APPLICAZIONI WEB

BLAZOR - Server Side Rendering

Indice generale

CONCETTI GENERALI

1 Applicazioni e applicazioni distribuite

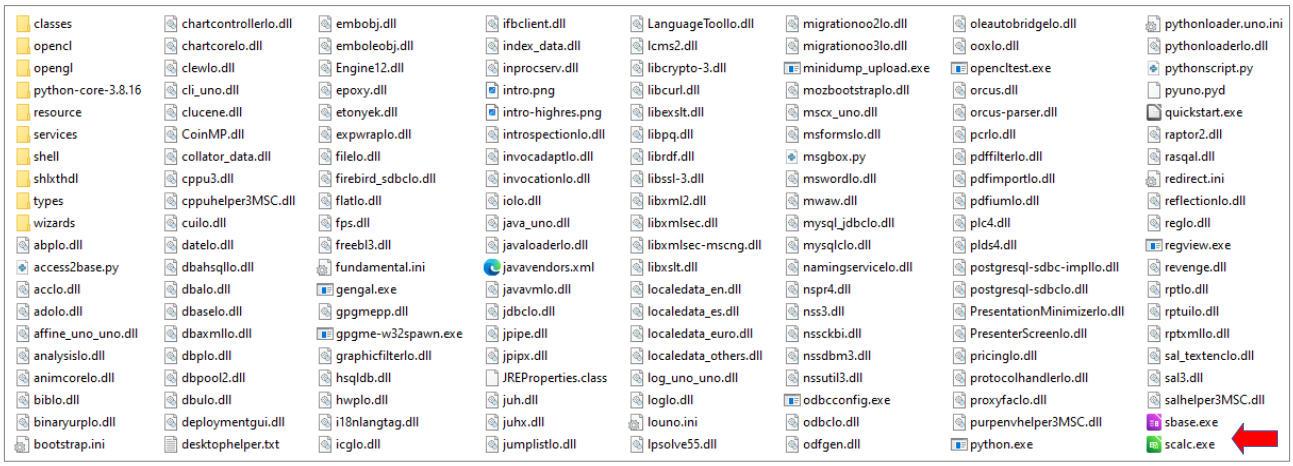
Questo capitolo introduce la terminologia e i concetti generali legati al funzionamento delle *applicazioni web*. Innanzitutto, però, è necessario chiarire il significato del termine *applicazione*.

1.1 Applicazioni

Un'applicazione è un software dedicato alla produttività dell'utente: scrivere, disegnare, programmare, etc. Non tutti i software sono applicazioni; non lo sono il *sistema operativo*, i *device driver*, in generale i programmi usati per il funzionamento e la gestione del computer.

Un'applicazione ha per definizione un'interfaccia utente e può essere composta da vari *artefatti software*: programmi eseguibili, librerie, script, file di configurazione e risorse necessarie al suo funzionamento.

La figura mostra un estratto della cartella **C:\Program Files\LibreOffice\Program**, dove è installato **Libre Office**.



Le freccia rossa in basso a destra indica il file **scalc.exe**, il programma eseguibile di **Calc**. Il file **scalc.exe**, però, non è un'*applicazione*, ma soltanto parte di essa; il funzionamento di **Calc** richiede molti dei file e delle cartelle mostrati in figura, compresi gli script di *python* (estensione **py**), e i file di configurazione (estensione **ini**).

1.1.1 Applicazioni standalone

Calc, come **Excel**, **Writer**, **Word**, **PhotoShop** e altri software di produttività personale, è un'*applicazione standalone*, o *applicazione desktop*. Un'*applicazione standalone* non necessita di risorse remote per il proprio funzionamento, dunque non richiede una connessione di rete.

In conclusione: tutti gli *artefatti software* necessari al suo funzionamento sono presenti nel sistema dove è installata

1.1.2 Applicazioni standalone che gestiscono più processi

Normalmente l'esecuzione di un programma produce un singolo processo, ma non è necessario che sia così. Alcune applicazioni producono l'esecuzione di vari processi, che possono svolgere funzioni distinte, oppure replicare la stessa funzione con lo scopo di aumentare la capacità di elaborazione dell'applicazione.

Nelle *applicazioni standalone* tutti i processi sono in esecuzione sullo stesso sistema.

