ASP.NET MVC

INTRODUZIONE ALLE APPLICAZIONI WEB SU APS.NET

Anno 2022/2023

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*.

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*, in esecuzione sulle macchine degli utenti finali.[1] I client comunicano con il processo server utilizzando il protocollo HTTP.

Il processo server è gestito da un altro processo chiamato *web server*[2], il quale 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 e non dei client.

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.

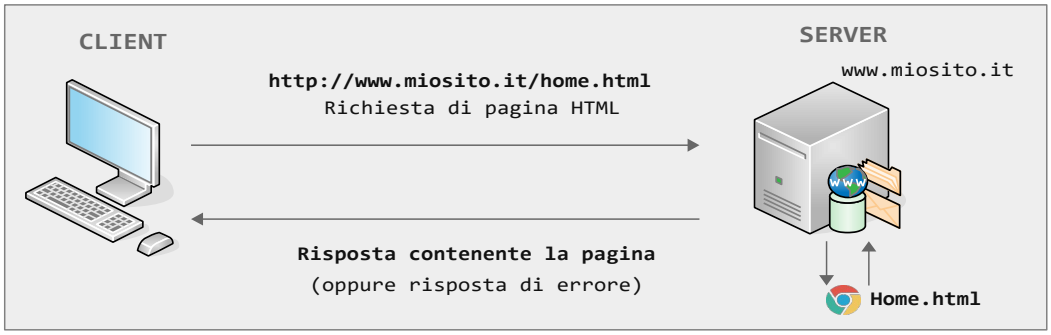
1.2.1 Siti web statici

Un *sito web statico* fornisce un servizio di natura documentale. Il *browser* richiede un documento mediante un URL (***U**niform **R**esouce **L**ocator*), il server lo carica da disco e lo invia al client. Nella maggior parte dei casi i documenti richiesti sono pagine HTML, le quali rendono possibile la navigazione tra i contenuti del sito attraverso 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 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 accadere che un processo server sia a sua volta un client di un altro processo server.

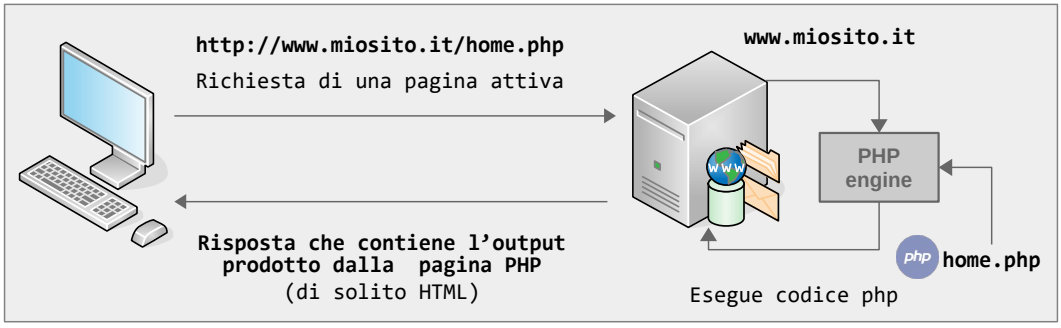
2 Spesso si usa il termine *web server* anche per indicare la macchina fisica che ospita l'applicazione.



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 delle cosiddette *active server pages*: file che integrano al proprio interno sia codice HTML che istruzioni in un certo linguaggio di programmazione.

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 esegue il codice PHP. Il risultato di questa esecuzione, di norma HTML, viene inviato al client.

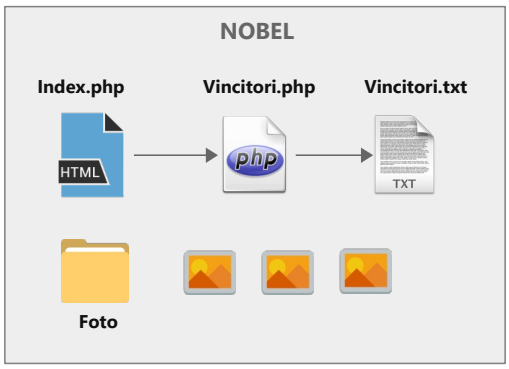


È importante sottolineare che l'esecuzione della pagina – **home.php**, nell'esempio – avviene interamente nel server. Il client riceve soltanto il risultato finale.

1.3 Un semplice sito web di esempio

Di seguito propongo un semplice sito composto da due pagine, una HTML, l'altra PHP, con l'obiettivo di mostrare un sito con contenuti statici e dinamici e usarlo come termine di paragone 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 del premio Nobel. Le informazioni sui vincitori sono memorizzate in un 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 su di essa alcuna elaborazione

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

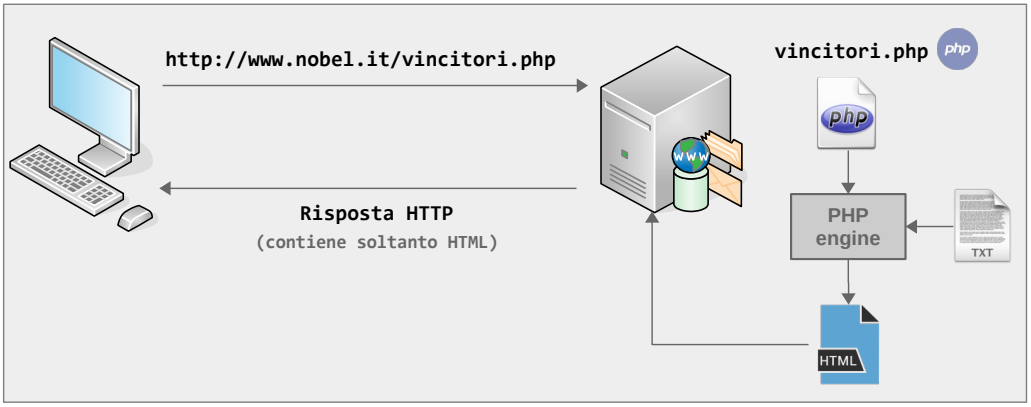
Quando l'utente clicca il *link*, il client richiede al server la pagina **Vincitori.php**:

```
<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 esegue il codice PHP, il cui output viene integrato con l'HTML della pagina. L'output finale, puro HTML, 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, a partire dalla richiesta della pagina **Vincitori.php**, conduce alla risposta HTTP contenente l'HTML inviato al browser:



1.4 Conclusioni

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

In queste pagine il codice esecutivo è integrato, "mescolato", all'interno del codice HTML che stabilisce la presentazione dei contenuti. 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 questo problema.

2 Introduzione ad ASP.NET Core

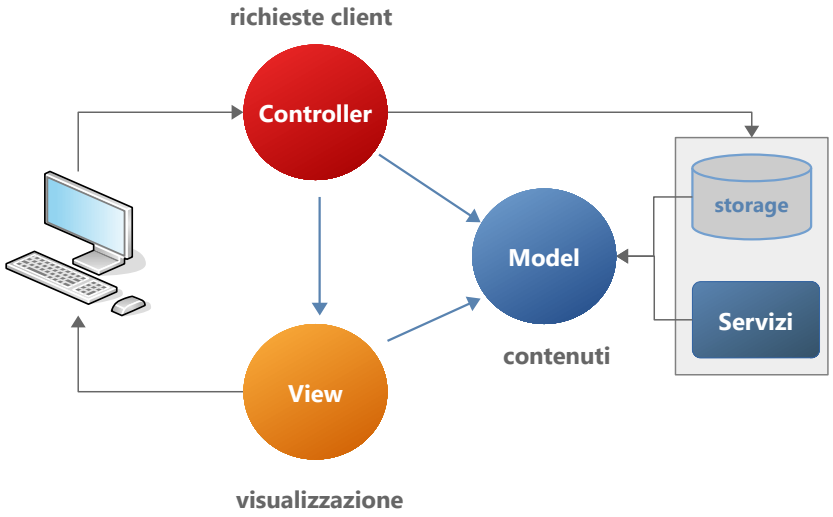
ASP.NET Core (***A**ctive **S**erver **P**ages **.NET Core***) rappresenta un *framework cross-platform* per la realizzazione di applicazioni web. Il *framework* è basato su .NET Core, la versione multi piattaforma del .NET Framework.

ASP.NET consente di realizzare vari tipi di applicazione, alcune delle quali agiscono specificatamente sul client. Questo tutorial si occupa della 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 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 un'applicazione di gestione di una biblioteca, fanno parte del *model* le *entity class* `Libro`, `Autore`, `Genere`, etc.
- *View*: presentano i contenuti; rappresentano dunque l'interfaccia utente dell'applicazione.
- *Controller*: elaborano le richieste dell'utente, ottengono i dati (*model*) che soddisfano tali richieste e li passano alle *view*, che li presentano all'utente.



2.1.1 Relazione tra i componenti Model, View e Controller

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

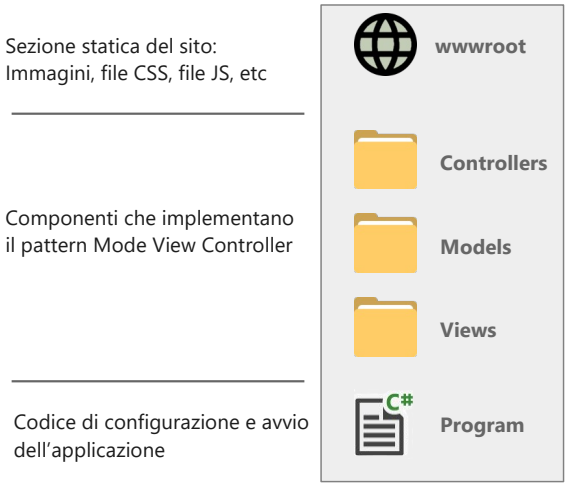
Il *model* è indipendente da *view* e *controller*: non dipende dunque dal tipo applicazione, dalla sua architettura o interfaccia utente. Lo stesso *model* potrebbe essere utilizzato in applicazioni desktop, *mobile*, web, etc.

Il *view* dipende dal *model*, poiché ha la funzione di presentarlo, ma non dipende dal *controller*. 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* 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 si trovano nelle tre cartelle centrali. Per quanto riguarda gli altri:

- la cartella **wwwroot** definisce la parte statica del sito, nella quale sono collocate eventuali pagine HTML, immagini, file CSS e librerie javascript.
- Il file **Program** contiene il codice di avvio e configurazione dell'applicazione. È qui che sono definiti i servizi utilizzati, come ad esempio Entity Framework.

2.2.1 Architettura del pattern MVC

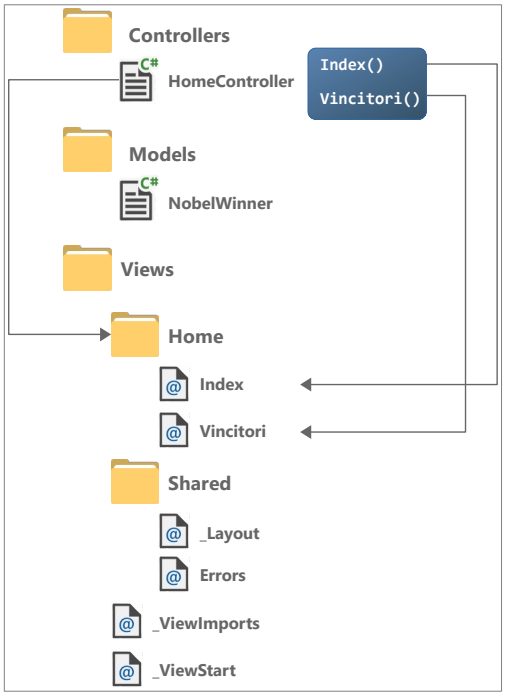
Lo schema a destra riassume la struttura dei componenti che implementano il pattern MVC. La separazione di componenti non è soltanto logica, ma anche fisica, poiché risiedono in cartelle e file separati.

L'applicazione definisce un solo *controller*, `HomeController`, il quale definisce un metodo (*metodo action*) per ogni *view* che deve visualizzare. (`Index()` e `Vincitori()`, nell'esempio)

Le *view* sono file con estensione **cshtml** e sono collocate in sottocartelle. 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, le *view* **_ViewImports** e **_ViewStarts** non presentano contenuti, ma hanno la funzione di semplificare la realizzazione delle altre *view*.



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.

2.3 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 diverso da quello schematizzato nel paragrafo 1.3.

2.3.1 Codifica degli URL: route

In ASP.NET gli URL hanno una struttura chiamata *route*: non specificano l'indirizzo della risorsa, ma il *controller* e l'*action* che devono rispondere alla richiesta. In sintesi: *una route specifica un metodo da eseguire*.

Una *route* rispecchia la seguente struttura:

`/[controller]/[action]/[id]`

Le parentesi quadre indicano che nessuno degli elementi è obbligatorio. Se omessi, il *controller* e/o l'*action* sono sostituiti dai valori predefiniti: **home** e **index**.

Nel caso del sito Nobel, le *route* per ottenere la *home page* e la pagina dei vincitori sono:

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

Per quanto riguarda la *home page*, la *route* può ridursi al simbolo `/`, dato che **home** e **index** sono il *controller* e l'*action* predefiniti.

2.3.2 Route con parametro

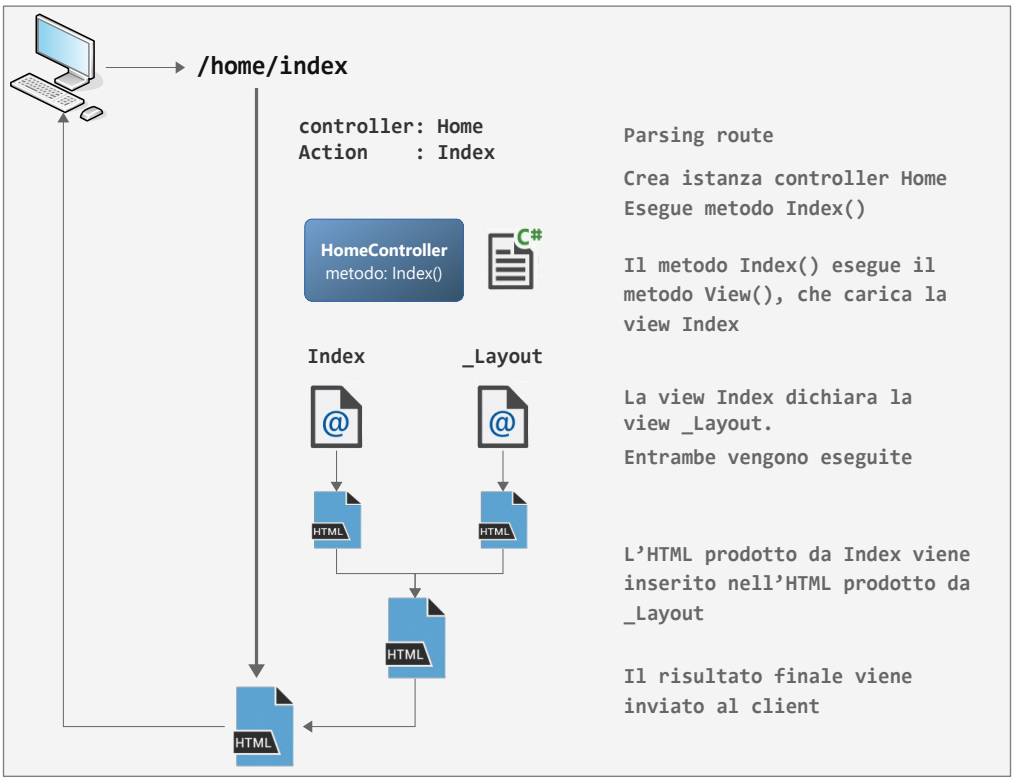
Una *route* può specificare un parametro facoltativo (campo **id** nella struttura della *route*) . (4.2)

2.3.3 Elaborazione della richiesta

È opportuno ribadirlo: alla richiesta del client corrisponde sempre l'esecuzione di un metodo di un *controller*.

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

Supponi, ad esempio, che il client richieda la *home page* del sito, specificando l'URL **http://www.nobel.it/**. Lo schema seguente mostra a grandi linee come viene elaborata la richiesta.



È il metodo `Index()` che stabilisce il contenuto da restituire al client:

1. Il metodo `Index()` carica la *view* **Index**, che stabilisce come visualizzare i contenuti.
2. La view **Index** dichiara la *layout view* **_Layout**, che definisce la struttura generale delle pagine del sito.
3. Viene eseguito il codice esecutivo contenuto in entrambe le *view*.
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 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 fornisce una tecnologia, chiamata *razor*, che consente di integrare in modo semplice il codice C# e quello HTML. Infatti, mediante il carattere `@` è possibile inserire un costrutto C# in qualsiasi punto della *view.*

Seguono alcuni esempi (per un approfondimento vedi: Guida completa Razor [4])

2.4.1 Blocchi di codice

Un *blocco* è rappresentato da una coppia di parentesi graffe. Le variabili dichiarate in un *blocco* sono accessibili ovunque nella *view*, a partire dalla linea successiva al blocco.

Ad esempio:

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

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

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

2.4.2 Costrutti di controllo

`if`, `for`, `foreach`, `while`, etc. I costrutti non devono terminare con il carattere `;`. Ad esempio:

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

2.4.3 Espressioni implicite

Le *espressioni implicite* sono costrutti che producono un valore: costanti, variabili, espressioni di qualunque tipo, chiamate a metodi e proprietà.

Ad esempio:

