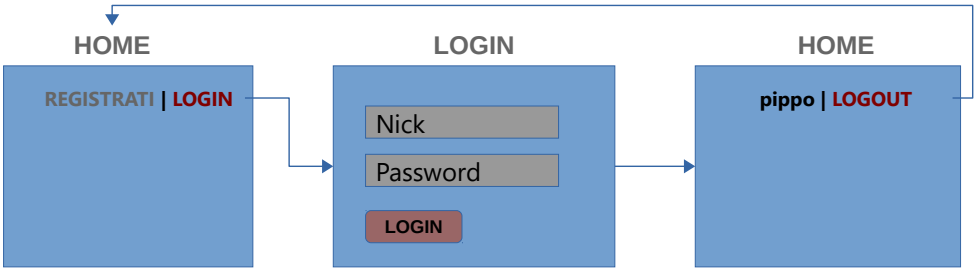
Il termine *autenticazione* disegna il processo con il quale un'applicazione stabilisce l'identità dell'utente, allo scopo di fornire servizi specifici e/o autorizzare (o negare) l'accesso a determinate risorse. Tale processo si basa su alcune premesse:

- L'utente deve essere stato precedentemente registrato.[8]
- 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 dovrebbe fornire la possibilità all'utente di ritornare anonimo (*logout*)

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

9.1 Implementazione dei processi di login/logout

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



In sintesti: l'applicazione 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*.

9.1.1 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
{
```

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

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

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

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

1. Validare i dati inseriti.
2. Verificare se le credenziali corrispondono 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)
{
  1 string scheme = CookieAuthenticationDefaults.AuthenticationScheme;

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

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

Il metodo `SignIn()`:

1 Stabilisce il tipo di autenticazione: basata su *cookie*.

2 Crea un'*identità* per l'utente da autenticare. A questa identità associa il nome completo dell'utente, memorizzato nel record `user`.

3 Esegue 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.)

9.1.2 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 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*.

9.2 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:

- `Contex`, pur con nome diverso, referenzia 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:

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

9.3 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 UseEndPoints()
        app.UseAuthentication();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
}
```

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