1.1.3 Applicazioni standalone che accedono alla rete

Le *applicazioni standalone* non necessitano di risorse remote, ma possono usare Internet per svolgere varie funzioni:

- Verificare l'esistenza di aggiornamenti ed eventualmente scaricarli e installarli.
- Scaricare e utilizzare risorse on-line.
- Accedere agli "aiuti" sulle funzioni dell'applicazione. Etc.

Tutto ciò non cambia la loro natura *standalone*, dunque la capacità di funzionare autonomamente.

1.2 Applicazioni distribuite

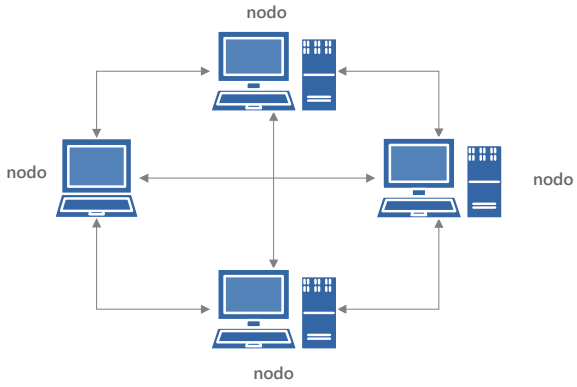
Un'*applicazione distribuita*, o *applicazione di rete*, è un software in esecuzione su più computer. Gli *artefatti software* sono eseguiti come processi ospitati in sistemi distinti e comunicano tra loro attraverso la rete.

Le *applicazioni distribuite* si distinguono per la loro architettura, che ricade normalmente in due categorie:

- Architetture *peer-to-peer* (**P2P**).
- Architetture *client-server*.

1.2.1 Applicazioni peer-to-peer

Nelle architetture P2P le funzioni dell'applicazione sono replicate in più computer. Ogni computer, definito *nodo dell'applicazione*, esegue lo stesso software e può comunicare con tutti gli altri.



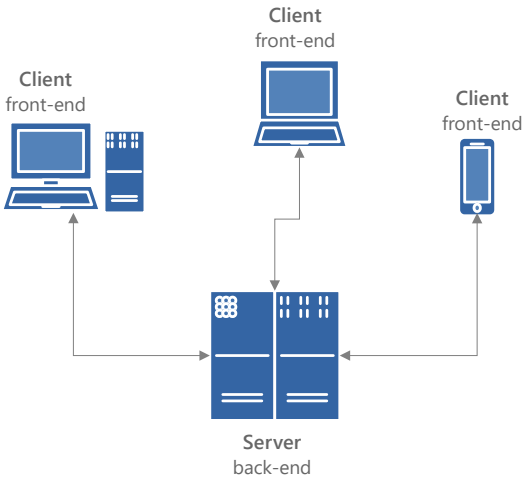
Le applicazione P2P sono così chiamate perché non esiste un *nodo dell'applicazione* speciale rispetto agli altri. Ogni *nodo* fornisce dei servizi agli altri *nodi* e "consuma" i servizi degli altri *nodi*.

Ad esempio, in un software di ***file sharing*** **P2P**, i file da condividere sono "sparsi" nei *nodi dell'applicazione*. Un utente può cercare e scaricare i file indipendentemente dalla loro collocazione. Allo stesso tempo, l'utente rende disponibile il proprio computer per la memorizzazione di file da condividere con gli altri utenti della stessa applicazione.

1.2.2 Applicazioni client-server

Nelle architetture *client-server*, funzioni distinte dell'applicazione vengono eseguite su computer distinti. Il termine *client-server* si riferisce all'esistenza di due funzioni principali:

- Il *client*: implementa l'interfaccia utente e dunque viene usata dall'utente finale. I *client* gestiscono il cosiddetto ***front-end*** dell'applicazione, con il quale interagisce l'utente.
- Il *server*: fornisce i servizi ai *client*. Il *server* implementa il cosiddetto ***back-end*** dell'applicazione.



Nelle applicazioni *client-server*, i servizi e i dati sono collocati sul *server*; i client hanno la sola funzione di consentire all'utente di interagire con l'applicazione.

Ad esempio, in un servizio *client-server* di **memorizzazione file**, tutti i file sono collocati nel *server*, il quale li rende disponibili agli utenti attraverso i *client* dell'applicazione.