```
@DateTime.Now

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

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

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 l'espressione.)

2.4.4 Espressioni esplicite

Le *espressioni esplicite* sono rappresentate da una coppia di `( )` contenenti l'espressione.

Ad esempio:

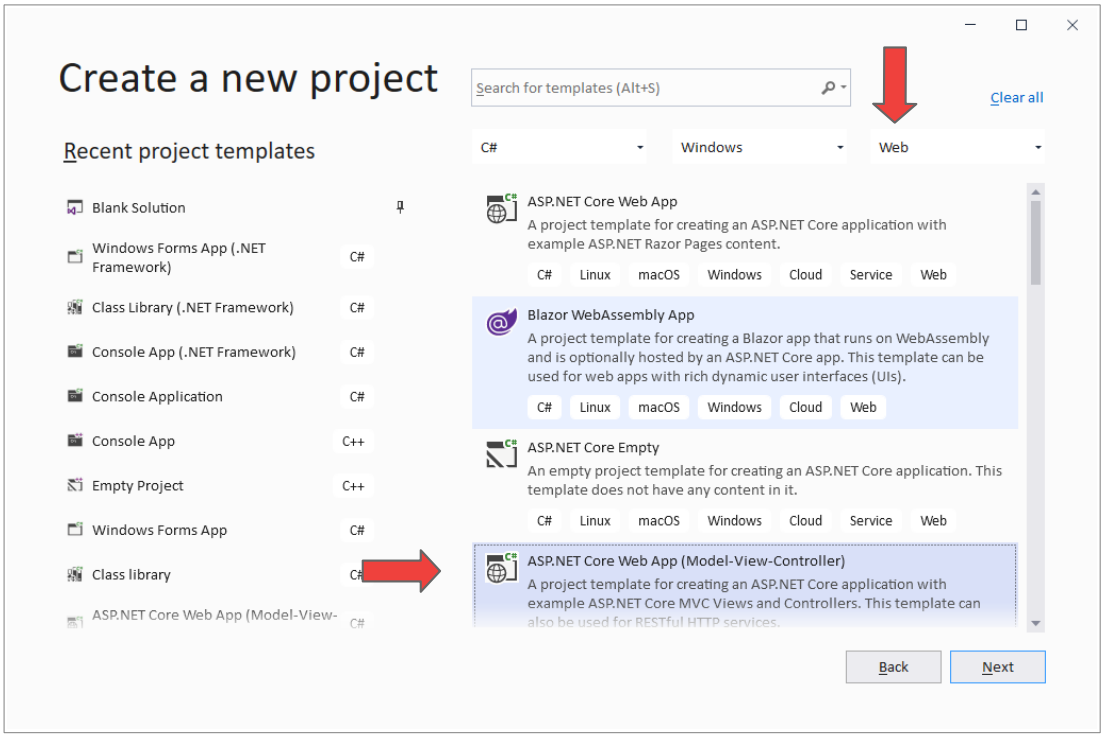
```
<p>L'altra settimana: @( DateTime.Now - TimeSpan.FromDays(7) )</p>
```

3 Creare applicazioni ASP.NET MVC

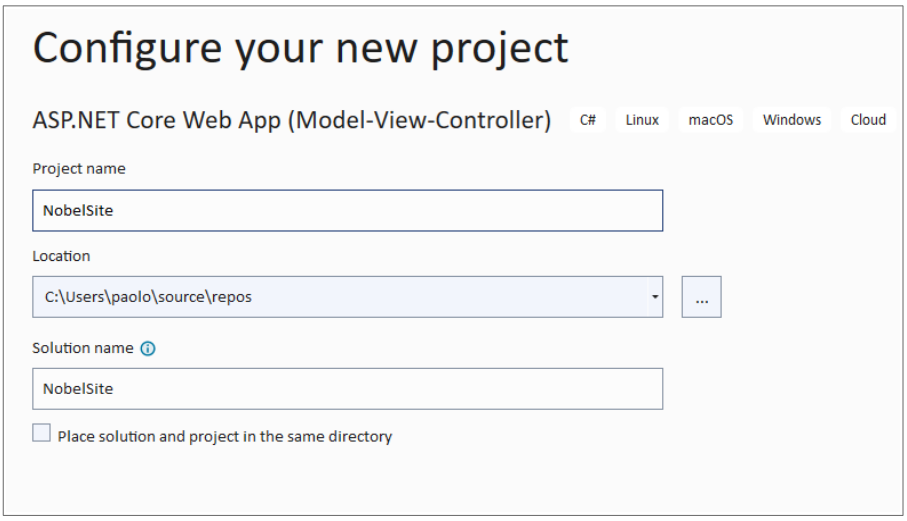
Di seguito mostrerò come realizzare un sito ASP.NET MVC che emuli il comportamento di quello realizzato in PHP. Nel mentre aggiungerò altri dettagli sul funzionamento di ASP.NET Core.

3.1 Creazione di un progetto

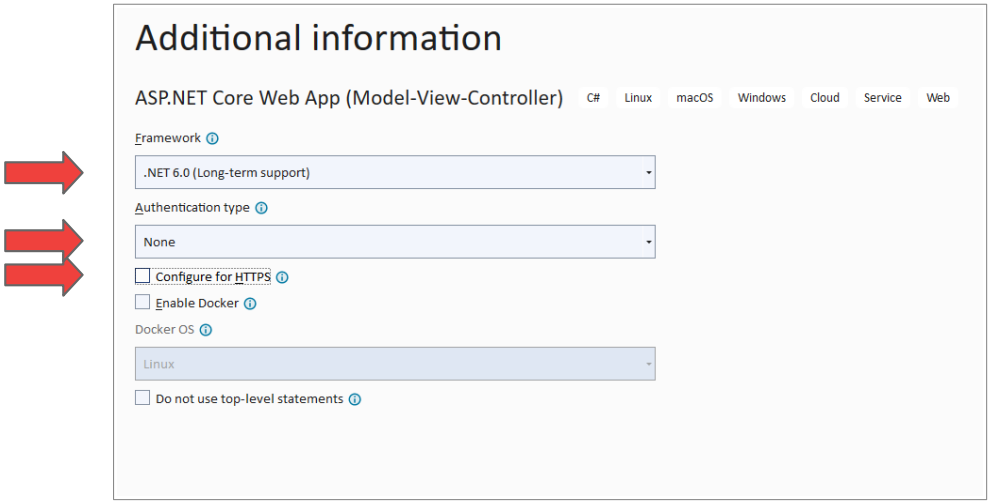
Si crea un progetto selezionando il tipo di applicazione **ASP.NET Core Web App (Model-View-Controller)**, nella categoria **web**:



Successivamente si inserisce il nome e la collocazione della *solution*.



Infine si configurano alcuni elementi dell'applicazione:



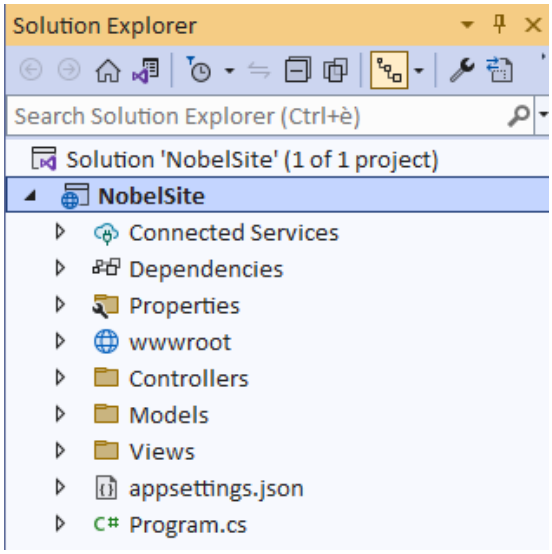
3.2 Struttura generale del progetto

ASP.NET crea un'applicazione già funzionante, dotata di *view* e *layout view* che forniscono dei contenuti dimostrativi.

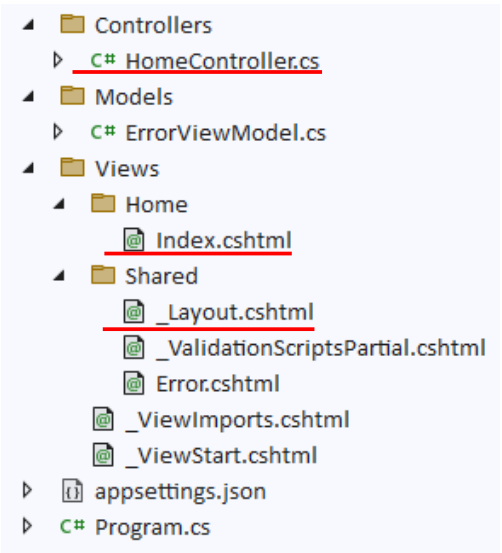
Allo scopo di partire da zero, ho eliminato quasi tutto il codice generato automaticamente, lasciando la sola *view* **Index** e riducendola, insieme alla **_Layout** *view*, a un contenuto minimale:

Il *solution explorer* mostra la seguente struttura:

Struttura generale



Struttura pattern MVC



3.3 Contenuto delle *view* *_Layout* e *Index*

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>
```

Ho evidenziato 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 presentare contenuti, ma stabiliscono il codice da eseguire al caricamento di ogni *view*, evitando che sia necessario specificarlo ogni volta:

- **_ViewStart** stabilisce la *layout view* che definisce 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 NobelSite
@using NobelSite.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

3.4 Classe HomeController

Segue l'`HomeController` (anche in questo caso ho ridotto il codice al minimo):

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

Il metodo `View()` restituisce la *view* corrispondente a `Index()`. Infatti, 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 la localizza all'interno della struttura del sito e la esegue usando il pattern:

- *controller* `HomeController` → cartella **Home**
- *metodo action* `Index()` → *view* **Index**

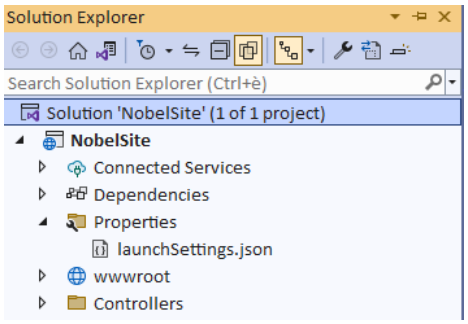
3.5 Configurare l'esecuzione dell'applicazione

Insieme ad ASP.NET sono installati due web server: **Kestrel** e **IIExpress**. Di default, l'applicazione viene eseguita su **Kestrel** all'indirizzo **localhost** e a un numero di porta casuale.

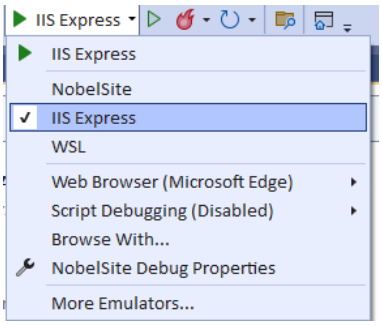
Poiché intendo eseguire l'applicazione su **IIExpress** alla porta 5000, è necessario modificare le impostazioni di avvio. Per farlo è possibile aprire le proprietà del progetto, oppure modificare il file **launchSettings.json**, collocato nella cartella **Properties**:

Titolo

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  ...
}
```



Si può scegliere il *web server* sul quale eseguire l'applicazione selezionando il menù a discesa del comando di esecuzione. Questo consente anche scegliere il browser da utilizzare per navigare sull'applicazione.