1.3 Applicazioni standalone vs applicazioni distribuite

La differenza architetturale tra *applicazioni standalone* e *applicazioni distribuite* riguarda vari aspetti, tra i quali il livello di interattività che è possibile implementare, dunque la qualità dell'*esperienza utente* nell'usare l'applicazione.

Nelle *applicazioni standalone* il codice che gestisce le azioni dell'utente viene eseguito nello stesso processo che fornisce le risposte a quelle azioni. Ciò consente una notevole interattività, poiché la latenza tra l'azione dell'utente e la risposta dell'applicazione è minima, se non nulla. È dunque possibile implementare delle cosiddette ***rich user interface***, interfacce utente che implementano molte funzioni, con un alto livello di interattività e un'ottima *esperienza utente*.

Nelle *applicazioni distribuite*, le azioni dell'utente vengono gestite nel *client* e comunicate al *server*, il quale restituisce al *client* i risultati. La latenza tra un'azione dell'utente e la risposta dell'applicazione può essere elevata e questo rende difficile, se non addirittura impossibile, implementare efficacemente alcune funzioni tipiche delle interfacce utente.

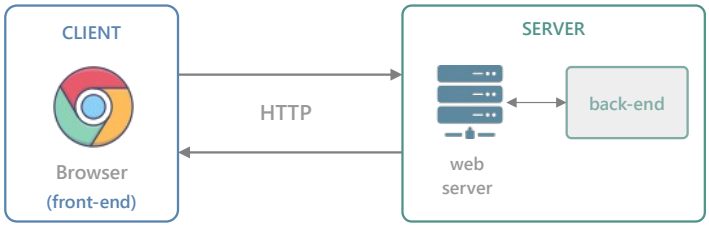
Esistono comunque *applicazioni distribuite* che forniscono dei ***rich client***, in grado di fornire un'*esperienza utente* simile a quella delle *applicazioni standalone*. (2.3)

2 Introduzione alle applicazioni web

Le *applicazioni web* sono:

- *applicazioni distribuite* caratterizzate da un'architettura *client-server*,
- che usano il protocollo HTTP per la comunicazione di rete,
- un *web browser* come *client* e
- il linguaggio HTML per l'interfaccia utente.

La parte di applicazione che funge da *server* (il *back-end*) è collegata a un software chiamato ***web server***, il quale fa da tramite per la comunicazione con il *client*. Tra i *web server* più usati ci sono **Internet Information Services** e **Apache**.



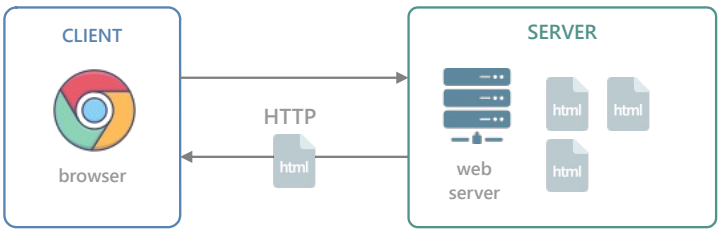
Questa architettura sta alla base delle *applicazioni web*, le quali possono avere soluzioni diverse sull'implementazione del *font-end*: ***server-side rendering*** (SSR) e ***client-side rendering*** (CSR). In queste definizioni, i termini *client-side* e *server-side* (*lato client* e *lato server*) si riferiscono al sistema, *client* o *server*, nel quale viene eseguito il codice di visualizzazione.

Innanzitutto, però, è necessario introdurre l'architettura più semplice in assoluto, che esiste fin dalla nascita del World Wide Web: i *siti web statici*.

2.1 Siti web statici

I *siti web statici* implementano un servizio documentale che consente la navigazione tra i contenuti, memorizzati in pagine HTML.

In un *sito statico*, il *back-end* è rappresentato dal solo *web server*. Il *client* invia una richiesta HTTP contenente l'URL (*Uniform Resource Locator*) di un documento, il *web server* lo carica dal disco e ne spedisce il contenuto al *client*.