L'esecuzione produce:

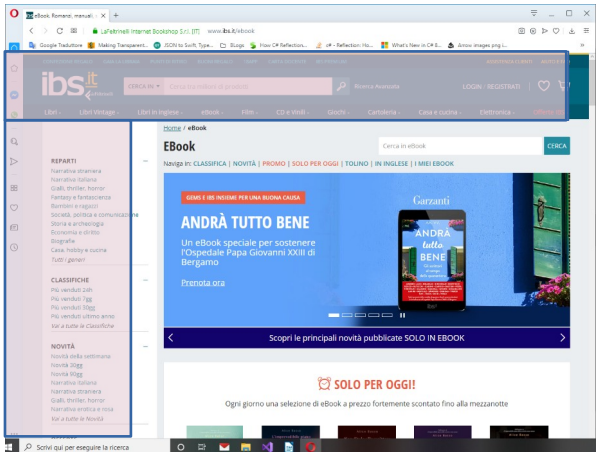
1. L'avvio del *web server* **IIExpress** (nell'area di notifica appare l'icona corrispondente), se non è già in esecuzione.
2. L'avvio dell'applicazione.
3. L'apertura del browser e la richiesta all'URL: **localhost:5000/home/index**

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

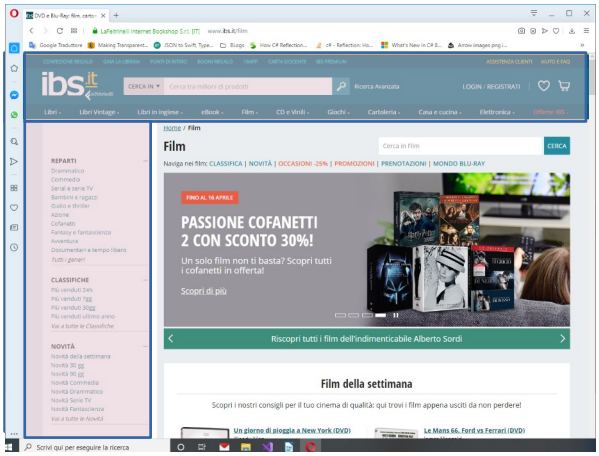
Il modello di presentazione dei contenuti adottato da ASP.NET Core viene incontro a un problema tipico nella realizzazione di siti web: evitare la duplicazione nelle pagine del sito.

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

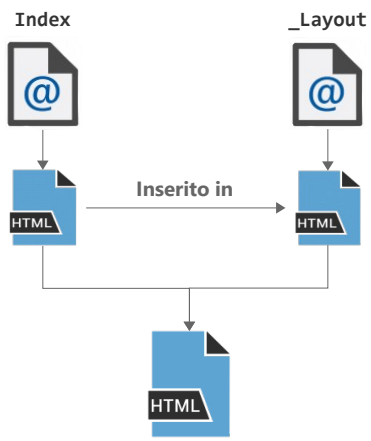


Nella maggior parte dei siti, infatti, le pagine condividono una struttura, dei contenuti e delle risorse comuni, come fogli di stile e librerie *javascript*.

A questo scopo tutti i *framework* per lo sviluppo di siti web implementano il concetto di *master page*, o *layout page*, e cioè la possibilità di collocare la struttura comune in un componente particolare, in modo che le pagine del sito possano limitarsi a presentare i contenuti specifici.

In ASP.NET 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, il contenuto della *view* viene inserito nel contenuto prodotto dalla *layout view*.

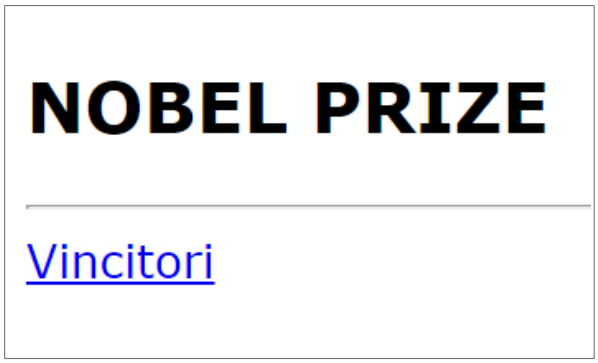


Nell'esempio, **Index** e **_Layout** producono il seguente risultato:

HTML inviato 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

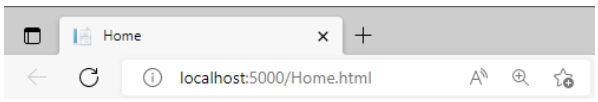


(l'HTML evidenziato è prodotto da **Index**, mentre il resto è prodotto da **_Layout**.)

3.7 Impostare il titolo della pagina: uso del dizionario ViewData

Il titolo delle pagine web, visualizzato sulla barra del titolo del browser, viene impostato mediante il tag **title**, definito all'interno del tag **head**.

```
<html lang="en">
<head>
    <title>Home</title>
</head>
<body>
    ...
</body>
</html>
```



In ASP.NET i contenuti inviati al client sono definiti nelle *view*, ma il tag **head** è specificato una volta per tutte nella *layout view*. Dunque: il titolo della pagina non può essere definito nelle *view*, come invece dovrebbe!

La soluzione a questo problema 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>
...
```

4 Implementare il Model View Controller

Partendo dal progetto appena creato, mostrerò in azione i tre elementi di base delle applicazioni web MVC.

4.1 Visualizzazione dei vincitori

L'obiettivo è implementare una pagina che visualizzi l'elenco dei vincitori del premio Nobel. Si suppone che i dati si trovino nel file CSV **Vincitori.txt** collocato nella cartella **Data**. Si suppone inoltre che nella *home page* ci sia un *link* per richiedere la pagina suddetta.

Per implementare questa funzione occorre:

1. Rappresentare il singolo vincitore: il *model*.
2. Visualizzare l'elenco dei vincitori: la *view*.
3. Ricevere la richiesta dal browser, caricare i dati e passarli alla *view*: il *controller*.

4.1.1 Rappresentare il singolo vincitore

È sufficiente un *record*:

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

4.1.2 Visualizzare i vincitori: view Vincitori

Per creare una *view***,** Visual Studio fornisce la voce di menù necessaria, ma è possibile eseguire il semplice copia incolla di un'altra *view* (**Index** in questo caso).

Nella *view* **Vincitori** occorre scorrere l'elenco dei vincitori per visualizzarne il nome:

```
@{
    ViewData["Title"] = "Vincitori";
}
<div>
    @foreach (var wi in ???)  // <- dove sono memorizzati i vincitori?
    {
        <p>@wi.FullName</p>
    }
</div>
```

Per ottenere l'elenco dei vincitori da visualizzare, si potrebbe scrivere il codice necessario nella *view*, ma in questo modo non sarebbe rispettata la separazione delle funzioni dettata dal pattern MVC.

È compito del *controller* ottenere i dati e passarli alla *view*.

4.1.3 Passaggio dei contenuti alla view Vincitori

La funzione dell'`HomeController` è quella di:

1. Rispondere alla richiesta della pagina **vincitori**.
2. Caricare i dati dei vincitori dal file CSV.
3. Passare i dati alla *view* **Vincitori**.

Il punto 2) è implementato da un metodo che legge il file CSV e restituisce una lista di record `NobelWinner`:

```
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;
  }
}
```

(L'uso del *modulo* `File` richiede di specificare anche il *namespace*, poiché il nome `File` collide con un metodo della `HomeController`.)

4.1.4 Passaggio *dei contenuti* alla view Vincitori: uso di ViewData

Esistono due modi per passare i vincitori alla *view*; il primo vede l'uso del dizionario `ViewData`:

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

    public IActionResult Vincitori()
    {
        ViewData["vincitori"] = LoadNobelWinners();
```

```
        return View();
    }
    ...
}
```

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

L'uso di `ViewData` per passare i contenuti alla *view* non la soluzione migliore. La strategia predefinita prevede innanzitutto che la *view* dichiari il *model* mediante la direttiva `@model`:

```
@model <tipo-dei-dati>
```

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>
```

L'uso di questa direttiva produce la dichiarazione di una variabile di nome `Model` di tipo `List<NobelWinner>`.

Nell'`HomeController`, si passano i dati alla *view* specificandoli come parametro al metodo `View()`:

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

4.2 Passare dati in una richiesta: *data binding*

Nell'esempio presentato, la richiesta del client specifica soltanto la *route* da eseguire (**/home/vincitori**). 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 un elenco di articoli; cliccando su uno qualsiasi di essi si apre una pagina che ne mostra i dettagli. In questo caso la richiesta dovrà specificare sia la pagina che visualizza i dettagli, sia l'articolo da visualizzare.

In ASP.NET si possono usare due tecniche, o una loro combinazione:

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.

In entrambi i casi, ASP.NET applica un meccanismo chiamato ***databinding***, che associa il valore/i presenti nell'URL al parametro/i del metodo che dovrà elaborare la richiesta.

4.3 Passare un parametro nella *route*

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

`/[controller]/[action]/[id]`

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

Il nominativo del vincitore rappresenta il campo **id** della *route*.

4.3.1 Gestire una richiesta con parametro nella route

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

```
public IActionResult Vincitore(string id)  //id -> nome compoleto del vincitore selezionato
{
    ...
}
```

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

4.4 Usare una *query string*

Una *query string* 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>
```

4.4.1 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 meccanismo di *databinding* si occuperà di trovare la corrispondenza tra la chiave della *query string* e il parametro del metodo. (Anche in questo caso non esiste distinzione tra maiuscole/minuscole.)

4.4.2 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.

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

I *tag helper* sono chiamati anche attributi *server-side*, poiché vengono processati da ASP.NET e non dal browser. Sono usati in molti scenari, tra i quali separare le parti che compongono l'URL di un *hyperlink*:

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

       <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, nell'eseguire la *view*, a tradurre la seconda forma nella prima.

5 Un esempio completo di applicazione: MotoGP

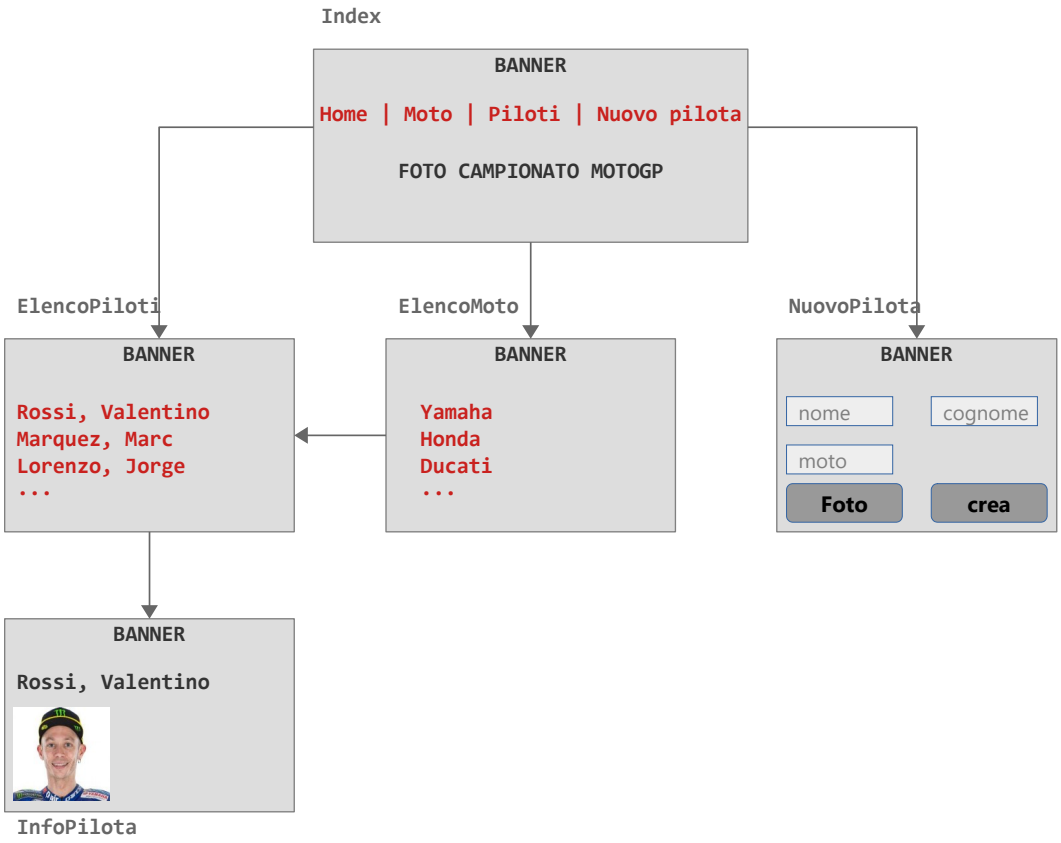
Di seguito svilupperò un'applicazione allo scopo di approfondire le funzioni già introdotte e introdurne di nuove.