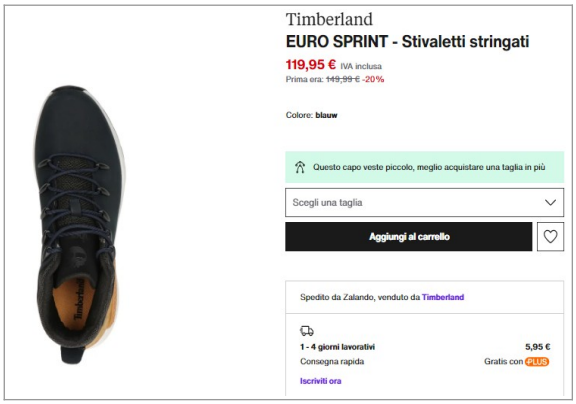
La parola "statico" indica che i contenuti inviati al *client* non vengono elaborati; sono memorizzati nei file e sono sempre gli stessi.

Ciò detto, l'accesso a risorse statiche è una prerogativa anche delle applicazioni che implementano il SSR e CSR. (2.4)

2.2 Server Side Rendering

Nella maggior parte delle *applicazioni web* i contenuti non sono memorizzati in pagine HTML, ma in database, oppure ottenuti da servizi esterni.

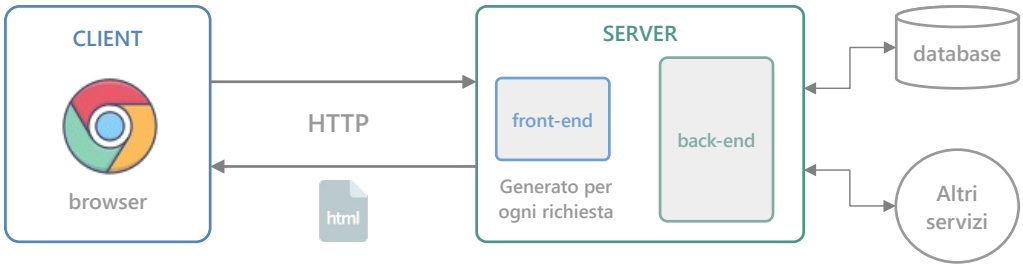
Immagina un sito di *e-commerce*. Quando l'utente accede alla pagina con i dettagli di un prodotto, l'applicazione ottiene i dati da un database e li visualizza mediante codice HTML.



Si parla di *sito dinamico*, perché i contenuti, nella forma finale mostrata all'utente, vengono generati "al volo" sulla base dei dati caricati dal database.

Esistono varie soluzioni per ottenere questo risultato, che dipendono anche dal livello di interattività che si desidera implementare

La soluzione più semplice, largamente adottata in molti siti web, usa il *server-side rendering*. Il codice HTML, comprensivo dei contenuti, viene generato sul *server* e successivamente spedito al *client*. Il browser ha un ruolo completamente passivo: riceve il contenuto HTML e lo visualizza.

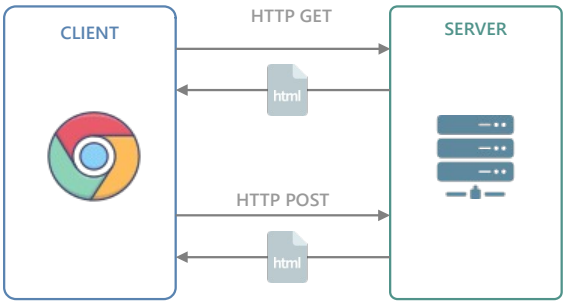


2.2.1 Interattività nelle architetture SSR

Nelle architetture SSR tutto il codice applicativo risiede nel *server*, dunque l'interattività è limitata all'uso di richieste HTTP[1].

Nelle applicazioni che adottano il SSR, l'operatività dell'utente si riduce a:

- Cliccare su un *hyperlink* o digitare un URL sulla barra degli indirizzi: il *client* invia una richiesta HTTP GET al *server*. (3.2)
- Cliccare su un bottone che esegue l'invio di un *form* HTML: il *client* invia una richiesta POST o GET al *server*. (3.3)



Queste azioni producono il caricamento di una nuova pagina HTML, oppure il ricaricamento della stessa pagina con nuovi dati.

1 Tecnicamente è possibile, ed è stato fatto, fornire un alto livello di interattività anche usando il SSR, ma è inefficiente, poiché ad ogni azione dell'utente corrisponde una richiesta/risposta HTTP, con i problemi di latenza e inefficienza che questo comporta.

2.2.2 Applicazioni web stateless

Il modello *richiesta HTTP → risposta HTTP* delle applicazioni SSR, oltre a limitare l'interattività, ha un'altra implicazione: nel gestire la richiesta corrente, il *server* non mantiene memoria delle richieste precedenti e del risultato che hanno prodotto.

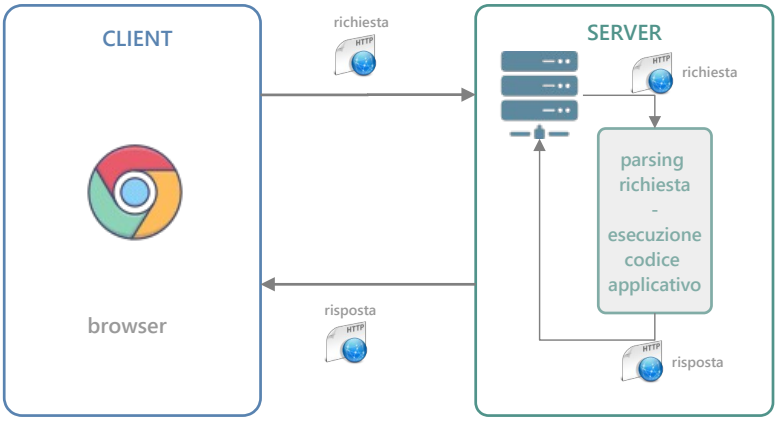
Si parla di comunicazione ***stateless*** (senza memoria); è insita nel protocollo HTTP e pone un limite al tipo di interazione che può essere instaurata tra *client* e *server*. Esistono vari modi per superare questo limite, ma devono essere programmati nel codice applicativo.

2.2.3 Programmazione delle applicazioni SSR

La programmazione *lato server* può essere realizzata con qualsiasi linguaggio, ma richiede un *framework* che fornisca perlomeno le seguenti funzioni[2]:

- Comunicazione con il *web server*: ricevere le richieste HTTP e inviare le risposte HTTP.
- Accesso alle informazioni memorizzate nelle richieste HTTP.
- Scrittura delle risposte HTTP: generazione del codice HTML da inviare al *client*.
- Compilazione ed esecuzione del codice applicativo.
- Creazione e gestione dei *cookie* (3.7).

I *framework* di sviluppo di *applicazioni web* forniscono queste e altre funzioni e consentono di ignorare i dettagli della comunicazione HTTP con il *client*.



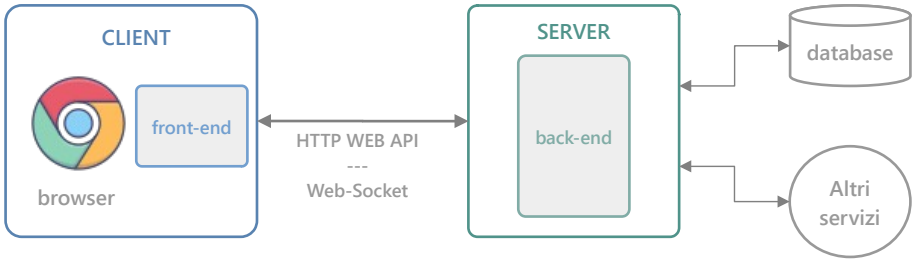
Esistono *framework* per ogni linguaggio, tra i quali: **Laravel** (PHP), **Node.Js** (JavaScript), **Ruby on Rails** (Ruby) e **ASP.NET Core**, sul quale si basa **Blazor**, che utilizza il linguaggio C#.

2.3 Client Side Rendering

Nelle architetture CSR, l'interfaccia utente viene programmata per essere eseguita sul *client*; questo consente di implementare un alto livello di interattività, poiché le azioni dell'utente vengono gestite dal codice in esecuzione nel browser. Queste applicazioni vengono definite anche **RIA** (*Rich Internet Application*), per via della loro interfaccia utente ricca di funzioni.

2 I *framework* moderni forniscono molte altre funzioni, tra le quali: creazione di sessioni utente, autenticazione e autorizzazione degli utenti.

Il *client* comunica con il *server* soltanto per ottenere/modificare i dati gestiti dall'applicazione (2.3.2).



2.3.1 CSR e Simple Page Application

Una soluzione molto comune che adotta il CSR è definita ***Simple Page Application***: il *front-end* è rappresentato da una singola pagina HTML, ottenuta una volta per tutte dal *server* durante il primo collegamento del *client*. Dopodiché, il *client* gestisce l'interfaccia utente modificando direttamente il *Document Object Model* (**DOM**) della pagina, presentando all'utente visualizzazioni diverse.