5.1 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, ognuna delle quali presenta un menù che fornisce l'accesso 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.



I dati sono memorizzati nel database **MotoGP**, che definisce le tabelle **Moto** e **Piloti**. Per l'accesso ai dati utilizzerò Entity Framework.

5.1.1 View e controller

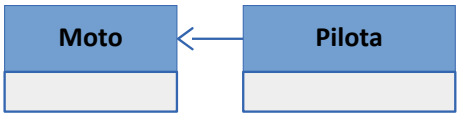
Intendo definire le seguenti *view:*

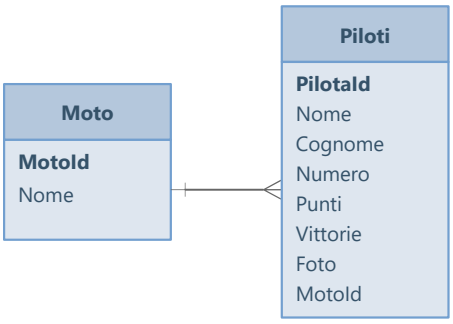
- **Index**: è la *home page* e mostra una foto.
- **ElencoMoto**: visualizza l'elenco delle marche iscritte al mondiale. Consente di cliccare su una moto e ottenere l'elenco dei piloti che corrono con essa. (*view* **ElencoPiloti**)
- **ElencoPiloti**: visualizza un elenco dei piloti. Questi possono essere tutti i piloti iscritti al campionato, oppure i piloti che corrono con una determinata moto.
  Consente di cliccare sul nominativo di un pilota e ottenere le informazioni disponibili (*view* **DettagliPilota**)
- **DettagliPilota**: visualizza le informazioni sul singolo pilota.
- **NuovoPilota**: consente l'inserimento di un nuovo pilota.

Tutte le richieste saranno gestire dall'`HomeController`. In uno scenario realistico, con decine o centinaia di *view*, l'applicazione suddividerebbe le varie funzioni in più *controller*.

5.2 Model (*entity model*)

Il *model* è rappresentato dai *record* `Pilota` e `Moto`, collocati nella cartella **Models**. Le due *entity class* mappano le tabelle **Moto** e **Piloti**:






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

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

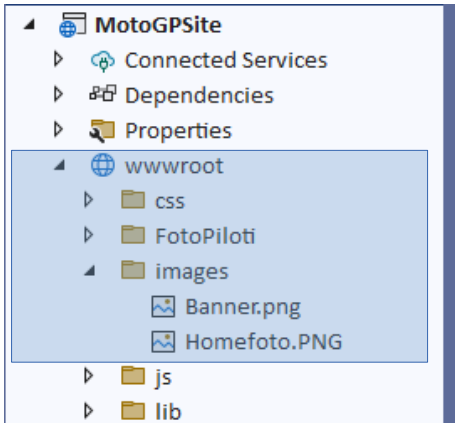
Per semplicità, nella cartella **Models** è collocata anche la classe *context*, `MotoGPContext`.

5.3 Layout view

La *layout view* visualizza un banner e il menù, poiché entrambi gli elementi devono apparire 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 il foglio di stile **site.css** e l'immagine del banner. Entrambi sono memorizzati nella cartella `wwwroot`, all'interno della quale sono presenti anche le foto dei piloti.



5.4 Home page

La *view* **Index** si limita a visualizzare un'immagine:

Index

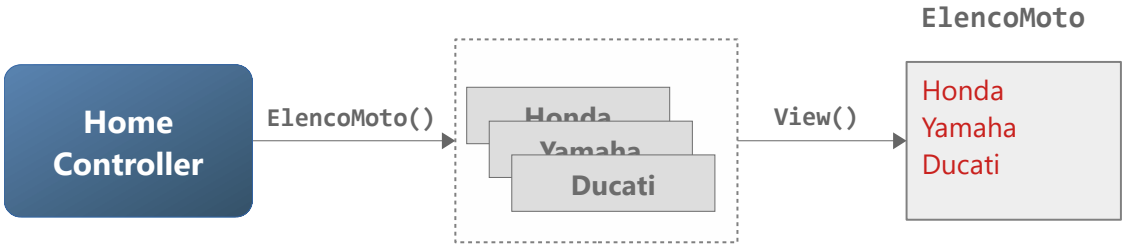
```
@{ViewData["Title"] = "Home";}

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

Metodo Index()

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

5.5 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 raccolta di `Moto`.
- Il *controller* ottiene questo elenco da `MotoGPStore` e lo passa alla *view*.

Segue l'implementazione della *view*:

```
@{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>
```

Alternativamente avrei potuto scrivere:

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

Ogni *link* referenzia l'*action* **elencoPiloti** e specifica l'id della moto visualizzata.

5.5.1 Controller: ottenere il model e passarlo alla *view*

Il compito del *controller* è quello di usare la classe *context* per ottenere i dati dal database e passarli alla *view*:

```
public class HomeController : Controller
{
    MotoGPContext db = new MotoGPContext();
    ...
    public IActionResult ElencoMoto()
    {
        return View(db.Moto);  // passa le moto alla view
    }
}
```

5.6 Visualizzazione dell'elenco dei piloti - ElencoPiloti

La visualizzazione dei piloti rappresenta un esempio di come il pattern MVC faciliti la separazione tra l'elaborazione delle richieste e la presentazione dei dati. I piloti possono essere visualizzati in due circostanze diverse:

- 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.

Questa differenziazione non riguarda la visualizzazione in sé; la *view* **ElencoPiloti** si limita a ricevere una raccolta di piloti e a visualizzarla.

```
@{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 pilota viene visualizzato mediante un *link*, in modo che l'utente possa cliccarlo per accedere alla pagina di informazioni sul pilota.
- Poiché viene visualizzata anche la moto del pilota, è necessario che nella proprietà di navigazione `Moto` siano stati caricati i dati necessari.

5.7 Caricamento dei piloti

Esistono due richieste distinte che producono un elenco di piloti:

`home/ElencoPiloti>` (tutti i piloti)

e

`home/ElencoPiloti/<id>` (piloti che corrono con una determinata moto)

D'altra parte esiste una sola *view* con la funzione di visualizzarli. Ciò "rompe" la normale corrispondenza: una richiesta → un *metodo action* → una *view*.

Una soluzione sembrerebbe consistere nel definire due metodi con lo stesso nome, uno dei quali dichiara un parametro `id`:

```
public class HomeController : Controller
{
    ...
    // gestisce la richiesta: /Home/ElencoPiloti
    public IActionResult ElencoPiloti()
    {
        ...
    }
    // gestisce la richiesta: /Home/ElencoPiloti/<id>
    public IActionResult ElencoPiloti(int id)
    {
        ...
    }
}
```

Purtroppo non funziona: il meccanismo di *databinding* non fa distinzione tra i due metodi con lo stesso nome, i quali sarebbero entrambi eleggibili a rispondere alle due richieste. In conclusione: ASP.NET non sa quale metodo scegliere e produce un errore di esecuzione.

5.7.1 Usare un parametro id nullabile

Una soluzione è definire un solo metodo con un parametro `id` nullabile.

Se la richiesta viene dal menù e dunque non specifica l'**id** il meccanismo di *databinding* assegnerà `null` al parametro `id`. Se la richiesta viene dalla *view* **ElencoMoto**, il parametro `id` conterrà la chiave della moto selezionata.

Nel metodo basterà verificare se `id` è nullo per decidere come rispondere alla richiesta:

```
public class HomeController : Controller
{
    ...
    public IActionResult ElencoPiloti(int? id)
    {
        var piloti = id is null
            ? db.Piloti                                          // -> /Home/ElencoPiloti
            : db.Piloti.Where(p => p.MotoId == id.Value);    // -> /Home/ElencoPiloti/<id>

        return View(piloti.Include(p => p.Moto));   // include la moto (eager loading)
    }
}
```

Nota bene: indipendentemente dal fatto che siano caricati tutti i piloti o soltanto quelli che corrono con una certa moto, viene usato il metodo `Include()` per caricare anche i dati delle moto.

5.7.2 Usare un normale parametro intero

Una seconda soluzione prevede l'uso di un normale parametro `id` intero, poiché si basa sull'assunto che le chiavi *identity* partono dal valore 1. Se la richiesta viene dal menù e dunque non specifica l'**id**, il meccanismo di *databinding* assegnerà zero al parametro `id` del metodo.

```
public class HomeController : Controller
{
    ...
    public IActionResult ElencoPiloti(int id)
    {
        var piloti = id == 0
            ? db.Piloti                                  // -> /Home/ElencoPiloti
            : db.Piloti.Where(p => p.MotoId == id);    // -> /Home/ElencoPiloti/<id>

        return View(piloti.Include(p => p.Moto));   // include la moto (eager loading)
    }
}
```

5.7.3 Usare due metodi che producono l'elenco dei piloti

Infine, la terza possibilità è quella di usare due metodi distinti, che restituiscono uno l'elenco completo dei piloti, l'altro i piloti che corrono con una determinata moto.

In questo caso l'URL del menù principale specificherà l'*action* **ElencoPiloti**, mentre l'URL della *view* **ElencoMoto** utilizzerò l'*action* **ElencoPilotiMoto**.

```
public class HomeController : Controller
{
    ...
    public IActionResult ElencoPiloti()  // -> /Home/ElencoPiloti
    {
        return View(db.Piloti.Include(p => p.Moto));
    }
```

```
    public IActionResult ElencoPilotiMoto(int id)   // -> /Home/ElencoPiloti/<id>
    {
        return View(db.Piloti.Where(p => p.MotoId == id).Include(p => p.Moto));
    }
}
```

5.8 Informazioni sul pilota

La view **DettagliPilota** viene caricata in risposta al *link* sul pilota nella *view* **ElencoPiloti**:

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

Il metodo `DettagliPilota()` riceve l'**id** del pilota, lo cerca e lo passa alla *view* omonima:

```
public class HomeController : Controller
{
    ...
    public IActionResult DettagliPilota(int id)
    {
        var pilota = db.Piloti.Find(id);
        db.Entry(pilota).Reference(pilota => pilota.Moto).Load(); // carica dati moto
        return View(pilota);
    }
}
```

Nota bene: dopo aver trovato il pilota viene usato l'*explicit loading* per caricare la moto del pilota.

La *view* dichiara il tipo `Pilota` come *model* e ne visualizza i dati:

```
@{ViewData["Title"] = "Pilota";}
@model Pilota
@{
    string statistiche = $"Vittorie: {Model.Vittorie} --- Punti: {Model.Punti}";
    string moto = $"{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>
```

Per semplificare il codice HTML, utilizzo due variabili contenenti le statistiche e il nome della moto del pilota.

6 Inserimento dei dati

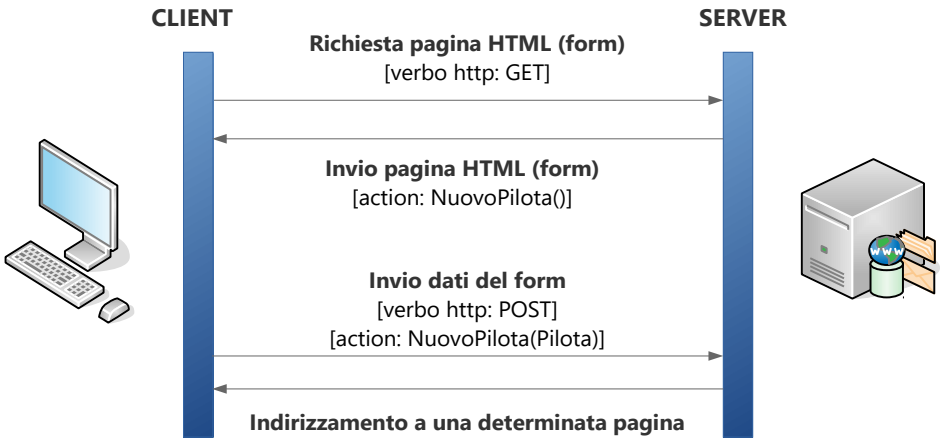
La creazione di un nuovo pilota richiede di implementare:

- Un form HTML per l'inserimento dei dati.
- L'uso di una *casella di riepilogo* per selezionare la moto del pilota. Questa deve essere popolata con l'elenco delle moto.
- La possibilità di "uploadare" la foto del pilota.

6.1 Gestione di un form HTML

La gestione 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. Il server verifica la validità dei dati, li inserisce nel database e reindirizza il browser a un'altra pagina.



Questo processo è gestito mediante due *metodi action*, ai quali si assegna di norma lo stesso nome. Il primo carica la *view* contenente il form vuoto; il secondo riceve i dati inseriti nel form e procede all'inserimento nel database.

Pur avendo lo stesso nome, ASP.NET non fa confusione nel selezionare il metodo giusto tra i due, poiché sono decorati con attributi che ne stabiliscono il ruolo: `[HttpGet]` e `[HttpPost]`. Il primo corrisponde al verbo HTTP **GET** (usato quando il browser chiede il form); il secondo corrisponde al verbo HTTP **POST** (usato dal browser quando invia il form all'applicazione)

```
[HttpGet]
public IActionResult NuovoPilota() // -> restituisce la pagina destinata al browser
{
    return View();
}

[HttpPost]
public IActionResult NuovoPilota(Pilota pilota)  // -> elabora i dati provenienti dal
{                                                //    browser
     //... inserisce pilota nel database
}
```

L'attributo `[HttpGet]` non è obbligatorio, poiché viene considerato il verbo di default quando viene elaborata una richiesta.