Anche in questo caso, comunque, pur gestendo localmente le azioni dell'utente, il *client* deve comunicare con il *server*, poiché è nel *back-end* che risiede il codice che ottiene e modifica i dati.

2.3.2 Comunicazione tra client e server mediante web-socket

Un *web-scocket* è una connessione TCP bidirezionale che consente a *client* e *server* di scambiarsi dati in modo efficiente.

Questo tipo di connessione è alla base di un'implementazione del CSR chiamata *server-interattivo*. In pratica consente di scrivere codice *lato server* che modifica l'interfaccia utente; le modifiche vengono inviate al *client* mediante la connessione *web-socket*, il quale provvederà ad aggiornare il *Document Object Model* della pagina.

2.3.3 Comunicazione tra client e server mediante Web API

Il termine Web API (Web *Application Programming Interface*) indica un'interfaccia di programmazione – un insieme di metodi – eseguibile da processi remoti mediante il protocollo HTTP.

Diversamente da quanto accade con l'uso di un *web-socket*, non viene stabilita una connessione permanente con il *server*; il *client* invia una richiesta HTTP per ottenere dei dati, il *server* restituisce i dati come *corpo* di una risposta HTTP, normalmente in formato JSON.

2.3.4 Programmazione delle applicazioni CSR

La maggior parte delle applicazioni che implementano il CSR sono basate sul linguaggio JavaScript, poiché è il linguaggio di programmazione adottato dai *web browser*.

Esistono molti *framework* utilizzati per sviluppare applicazioni CSR e si basano quasi tutti su JavaScript; i più conosciuti sono **AngularJS**[3], **ReactJS**, **Vue**. Questi mettono a disposizione funzioni e librerie di codice che semplificano lo sviluppo dell'interfaccia utente.

Anche **Blazor**, basato su **ASP.NET Core**, fornisce una piattaforma di programmazione *lato client*, con la differenza che usa il linguaggio C#, il quale viene tradotto in linguaggio ***WebAssembly*** (o *wasm*) per poter

3 Esiste anche il *framework* **Angular**, che usa il linguaggio di programmazione **TypeScript**.

essere eseguito all'interno del browser.

2.4 Architetture miste

In molti scenari, le soluzioni architetturali presentate nei paragrafi precedenti sono adottate contemporaneamente nella stessa applicazione:

- Accesso a *risorse statiche*, restituite direttamente dal *web server*.
- Uso del SSR per la generazione dinamica di contenuti HTML.
- Uso del CSR per l'implementazione del *front-end* nel browser.

Ad esempio, indipendentemente dal tipo di applicazione, i *fogli di stile* (file CSS), le immagini, i file JavaScript referenziati nelle pagine HTML sono *risorse statiche* che vengono caricate dal *web server* e restituite al *client* senza che sia necessario alcun intervento del programmatore.

Inoltre, esistono applicazioni che usano contemporaneamente il SSR e il CSR, adottando metodi di *rendering* diversi per implementare funzioni distinte.

3 Introduzione al protocollo HTTP

Il protocollo HTTP è stato creato alla fine degli anni 80 ed è alla base del World Wide Web. Usato dai browser per navigare tra i contenuti ipertestuali dei siti web, è diventato un protocollo di riferimento per lo sviluppo di *applicazioni distribuite* di tipo *client-server*.

HTTP si appoggia al *protocollo di trasporto* TCP e si basa su una comunicazione di tipo *richiesta→risposta*. Il *client* apre una connessione TCP, invia al *server* una richiesta, legge la risposta del *server* e chiude la connessione.[4]

Il *server* restituisce sempre una risposta; se non è in grado di soddisfare la richiesta, restituisce una risposta di errore. L'elaborazione di una richiesta implica dunque un cosiddetto ***round trip***, cioè un'*andata-e-ritorno* tra *client* e *server*.

Il protocollo usa prevalentemente il formato testo ed è *case insensitive*.

3.1 Richieste HTTP

HTTP prevede vari tipi di richiesta, caratterizzati da un identificatore chiamato *metodo*: GET, POST, HEAD, PUT, DELETE, PATCH, TRACE, OPTIONS, CONNECT. I *metodi* più impiegati sono **GET** e **POST**.