6.2 Implementazione del form HTML

Segue una prima versione della *view* **NuovoPilota**; non considera l'input della moto e l'upload dell'immagine:

```
@{
    ViewData["Title"] = "Nuovo pilota";
}
<div>
    <form action="/home/NuovoPilota" method="post">
        <input type="text" name="Nome" placeholder="Nome"/>
        <input type="text" name="Cognome" placeholder="Cognome"/>
        <input type="text" name="Numero" placeholder="Numero"/>
        <input type="text" name="Punti" placeholder="Punti"/>
        <input type="text" name="Vittorie" placeholder="Vittorie"/>
        <input type="submit" value="Crea" />
    </form>
</div>
```

Nota bene:

- Gli attributi **action** e **method** identificano il metodo `NuovoPilota(Pilota)` come quello che riceverà i dati inseriti dall'utente.
- Nei tag **input** è l'attributo **name** che determina a quale proprietà del *record* `Pilota` sarà assegnato il valore corrispondente.

6.3 Implementazione del form usando i tag-helper

ASP.NET fornisce dei *tag helper* (4.5) che semplificano l'implementazione di un form e consentono di sfruttare l'intellisense di Visual Studio nella valorizzazione degli attributi **name** dei tag **input**.

Segue una seconda versione della *view* **NuovoPilota** che usa questa funzione:

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

        <input type="submit" value="Crea" />
    </form>
</div>
```

La *view* dichiara il *model;* ciò, unito all'uso dei *tag helper* **asp-for**, fa sì che ASP.NET sia in grado di generare

gli attributi **type** e **name** per i tag di **input**.

6.4 Elaborare i dati del form: inserire il pilota

L'invio del form da parte del browser produce una richiesta HTTP **POST** alla quale è allegata una stringa contenente coppie *chiave-valore*, dove le chiavi sono i nomi dei tag **input** e i valori i dati inseriti. Ad esempio, se l'utente inserisce i dati di **Francesco Bagnaia**, viene allegata la stringa:

Nome=Francesco&Cognome=Bagnaia&Numero=63&...

Se il metodo che elabora la richiesta definisce un *record* come parametro, il meccanismo di *databinding* utilizza le chiavi per stabilire a quali del proprietà del *record* assegnare i valori corrispondenti.

Segue l'implementazione del metodo che elabora la richiesta:

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota)  // pilota viene popolato con i dati inseriti
{
    db.Piloti.Add(pilota);
    db.SaveChanges();
    return RedirectToAction("ElencoPiloti");    // indirizza l'utente alla view ElencoPiloti
}
```

Dopo aver inserito il pilota nel database, il metodo utilizza `RedirectToAction()` per reindirizzare l'utente alla *view* **ElencoPiloti**.

6.5 Inserimento della moto (selezione dall'elenco di moto)

Nel form manca ancora la possibilità di selezionare la moto con la quale corre il pilota.

Il modo corretto per l'inserimento della moto è quello di consentire all'utente di selezionarla dall'elenco di moto mediante un tag **select** (*casella di riepilogo*). Qui sorge il problema di come visualizzare le moto nel form, poiché il *model* utilizzato è di tipo `Pilota`, il quale non definisce l'elenco delle moto.

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

```
[HttpGet]
public IActionResult NuovoPilota()
{
    ViewData["listaMoto"] = db.Moto;
    return View();
}
```

Nella *view* si memorizza l'elenco in una variabile e 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" ...>
       ...
```

```
        <select asp-for="MotoId"
                asp-items="@(new SelectList(listaMoto,"MotoId","Nome"))">
            <option value="">Scegli la moto</option>
        </select>

        <input type="submit" value="Crea" />
    </form>
</div>
```

Alternativamente, si può generare il tag **select** con un `foreach`:

```
<div>
    <form asp-controller="Home" ...>
       ...
        <select asp-for="MotoId"
                asp-items="@(new SelectList(listaMoto,"MotoId","Nome"))"></select>

        <select asp-for="MotoId">
            <option value="">Scegli la moto</option>
            @foreach (var moto in listaMoto)
            {
                <option value="@moto.MotoId">@moto.Nome</option>
            }
        </select>

        <input type="submit" value="Crea" />
    </form>
</div>
```

Nota bene: in entrambi i casi, ho scritto un tag **option** con **value** vuoto. Questo è il valore predefinito; serve per visualizzare la richiesta di input e viene inviato al server se l'utente non seleziona un valore dall'elenco.

Che sia usata la prima o la seconda tecnica, al browser viene restituito il seguente frammento HTML:

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

6.6 Upload del file con la foto del pilota

Resta da implementare la funzione di *upload* della foto del pilota. Per farlo occorre:

- Utilizzare un tag **<input type="file">**.
- Nel metodo `NuovoPilota()` accedere al file inviato dal browser.
- Ottenere il percorso della cartella **FotoPiloti**, collocata in **wwwroot** 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" ...>
        ...
        <input type="file" name="fileFoto"/>
    </form>
</div>
```

6.6.1 Salvare il file su disco

Al metodo `NuovoPilota()` occorre aggiungere un secondo parametro di tipo `IFormFile`; questo memorizza le informazioni sul file caricato e consente di salvarlo su disco.

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota, IFormFile fileFoto)
{
    if (fileFoto != null) // è stato uploadato un file?
    {
        var ext = Path.GetExtension(fileFoto.FileName);
        var nomeFile = $"wwwroot/FotoPiloti/{pilota.Cognome}{pilota.Nome}{ext}";
        var fs = new FileStream(nomeFile, FileMode.Create);
        fileFoto.CopyTo(fs);  // salva il file su disco
        fs.Close();
        pilota.Foto = Path.GetFileName(nomeFile);
    }
    db.Piloti.Add(pilota);
    db.SaveChanges();
    return RedirectToAction("ElencoPiloti");
}
```

Il codice che gestisce l'*upload* verifica se è stato caricato un file. In caso positivo:

- Ottiene l'estensione del file.
- Crea il percorso di destinazione, creando un nome di file composto da cognome e nome del pilota e dall'estensione del file.
- Crea un `FileStream` e lo usa per salvare il file.
- Assegna il nome breve del file alla proprietà `Foto` del pilota.

7 Validare i dati

I metodi che gestiscono l'input o la modifica dei dati dovrebbero sempre verificare che i dati ricevuti siano validi. In caso contrario l'applicazione dovrebbe visualizzare un messaggio di errore e riproporre il form.

A questo scopo, 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 nel form di inserimento/modifica del dati.

Questo meccanismo parte dalla configurazione del *model*, stabilendo mediante degli *attributi*, o *annotazioni*, i vincoli che le proprietà devono soddisfare.

7.1 Validazione dei singoli campi del pilota

Considera nuovamente la funzione di inserimento del pilota. Perché l'input sia valido occorre che i dati siano innanzitutto inseriti e che soddisfino determinati criteri.

```
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.)

Nota bene: non ho usato l'attributo `[Required]` sulle proprietà intere (`MotoId`, `Numero`, `Punti`, `Vittorie`), poiché non svolgerebbe alcuna funzione. Per definizione, infatti, le proprietà di *tipo valore* non possono essere `null` e dunque il loro valore è obbligatorio per definizione.

7.1.1 Validazione del model

Nel metodo `NuovoPilota()`, prima di procedere all'elaborazione dell'input, occorre verificare che i criteri stabiliti nel *model* siano stati soddisfatti. Lo si fa verificando la proprietà `ModelState.IsValid`:

```
[HttpPost]
public IActionResult NuovoPilota(Pilota pilota, IFormFile file)
{
    if (!ModelState.IsValid)  // l'input soddisfa i criteri stabiliti nel model?
    {
        ViewData["listaMoto"] = db.Moto;
        return View(pilota);  // ricarica il form con i dati già inseriti
    }

    // ... procede all'inserimento

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

Se il *model* non è valido, viene riproposto il form di input, passando alla *view* l'oggetto pilota. Ciò consente di riproporre i dati già inseriti, evitando che l'utente debba ricominciare da zero.

7.2 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 gli errori relativi ai singoli campi.
- **asp-validation-summary** è utilizzato per visualizzare un riepilogo e/o dei messaggi di errore personalizzati.

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 generali.

```
@{
    ViewData["Title"] = "Nuovo pilota";
    var listaMoto = ViewData["listaMoto"] as IEnumerable<Moto>;
}
@model Pilota
<div>
    <form asp-controller="Home" asp-action="NuovoPilota"
                                method="post" enctype="multipart/form-data">

        <div asp-validation-summary="ModelOnly" class="error-text"></div>

        <input asp-for="Nome" placeholder="Nome" />
```

```
        <span asp-validation-for="Nome" class="error-text"></span>

        <input asp-for="Cognome" placeholder="Cognome" />
        <span asp-validation-for="Cognome" class="error-text"></span>
        ...
    </form>
</div>
```

Il tag **div** ha la funzione visualizzare il riepilogo degli errori. Il valore **ModelOnly** indica che non saranno visualizzati i singoli errori relativi ai campi, ma soltanto un messaggio generale.
I singoli messaggi di errore vengono visualizzati attraverso i tag **span** posizionati accanto ai tag di **input**.

7.2.1 Aggiungere i messaggi di errore di riepilogo

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"] = db.Moto;
        ModelState.AddModelError("", "Uno o più campi contengono dati corretti");
        return View(pilota);
    }
    ...
}
```

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

7.2.2 Personalizzare i messaggi di errore

I messaggi di errore sono stabiliti da ASP.NET. È possibile personalizzarli, oppure 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; }
    ...
}
```

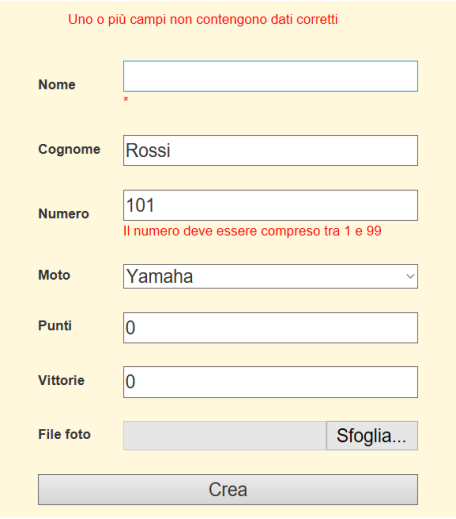
Poiché con i *tipi valore* l'attributo `[Required]` non svolge alcuna funzione, non è possibile personalizzare il messaggio di errore per le proprietà `MotoId`, `Numero`, `Punti` e `Vittorie`. Una soluzione a questo problema viene esaminata in 8.2.

7.3 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* che elabora il form 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"] = db.Moto;
        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 di moto non valido:



8 Utilizzare i *view model*

Nella funzione di inserimento di un pilota, il *controller* passa il *model* `Pilota` alla *view* e, mediante `ViewData`, l'elenco delle moto. Nel processare l'input dell'utente, il *controller* riceve un oggetto di tipo `Pilota` e un oggetto `IFormFile`, il quale consente l'accesso alla foto. Non si tratta di un buon modello di comunicazione tra *controller* e *view*:

- È complicato: sia il caricamento del form che la sua elaborazione coinvolgono due oggetti.
- La validazione richiede l'annotazione del tipo `Pilota` mediante *attributi*, ma non si dovrebbe legare la rappresentazione dei dati (il *model*) al modo in cui viene gestita la loro visualizzazione, errori compresi.
- Il tipo `Pilota` definisce delle proprietà (`PilotaId`, `Nominativo`, `Moto`) che non vengono utilizzate dalla *view*. In sostanza: la *view* riceve un oggetto contenente dati inutili alla visualizzazione.

Un approccio alternativo prevede che la comunicazione *controller→view* avvenga mediante un unico oggetto e che questo definisca tutto e soltanto ciò che serve per implementare la funzione richiesta.

Si parla in questo caso di *viewmodel*: un tipo la cui esistenza è strettamente legata alla *view* che lo utilizza.

8.1 Definire un *viewmodel*

Poiché un *viewmodel* è legato a una specifica *view*, per convenzione viene denominato con il nome della *view* e il suffisso `ViewModel`. L'inserimento di un nuovo pilota richiede la classe `NuovoPilotaViewModel`:

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

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

    public int MotoId { get; set; }

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

    [Range(0, 450, ErrorMessage = "Il valore deve essere compreso tra 0 e 450")]
    public double Punti { get; set; }

    [Range(0, 18, ErrorMessage = "Il valore deve essere compreso tra 0 e 18")]
    public int Vittorie { get; set; }

    public string Foto { get; set; }

    public IEnumerable<Moto> ElencoMoto { get; set; }

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

La classe definisce unicamente ciò che serve all'input del pilota, compreso l'elenco delle moto e gli *attributi* usati per la validazione dell'input.

8.1.1 Uso del viewmodel nella view

La nuova classe sostituisce `Pilota` nella *view*. Per il resto, cambia soltanto l'accesso all'elenco delle moto, che ora è memorizzato nella proprietà `ListaMoto` del *model*:

```
@{
    ViewData["Title"] = "Nuovo pilota";
}
@model NuovoPilotaViewModel
<div>
    <form asp-controller="Home" asp-action="NuovoPilota" method="post"
                                                         enctype="multipart/form-data">
        ...
        <select asp-for="MotoId"
                asp-items="@(new SelectList(Model.ListaMoto,"MotoId","Nome"))">
            <option value="">Scegli la moto</option>
            </select>
        ...
    </form>
</div>
```

8.1.2 Uso del viewmodel nel controller

Il metodo di caricamento del form carica la *view* passandogli un *viewmodel* vuoto, eccetto l'elenco delle moto:

```
[HttpGet]
public IActionResult NuovoPilota()
{
    var vm = new NuovoPilotaViewModel
    {
        ListaMoto = db.Moto
    };
    return View(vm);  // passa il viewmodel perché siano visualizzate le moto
}
```

Anche il metodo di elaborazione del form cambia solo superficialmente:

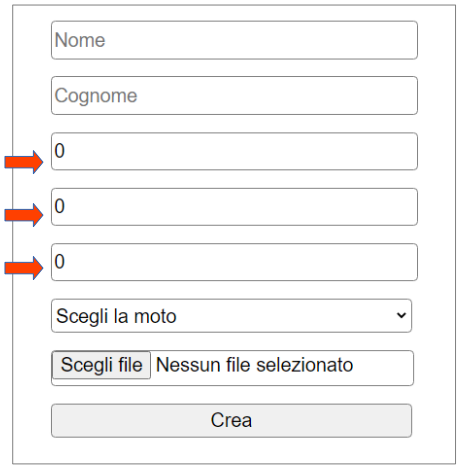
```
[HttpPost]
public IActionResult NuovoPilota(NuovoPilotaViewModel vm)
{
    if (!ModelState.IsValid)
    {
        vm.ListaMoto = db.Moto;
        return View(vm);
    }

    if (vm.FileFoto != null) // è stato selezionato un file?
    {
        var ext = Path.GetExtension(vm.FileFoto.FileName);
        var nomeFile = $"wwwroot/FotoPiloti/{vm.Cognome}{vm.Nome}{ext}";
        var fs = new FileStream(nomeFile, FileMode.Create);
        vm.FileFoto.CopyTo(fs);  // salva il file su disco
        fs.Close();
```

```
        vm.Foto = Path.GetFileName(nomeFile);
    }

    // crea un pilota a partire dal viewmodel
    var pilota = new Pilota
    {
        Nome = vm.Nome,
        Cognome = vm.Cognome,
        MotoId = vm.MotoId,
        Numero = vm.Numero,
        Punti = vm.Punti,
        Vittorie = vm.Vittorie,
        Foto = vm.Foto
    };
    ...
}
```

C'è una differenza importante: a partire dal *viewmodel* il metodo deve creare il pilota da inserire nel database. (Esistono software che automatizzano questo compito.)

8.2 Migliorare la visualizzazione del *viewmodel*

L'attuale soluzione è senz'altro da migliorare per quanto riguarda l'interfaccia utente.



Esistono due problemi:

- Per le proprietà di *tipo valore* non è possibile usare `[Required]` allo scopo di personalizzare il messaggio di errore.(7.2.2)
- Poiché la *view* **NuovoPilota** viene caricata insieme al *viewmodel*, le caselle di testo visualizzano il valore predefinito delle proprietà *value type*, invece che i *placeholder*.

Per entrambi esiste una soluzione completamente basata sul *viewmodel*:

- Rendere *nullabili* le proprietà *value type* i modo da poter applicare l'attributo `[Required]`.
- Utilizzare l'attributo `[Display]` per stabilire il *placeholder* delle caselle di testo.

```
public class NuovoPilotaViewModel
{
    [Display(Prompt = "Nome")]
    [Required(ErrorMessage ="*")]
    public string Nome { get; set; }

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

    [Required(ErrorMessage = "*")]
    public int? MotoId { get; set; }

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

Poiché adesso le proprietà *value type* sono *nullabili*, la *view* non visualizza più il valore predefinito (che è `null`), ma il *placeholder*.

Nella *view* **NuovoPilota** è possibile eliminare i *placeholder* dai tag HTML:

```
...
<div>
    <form asp-controller="Home" asp-action="NuovoPilota" method="post"
                                                         enctype="multipart/form-data">
        <div asp-validation-summary="ModelOnly" class="error-text"></div>
        <input asp-for="Nome" />
        <span asp-validation-for="Nome" class="error-text"></span>

        <input asp-for="Cognome" />
        <span asp-validation-for="Cognome" class="error-text"></span>
        ...
    </form>
</div>
```

Nel *controller* è necessario modificare il codice che crea il pilota a partire dal *viewmodel*:, poiché le proprietà `Numero`, `Punti`, etc del *viewmodel* sono *nullabili*:

```
[HttpPost]
public IActionResult NuovoPilota(NuovoPilotaViewModel vm)
{
    ...

    // crea un pilota a partire dal viewmodel
    var pilota = new Pilota
    {
        Nome = vm.Nome,
        Cognome = vm.Cognome,
        MotoId = vm.MotoId.Value,
```

```
        Numero = vm.Numero.Value,
        Punti = vm.Punti.Value,
        Vittorie = vm.Vittorie.Value,
        Foto = vm.Foto
    };
    ...
}
```

9 Configurare l'applicazione

ASP.NET Core implementa un processo di configurazione che consente di stabilire i servizi utilizzati dall'applicazione. Di seguito ne introduco le caratteristiche generali e fornisco un esempio di configurazione dell'oggetto *context* usato per accedere al database.

9.1 File Program

Il file **Program.cs** contiene il codice di configurazione e avvio dell'applicazione. È suddiviso in tre parti:

```
var builder = WebApplication.CreateBuilder(args);
                                                    Aggiunge i servizi all'applicazione
builder.Services.AddControllersWithViews();
-------------------------------------------------------------------------------------
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
}
app.UseStaticFiles();                                   Stabilisce le funzioni da usare

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
-------------------------------------------------------------------------------------
                                                        Avvia l'aplicazione
app.Run();
```

9.2 Stabilire e configurare i servizi da utilizzare

ASP.NET ha un'architettura modulare che prevede la definizione e la configurazione dei servizi richiesti dall'applicazione. Ad esempio, senza l'istruzione:

```
builder.services.AddControllersWithViews();
```

l'applicazione non implementerebbe il pattern *Model-View-Controller*. Oppure, senza l'istruzione:

```
app.UseStaticFiles();
```

l'applicazione non sarebbe in grado di restituire le risorse memorizzate in **wwwroot**, come immagini e file CSS.

9.3 Configurare la creazione automatica del *context*

Attraverso il codice di configurazione è possibile istruire ASP.NET a creare automaticamente il *context* e a passarlo ai *controller* che ne richiedono l'uso. Ciò consente di stabilire esternamente alla classe *context* i parametri del suo funzionamento, tra tutti: il *provider* e la *stringa di connessione*.

```
using MotoGPSite.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MotoGPContext>(
    options => options.UseSqlServer("Server=(localdb)\\mssqllocaldb;..."));

builder.Services.AddControllersWithViews();

var app = builder.Build();
...
```

9.3.1 Modifica della classe context

Perché la classe *context* possa utilizzare questa modalità di configurazione, è necessario che definisca un costruttore in grado di ricevere dall'esterno le opzioni di configurazione:

```
public class MotoGPContext: DbContext
{
    public MotoGPContext(DbContextOptions options): base(options) {}

    public DbSet<Pilota> Piloti { get; set; }
    public DbSet<Moto> Moto { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {...}
}
```

Nota bene: l'implementazione del metodo `OnConfiguring()` non è più necessaria.

9.3.2 Modifica dei controller

Infine, nei *controller* che richiedono l'uso del *context* è necessario aggiungere un parametro al costruttore: sarà ASP.NET, quando crea il *controller*, a passargli il *context* come argomento:

```
public class HomeController : Controller
{
    MotoGPContext db;

    public HomeController(MotoGPContext db) => this.db = db;
}
```

9.4 Utilizzare il file di impostazioni "appsettings.json"

ASP.NET implementa un sistema di configurazione delle impostazioni che può accedere a diverse sorgenti: file JSON, XML e testo, database, linea di comando, etc.

La sorgente più comune è il file **appsettings.json**, nel quale è possibile creare una sezione apposita per la *stringa di connessione* (è possibile avere più *stringhe di connessione*):

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

  "Logging": {
```

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

Si può accedere alla *stringa di connessione* usando il metodo `GetConnectionString()`:

```
using MotoGPSite.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

string cnStr = builder.Configuration.GetConnectionString("stringaConnessione");

builder.Services.AddDbContext<MotoGPContext>(options => options.UseSqlServer(cnStr));

builder.Services.AddControllersWithViews();

var app = builder.Build();
...
```

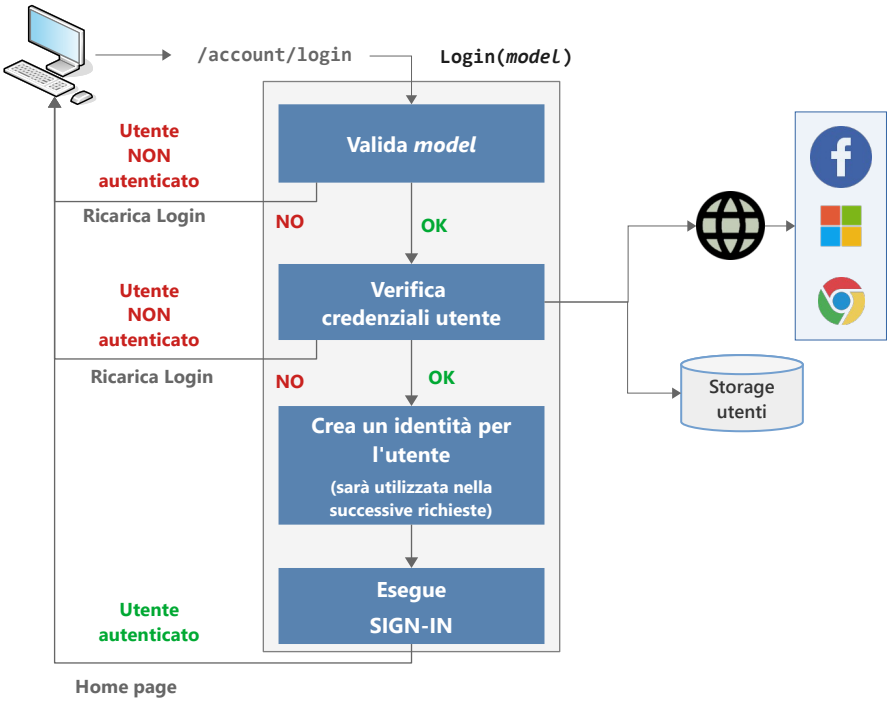
10 Autenticazione

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

- L'utente deve essere stato precedentemente registrato.[5]
- 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*)

10.1 Processo di autenticazione

In figura è schematizzato a grandi linee il processo di autenticazione:



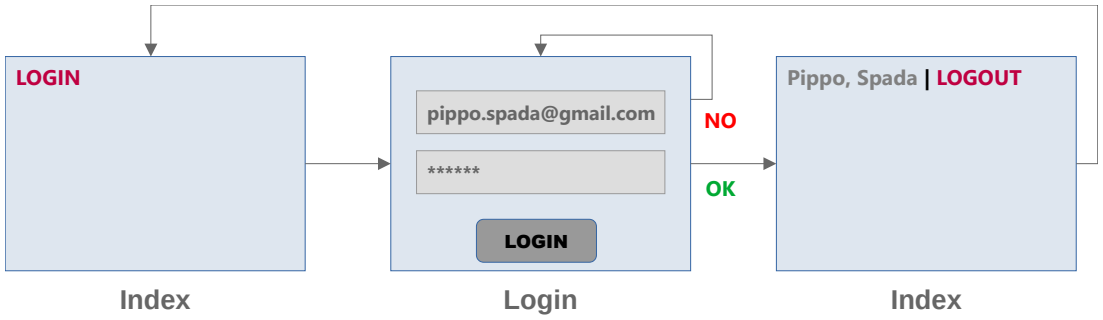
Un utente *anonimo* accede a una pagina di *login* e fornisce le proprie credenziali, che si suppongono già registrate presso l'applicazione, oppure presso un servizio esterno. Dopo la verifica delle credenziali, l'applicazione crea un'*identità* per l'utente e con essa esegue il *sign-in*, completando l'autenticazione. Ogni successiva richiesta farà riferimento all'*identità* creata e sarà riconosciuta dall'applicazione come proveniente da quell'utente in particolare.

Di seguito implementerò questo processo in una applicazione dimostrativa.

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

10.2 Struttura generale dell'applicazione

L'applicazione definisce due *view*.



Index si limita a visualizzare un *link* alla funzione di *login*, oppure il nome dell'utente già autenticato e un *link* alla funzione di *logout*.

La *view* **Login** implementa l'input delle credenziali e avvia il processo di autenticazione. Se l'autenticazione ha successo, l'utente verrà nuovamente indirizzato a **Index**, in caso contrario l'utente dovrà ripetere l'inserimento delle credenziali.

Utilizzando il *link* **LOGOUT**, l'utente ritornerà anonimo e sarà reindirizzato nuovamente a **Index**.

10.2.1 Memorizzare le credenziali degli utenti

Trattandosi di un'applicazione dimostrativa, le credenziali degli utenti sono gestite mediante un database SQL Server popolato con alcuni utenti ad ogni avvio dell'applicazione.

(Vedi tutorial **Entity Framework: generazione automatica del database**.)

Il database contiene la sola tabella **Utenti**, generata automaticamente a partire dalla classe `Account`:

```
public class Account
{
    public int AccountId { get; set; }          // -> AccountId  INT IDENTITY   NOT NULL
    public string Mail { get; set; }            // -> Mail       NVARCHAR (MAX) NOT NULL
    public string Nome { get; set; }            // -> Nome       NVARCHAR (MAX) NOT NULL
    public string Cognome { get; set; }         // -> Cognome    NVARCHAR (MAX) NOT NULL
    public string Password { get; set; }        // -> Password   NVARCHAR (MAX) NOT NULL

    public string Nominativo => $"{Cognome}, {Nome}";
}
```

10.3 Configurare il servizio di autenticazione

La funzione di autenticazione dev'essere configurata, altrimenti il processo sopra descritto non può essere eseguito. ASP.NET implementa un servizio di autenticazione sofisticato e personalizzabile; di seguito lo utilizzerò nella forma più semplice, configurando l'autenticazione basata su *coockie* e senza alcuna personalizzazione.

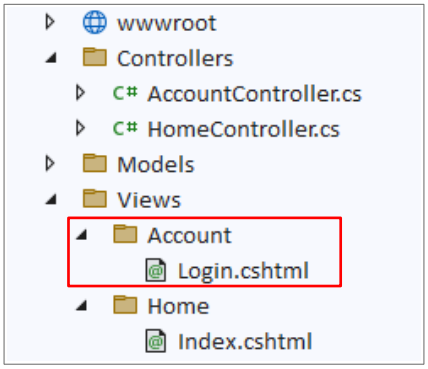
```
var builder = WebApplication.CreateBuilder(args);

string schema = CookieAuthenticationDefaults.AuthenticationScheme;
builder.Services.AddAuthentication(schema).AddCookie();

builder.Services.AddControllersWithViews();
...
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();
...
```

Le righe di codice evidenziate servono ad aggiungere la funzione di autenticazione e ad attivarla.

10.4 Funzioni di *login* e *logout*: AccountController

Il sistema di autenticazione è basato su convenzioni, le quali che possono comunque essere modificate. Di default, ASP.NET richiede la definizione dei metodi `Login()` e `Logout()` nella classe `AccountController`.

Sempre nel rispetto delle convenzioni, la *view* **Login** deve essere collocata nella sotto cartella **Account** della cartella **Views**:



Segue lo scheletro della classe `AccountController`:

```
public class AccountController : Controller
{
    UtentiContext db = new UtentiContext();

    [HttpGet]
    public IActionResult Login()
    {
        return View(new LoginViewModel());
    }

    [HttpPost]
    public IActionResult Login(LoginViewModel vm)   { ... }
    public IActionResult Logout()
    {
```

```
        // ... esegue il sign-out dell'utente
        return RedirectToAction("Index", "Home"); lo reindirizza alla view Index dell'Home...
    }
}
```

La classe non svolge ancora alcuna funzione, ma presenta già alcuni elementi importanti:

- Per l'input delle credenziali viene usato un *viewmodel*: `LoginViewModel`. (8)
- `Login()` passa alla *view* un *viewmodel* vuoto, in modo che la *view* utilizzi l'*attributo* `[Display]` per visualizzare i *placeholder*. (10.6.1)
- Il metodo `Logout()` esegue il *sign-out* dell'utente (da implementare) e lo reindirizza alla *view* **Index** dell'`HomeController`.

10.5 Home page: view Index

Nella versione iniziale contiene soltanto un *link* alla funzione di *login*:

```
@{
    ViewData["Title"] = "Home";
}
<div>
    <a asp-controller="Account" asp-action="Login" class="login">LOGIN</a>
</div>
```

Successivamente dovrà essere implementata la visualizzazione del nome dell'utente autenticato e del *link* alla funzione di *logout*.

10.6 Input delle credenziali

Occorre definire il *viewmodel* e la *view* **Login**.

10.6.1 LoginViewModel

Questa classe usa gli *attributi* per implementare correttamente l'input e la validazione:

```
public class LoginViewModel
{
    [Required(ErrorMessage = "*")]
    [Display(Prompt = "Mail")]
    public string Mail { get; set; }

    [Required(ErrorMessage = "*")]
    [Display(Prompt = "Password")]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

L'attributo `[DataType(DataType.Password)]` fa sì che il tag di **input** corrispondente sia di tipo **password**.

10.6.2 View Login

È un form di input con due caselle di testo e i tag necessari per visualizzare gli errori:

```
@{
    ViewData["Title"] = "Login";
}
@model LoginViewModel
<div>
    <form asp-controller="Account" asp-action="Login" method="post">

        <div asp-validation-summary="ModelOnly"></div>

        <input asp-for="Mail" />
        <span asp-validation-for="Mail" class="error-text"></span>

        <input asp-for="Password" />
        <span asp-validation-for="Password" class="error-text"></span>

        <input type="submit" value="LOGIN" />
    </form>
</div>
```

10.7 Implementare il processo di autenticazione

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

1. Validare i dati inseriti.
2. Verificare se le credenziali corrispondono a un utente registrato.
3. Eseguire il ***sign-in***.

```
[HttpPost]
public IActionResult Login(LoginViewModel vm)
{
    if (!ModelState.IsValid)
    {
        return View(vm);
    }
    var utente = db.Utenti
                .SingleOrDefault(u => u.Mail == vm.Mail && u.Password == vm.Password);
    if (utente == null)
    {
        ModelState.AddModelError("", "Mail o password sconosciuti");
        return View(vm);
    }

    SignIn(utente);

    return RedirectToAction("Index", "Home");
}
```

Il processo di *sign-in* crea un'identità per l'utente:

```
void SignIn(Account utente)
{
 1  string schema = CookieAuthenticationDefaults.AuthenticationScheme;

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

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

Il metodo `SignIn()`:[6]

1. Stabilisce il tipo di autenticazione basata su *cookie*. (La stringa `schema` memorizza lo stesso valore definito in fase di configurazione (10.3).
2. Crea un'*identità* per l'utente. A questa *identità* associa il nome completo dell'utente, memorizzato nel *record* `user`.
3. Esegue il *sign-in*: ogni successiva richiesta dell'utente viene associata all'*identità* precedentemente creata.

10.7.1 Identità e attestazioni

Un'*identità* (`ClaimsIdentity`) si basa su una o più *attestazioni* (`Claim`), che sono coppie *chiave-valore* contenenti informazioni sull'utente. A un utente può essere associata più di un'*identità*, una delle quali è considerata la principale.

Nel codice relativo al punto 2) creo una sola *identità*, basta su una sola *attestazione*, rappresentata dal nome completo dell'utente.

10.7.2 Oggetto HttpContext

`HttpContext` è una proprietà del *controller* che fornisce l'accesso a un oggetto contenente dati e servizi relativi alla richiesta in corso e. L'oggetto è accessibile, con il nome `Context`, anche nelle *view*. (10.9)

10.8 Logout

Il *logout* si riduce all'invocazione di un metodo che esegue il *sign-out* dell'utente ed elimina l'*identità* che gli era stata assegnata nel *sign-in*:

```
public IActionResult Logout()
{
    string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
    HttpContext.SignOutAsync(scheme).Wait();
    return RedirectToAction("Index", "Home");
   }
}
```

Dopo l'esecuzione di `SignOutAsync()`, le successive richieste dell'utente non sono più associate all'*identità*

6 Il metodo **HttpContext.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; lo uso soltanto per semplificare il codice.

precedentemente creata: l'utente è ritornato anonimo.

10.9 Home page: visualizzazione dello stato dell'utente

Ora è possibile modificare **Index** per visualizzare l'utente autenticato. A questo scopo si ottiene innanzitutto l'*identità* dell'utente e, se è autenticato, si visualizza il suo nome.

```
@{
    ViewData["Title"] = "Home";
    var identità = Context.User.Identity;
}

<div>
    @if (identità.IsAuthenticated)
    {
        <span>@identità.Name | </span>
        <a asp-controller="Account" asp-action="logout" class="login">LOGOUT</a>
    }
    else
    {
        <a asp-controller="Account" asp-action="Login" class="login">LOGIN</a>
    }
</div>
```

Alcune considerazioni:

- `Context` referenzia lo stesso oggetto `HttpContext` usato nel *controller*. (10.7.2)
- La proprietà `Name` memorizza il nome dell'utente impostato in `SignIn()` dall'istruzione:

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

11 Autorizzazione

L'*autorizzazione* stabilisce quali risorse sono accessibili, o negate, agli utenti in base alla loro *identità*. È una funzione che dipende dall'*autenticazione*.

L'autorizzazione si applica a un *metodo action* o a un intero *controller* e può essere:

- **Semplice**: è concessa sul fatto che l'utente sia *autenticato* o *anonimo*.
- **Basata su *attestazioni*** (*claim*): l'utente è autorizzato se le *attestazioni* contenute nella sua *identità* soddisfano determinati criteri.

  Ad esempio, un utente può essere autorizzato o meno ad accedere a determinate risorse sulla base dall'età.
- **Basata sui *ruoli***: l'utente è autorizzato in base al ruolo che gli è stato assegnato nell'applicazione.

  Ad esempio, in una biblioteca virtuale i tesserati potranno soltanto consultare il catalogo dei libri e prenotare dei prestiti. Il personale amministrativo avrà la facoltà di accedere alle funzioni dedicate all'inserimento e modifica dei contenuti.

11.1 Abilitare la funzione di autorizzazione

La funzione deve essere esplicitamente attivata in **Program**:

```
var builder = WebApplication.CreateBuilder(args);

string schema = CookieAuthenticationDefaults.AuthenticationScheme;
builder.Services.AddAuthentication(schema).AddCookie();

builder.Services.AddControllersWithViews();
...
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();
...
```

11.2 Autorizzazione semplice

Nell'autorizzazione semplice ci si limita a stabilire quali risorse sono accessibili/negate agli utenti anonimi uutilizzando l'attributo `[Authorize]`: soltanto gli utenti autenticati possono accedere a risorse gestite da metodi e *controller* decorati con `[Authorize]`.

Se si vuole negare l'accesso alla maggior parte dei *metodi action* di un *controller*, ma si vuole concedere l'accesso per alcuni metodi, si può applicare `[Authorize]` al *controller* e decorare i metodi accessibili con l'attributo `[AllowAnonymous]`.

11.3 Implementare l'autorizzazione semplice

Partendo dal progetto realizzato nel capitolo precedente, intendo realizzare un'applicazione che comprende tre *view*: **Index**, **Login**, **Autenticati**. Le prime due sono accessibili a tutti[7], mentre la terza soltanto agli utenti che hanno eseguito il *login*.

(Rispetto alla versione precedente versione, è la *layout view* a mostrare lo stato dell'utente e i *link* alle pagine del sito.)

11.4 Autorizzare l'accesso alla view Autenticati

È sufficiente decorare il *metodo action* con `[Authorize]`:

```
public class HomeController : Controller
{
    ...
    [Authorize]
    public IActionResult Autenticati()
    {
       return View();
    }
}
```

Se un utente anonimo tenta di accedere all'URL **/home/autenticati**, cliccando sul *link* o digitando l'URL nella barra degli indirizzi, sarà rediretto automaticamente alla *view* **Login**.

11.4.1 Autorizzare il logout

È importante comprendere che la funzione di autorizzazione non riguarda le *view*, ma gli URL e dunque i *metodi action* e i *controller*. Quindi è legittimo consentire l'esecuzione del metodo `Logout()` soltanto agli utenti autenticati:

```
public class AccountController : Controller
{
    ...
    [Authorize]
    public IActionResult Logout() { ... }
    }
}
```

11.5 Indirizzare l'utente autenticato alla risorsa richiesta

Quando l'utente naviga a una risorsa che richiede autenticazione, dopo essersi autenticato dovrebbe essere rediretto alla risorsa richiesta e non alla *home page*.

A questo scopo è possibile sfruttare l'URL generato da ASP.NET quando ridireziona l'utente anonimo alla funzione di *login:* infatti, all'URL **/account/login** viene aggiunta automaticamente una *query string* contenente l'URL richiesto dall'utente.

7 Sorvolo sul fatto che rendere accessibile la *view* **Login** agli utenti già autenticati non ha molto senso.

Ad esempio, se l'utente tenta di accedere alla *view* **Autenticati**, ASP.NET lo reindirizza a:

```
http://localhost:5000/Account/Login?ReturnUrl=/home/autenticati
```

Per utilizzare **ReturnUrl** e reindirizzare l'utente alla risorsa richiesta occorre implementare un processo che vede la "collaborazione" tra i due metodi `Login()` e la *view* di **Login**.

Innanzitutto, il metodo `Login()` che carica la *view* deve definire un parametro `returnUrl`. Questo è soltanto il primo passo, però. Infatti, è il metodo `Login()` che carica il form a ricevere il **ReturnUrl**, ma è il metodo `Login()` che elabora il form a doverlo utilizzare per reindirizzare l'utente. Dunque, occorre implementare un meccanismo che consenta al primo metodo di passare **ReturnUrl** al secondo:

```
[HttpGet]
public IActionResult Login(string returnUrl)
{
    ...
}

[HttpPost]
public IActionResult Login(LoginViewModel vm)
{
    ...
    // ridireziona a ...
}
```

In un'applicazione desktop non ci sarebbero problemi: basterebbe memorizzare il **ReturnUrl** in una variabile globale della classe, ma in ASP.NET i due metodi sono eseguiti in momenti diversi da due diverse istanze dell'`AccountController`, dunque questa soluzione non è praticabile.

L'approccio standard è memorizzare l'URL nel form di *login* inviato al browser. In pratica, l'applicazione passa il **ReturnUrl** al client, il quale lo "ripassa" all'applicazione all'invio del form: il client funge da "intermediario" per il passaggio di un valore tra due metodi del *controller*.

11.5.1 Memorizzare il ReturnUrl nel viewmodel

È la soluzione più semplice:

```
public class LoginViewModel
{
    [Required(ErrorMessage = "*")]
    [Display(Prompt = "Mail")]
    public string Mail { get; set; }

    [Required(ErrorMessage = "*")]
    [Display(Prompt = "Password")]
    [DataType(DataType.Password)]
    public string Password { get; set; }

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

Nell'`AccountController` occorre modificare il metodo che carica il form:

```
[HttpGet]
public IActionResult Login(string returnUrl)
{
    return View(new LoginViewModel { ReturnUrl = returnUrl });
}
```

11.5.2 Inviare il ReturnUrl al submit del form

Quando l'utente conferma un form, il browser invia i dati inseriti nei tag di input. Esiste però un secondo meccanismo che consente a un form di inviare dei dati: usare una *query string* nell'attributo **action**.

È sufficiente specificare il **ReturnUrl** con il *tag helper* appropriato:

```
...
<form method="post"
    asp-controller="Account" asp-action="Login" asp-route-returnUrl="@Model.ReturnUrl" >
                    (1)                (2)                              (3)

    ...
</form>
...
```

ASP.NET traduce i tre *tag helper* in:

```
<form method="post" action="/Account/Login?returnUrl=/home/autenticati">
                            (1)     (2)          (3)
    ...

</form>
```

11.5.3 Rendirizzare l'utente all'URL richiesto

Infine, è sufficiente una semplice modifica al metodo `Login()` che elabora il form:

```
[HttpPost]
public IActionResult Login(LoginViewModel vm)
{
    ...
    SignIn(utente);

    //return RedirectToAction("Index", "Home");
    return Redirect(vm.ReturnUrl);
}
```

Nota bene: occorre usare il metodo `Redirect()`, non `RedirectToAction()`, poiché `vm.ReturnUrl` memorizza un URL completo.

11.6 Autorizzazione basata su ruoli

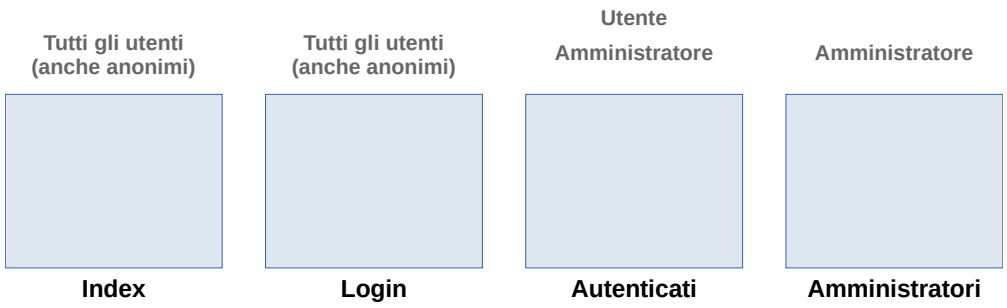
L'autorizzazione basata sui ruoli parte dalla premessa che gli utenti appartengano a uno o più ruoli, i quali sono utilizzati come discrimine per stabilire se concedere o negare l'accesso a determinate funzioni.

Ad esempio, agli utenti di un registro scolastico elettronico è assegnato almeno un ruolo: **alunno**, **genitore**, **docente**, **coordinatore di classe**, **amministratore**. Un utente, in base al proprio ruolo, può utilizzare funzioni del registro che sono negate ad altri utenti.

Partendo dal progetto precedente, aggiungo un ruolo – **Utente** o **Amministratore** – alla classe `Account` e dunque nella tabella **Utenti**:

```
public class Account
{
    public int AccountId { get; set; }          // -> AccountId  INT IDENTITY   NOT NULL
    public string Mail { get; set; }            // -> Mail       NVARCHAR (MAX) NOT NULL
    public string Nome { get; set; }            // -> Nome       NVARCHAR (MAX) NOT NULL
    public string Cognome { get; set; }         // -> Cognome    NVARCHAR (MAX) NOT NULL
    public string Password { get; set; }        // -> Password   NVARCHAR (MAX) NOT NULL
    public string Ruolo { get; set; }           // -> Ruolo      NVARCHAR (MAX) NOT NULL
    public string Nominativo => $"{Cognome}, {Nome}";
}
```

Al sito viene aggiunta la pagina **Amministratori**, accessibile soltanto per gli utenti con ruolo amministrativo. La figura riepiloga le pagine del sito e gli utenti che possono accedervi:



11.7 Aggiungere l'attestazione "ruolo" all'identità dell'utente

Il ruolo è un'*attestazione* (*claim*) sull'utente (10.7.1); pertanto deve essere integrato nella *identità* dell'utente durante il processo di *sign-in* (10.7).

Questo processo è implementato nel metodo `SignIn()` dell'`AccountController`:

```
void SignIn(Account utente)
{
    string scheme = CookieAuthenticationDefaults.AuthenticationScheme;

    var identity = new ClaimsIdentity(scheme);
    identity.AddClaim(new Claim(ClaimTypes.Name, utente.Nominativo));
    identity.AddClaim(new Claim(ClaimTypes.Role, utente.Ruolo));
    var principal = new ClaimsPrincipal(identity);

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

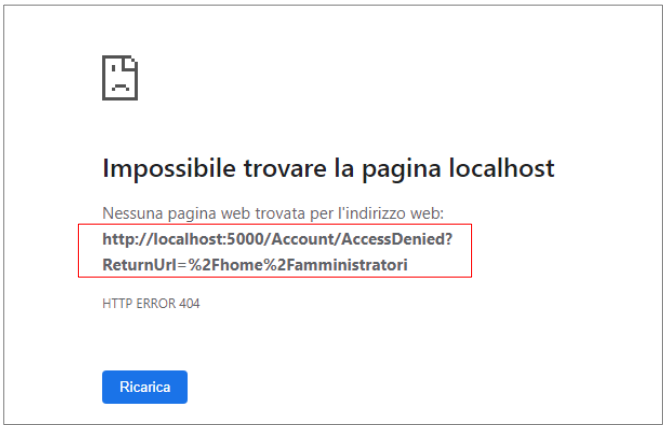
11.8 Autorizzare in base al ruolo

Si usa l'attributo `[Autorize]` specificando il ruolo (o i ruoli) per i quali la funzione è accessibile:

```
public class HomeController : Controller
{
    public IActionResult Index() // accessibile a tutti gli utenti (anche anonimi)
    {
        return View();
    }

    [Authorize]
    public IActionResult Autenticati() // accessibile ai soli utenti autenticati
    {
        return View();
    }

    [Authorize(Roles = "Amministratore")]
    public IActionResult Amministratori() // accessibile agli amministratori
    {
        return View();
    }
}
```

(Nel caso si dovessero specificare più ruoli, sarebbe necessario separarli con la virgola all'interno della stessa stringa.)

Un tentativo da parte di un utente non amministratore di accedere all'URL **/home/amministratori** produce il seguente risultato:



11.9 Implementare la funzione "accesso negato"

Dalla figura precedente si nota come nel negare l'accesso a una risorsa, ASP.NET reindirizza l'utente all'URL **/Account/AccessDenied**, completo di una *query string* contenente l'URL della risorsa negata. È necessario gestire questo URL, visualizzando un messaggio appropriato e consentendo all'utente di autenticarsi.

Di seguito fornisco una versione di base della *view* **AccessDenied**, che visualizza un messaggio e il nome dell'utente che ha tentato di accedere alla risorsa:

```
@{
    ViewData["Title"] = "Accesso negato";
}

<div>
    <h3 class="error-text">ACCESSO NEGATO</h3>
    <p>Si può accedere a questa pagina solo come 'amministratore''</p>

    <p>@User.Identity.Name</p>
</div>
```

Naturalmente è necessario implementare un *metodo action* nell'`AccountController`:

```
public class AccountController : Controller
{
    ...
    public IActionResult AccessDenied()
    {
        return View();
    }
}
```

11.10 Reindirizzare l'utente al login

Un'implementazione migliore della *view* prevede un *link* per accedere al *login*, *link* che comprende il **ReturnUrl** generato da ASP.NET quando ha negato l'accesso. In questo modo se l'utente si autentica come amministratore sarà automaticamente reindirizzato alla pagina che aveva richiesto.

A questo scopo è necessario passare alla view il **ReturnUrl**:

```
public IActionResult AccessDenied(string returnUrl)
{
    return View("AccessDenied", returnUrl);
}
```

La *view* lo usa nella *query string* del *link* che indirizza alla funzione di *login*:

```
@{
    ViewData["Title"] = "Accesso negato";
}

@model string   // è il ReturnUrl
<div>
    <h3 class="error-text">ACCESSO NEGATO</h3>
    <h4>Si può accedere a questa pagina solo come 'amministratore''</h4>

    <p>@User.Identity.Name</p>

    <a asp-controller="Account" asp-action="Login"
                    asp-route-returnurl="@Model">Login come amministratore</a>
</div>
```

Nota bene: poiché il *model* è una stringa, è necessario usare una versione del metodo `View()` che accetta come primo parametro il nome della *view* e come secondo il *model*. (Se venisse specificato il solo *model*, ASP.NET lo interpreterebbe come il nome della *view* da caricare.)

Appendice I: Proprietà *nullabili* e NRT

I tipi del linguaggio C# possono essere suddivisi in due categorie:

- I *reference types*: sono tipi *nullabili*; le variabili possono contenere `null` e dunque non memorizzare alcun valore.
- I *value types*: sono tipi *non-nullabili*; le variabili contengono sempre un valore, dunque non possono contenere `null`.

Esempio principe della prima categoria è il tipo `string`; esempio principe della seconda è il tipo `int`. Quanto detto ha delle implicazioni, che riassumo nel seguente codice:

```
string nome = null;   // OK
int altezza = null;   // Errore
...
class Persona
{
    public string Nominativo;  // -> null (valore predefinito)
    public int Età;            // -> 0    (valore predefinito)
}
```

Nel 2006 è stato introdotta una nuova categoria di tipi che unisce le caratteristiche dei *reference type* e dei *value type*: i ***nullable value types***. Questi si ottengono a partire dai *value type* con l'aggiunta del simbolo `?`:

```
string nome = null;   // OK
int altezza = null;   // Errore
int? età = null;      // OK
...
class Persona
{
    public string Nominativo;  // -> null (valore predefinito)
    public int Età;            // -> 0    (valore predefinito)
    public int? altezza;       // -> null (valore predefinito)
}
```

Nel 2019 sono stati introdotti i ***N**ullable **R**eference **T**ypes*. Non si tratta di una nuova categoria di tipi, poiché i *reference types* sono già *nullabili*; è una funzione del linguaggio che consente di creare una distinzione tra *reference types nullabili* e *reference types non-nullabili*.

L'idea è quella di poter dichiarare variabili di per sé *nullabili*, però con la garanzia che non contengano `null`. Ci pensa il linguaggio, analizzando il programma, eseguendo una *null reference analysis*, a segnalare il codice che non soddisfa il vincolo di *non-nullabilità*.

Per implementare questa funzione, invece di aggiungere un nuovo simbolo ai tipi *nullabili*, è stato deciso di allineare la sintassi alla scelta fatta nel 2005. In sintesi, per i *reference types*:

- L'uso del simbolo `?` indica che una variabile è *nullabile*.
- L'assenza del simbolo `?` indica che la variabile, pur appartenente a un tipo *nullabile*, si intende *non-nullabile*.

Il codice seguente riassume la situazione:

```
string nome = null;       // Avvertimento: vincolo non-nullabilità violato!
string? telefono = null; // OK
int altezza = null;       // Errore
int? età = null;          // OK
...
class Persona
{
    public string Nominativo;  // -> null (Avvertimento: vincolo non-nullabilità violato!)
    public string? Email;      // -> null (OK)
    public int Età;            // -> 0    (OK)
    public int? altezza;       // -> null (OK)
}
```

Nota bene:

- La *non-nullabilità* (assenza del simbolo `?`) viene violata non soltanto assegnando `null`, ma anche dichiarando una variabile *non-nullabile* senza assegnarle un valore.
- La violazione del vincolo non rappresenta un errore di programmazione: il compilatore emette un *warning* (avvertimento).

11.1 ASP.NET e *Nullable Reference Types*

La gestione della *nullabilità* introdotta con i NRT ha un impatto anche su ASP.NET, soprattutto sulla funzione di *validazione del model* usata nella gestione dei form inviati dal client. (7).

Durante l'esecuzione di questa funzione, ASP.NET esegue il *databinding* tra i campi del form e il parametro o i parametri del metodo che deve elaborarlo. Prima dell'introduzione dei NRT, questo processo considerava i *reference types* come *nullabili* e i *value types* come *non-nullabili*. Con l'introduzione dei NRT, anche i *reference types* non decorati con `?` sono considerati *non-nullabili*.

Per comprendere le implicazioni, considera la funzione di inserimento di una persona. Seguono il *model*, *view* e *controller*:

Model

```
public class Persona
{
    public string Nome { get; set; }
    public int Età { get; set; }
}
```

View

```
<form method="POST" action="/home/nuovapersona">
    <input asp-for="Nome" placeholder="Nome" />
    <input asp-for="Età" placeholder="Età" />
    <input type="submit" value="INSERISCI" />
</form>
```

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

    [HttpPost]
    public IActionResult Nuovapersona(Persona p)
```

```
    {
        if (ModelState.IsValid)
        {
            return RedirectToAction("Index");
        }
        return View(p);
    }
}
```

Il risultato restituito dalla proprietà `IsValid` dipende dal fatto che la funzione NRT sia abilitata oppure no.

11.1.1 Funzione NRT disabilitata (di default in .NET 5 o versioni precedenti)

La proprietà `Nome` della persona viene considerata *nullabile*, dunque l'input del nome non è obbligatorio. Al contrario, la proprietà `Età`, essendo un *value type*, è *non-nullabile*; dunque l'input dell'età è obbligatorio.

11.1.2 Funzione NRT abilitata (di default in .NET 6 o versioni successive)

La proprietà `Nome` della persona viene considerata *non-nullabile*, dunque l'input del nome è obbligatorio. Per la proprietà `Età` non cambia nulla.

11.2 Intervenire sulla nullabilità dei campi (obbligatorietà dell'input)

Se la funzione NRT è disabilitata ma si desidera rendere obbligatorio un campo *nullabile*, si può usare l'attributo `[Required]`. Al contrario, se si vuole rendere facoltativo un campo di tipo *non-nullabile*, si può dichiararlo *nullabile*:

```
public class Persona
{
    [Required]
    public string Nome { get; set; }   Obbligatorio
    public int? Età { get; set; }      Non obbligatoria
}
```

Se la funzione NRT è attivata, i campi di tipo *reference type* sono considerati obbligatori, dunque non esiste alcun bisogno di usare `[Required]`. (Ma lo si può usare per impostare il messaggio di errore).

```
public class Persona
{
    public string Nome { get; set; }   Obbligatorio
    public int Età { get; set; }       Obbligatoria
}
```

Al contrario, se si desidera rendere facoltativo un campo, basta decorarlo con ?:

```
public class Persona
{
    public string? Nome { get; set; }   Non è obbligatorio
    public int Età { get; set; }        Obbligatoria
}
```

La funzione NRT non influenza in alcun modo il comportamento dei *value types*.