Una richiesta è strutturata in quattro parti:

***Start line* (o *request line*)**	È composta da tre elementi: • Il *metodo*: GET, PUT, HEAD, etc. • La risorsa richiesta, più precisamente il *percorso* (*path*) della risorsa. • La versione del protocollo HTTP utilizzata.
Header	Un elenco di righe che fornisce informazioni sulla richiesta mediante coppie chiave-valore, separate dal carattere due punti. L'unico *header* obbligatorio è ***host***, che definisce l'URL del *server* destinatario della richiesta.
Riga vuota	Separa l'intestazione (*start line* più *header*) dal *corpo* della richiesta.
Corpo (opzionale)	Memorizza informazioni che il *client* invia al *server*. Soltanto le richieste di modifica/invio dei dati hanno un *corpo* (POST, PUT, PATCH).

3.2 GET: ottenere una risorsa dal server

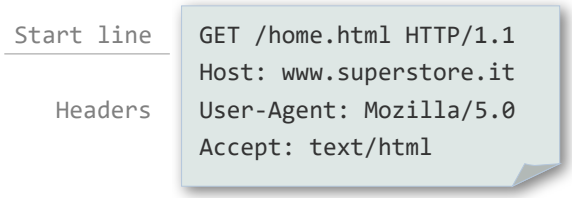
È la richiesta più comune; la *start line* specifica il *metodo* GET e l'*URL relativo* della risorsa, cioè l'URL privo della parte che identifica il *server*.

L'URL può referenziare qualsiasi cosa, non necessariamente una pagina HTML.

3.2.1 Ottenere una pagina HTML

Supponi di navigare alla pagina **home.html** del sito **www.superstore.it**. Il browser invia al *server* la seguente richiesta (sono omesse alcune informazioni negli *header*):

4 Opzionalmente, il protocollo consente di effettuare più richieste usando la stessa connessione.



L'*header* **Host** specifica il nome del sito, mentre l'*header* **Accept** specifica il tipo di contenuto che il *client* è disposto a ricevere; specificando **text/html**, il *client* informa il *server* che è disposto a ricevere soltanto contenuti HTML.

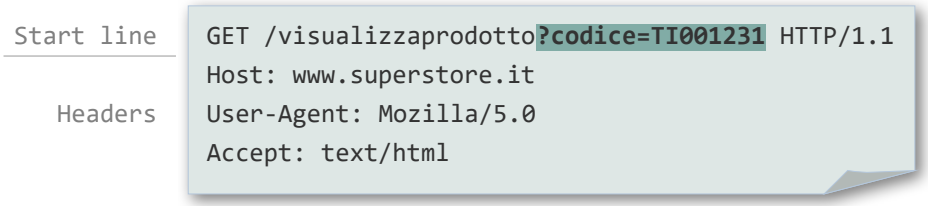
3.2.2 Ottenere un contenuto mediante query string

Una *query string* (*stringa di ricerca*) consente al *client* di specificare uno o più valori nell'URL della richiesta. È composta da un elenco di coppie *chiave=valore* ed è preceduta dal carattere **?**.

Ad esempio, supponi che la pagina **Sconti.html** del sito **www.superstore.it** visualizzi l'elenco dei prodotti scontati. Ogni prodotto è visualizzato mediante un *hyperlink* che specifica l'azione da eseguire cliccando su di esso e il codice del prodotto da visualizzare memorizzato in una *query string*:

```
<a href="visualizzaprodotto?codice=TI001231">Timberland – stivaletti stringati</a>
<a href="visualizzaprodotto?codice=NIK70843">Nike Air Jordan </a>
...
```

Cliccando sul primo *link* viene inviata al *server* la seguente richiesta:



La parte evidenziata mostra la *query string* contenente il codice del prodotto cliccato.

3.3 POST: inviare dei dati al server

Il metodo POST viene usato per inviare dati al *server*, di solito a seguito di un *form* HTML. Oltre alla *start line*, la richiesta definisce perlomeno:

- L'*header* **Content-Type** per indicare la modalità di codifica dei dati.
  Di default è: **x-www-form-urlencoded**.
- L'*header* **Content-Length** per indicare la dimensione in byte del *corpo*.
- Un *corpo* contenente i dati.

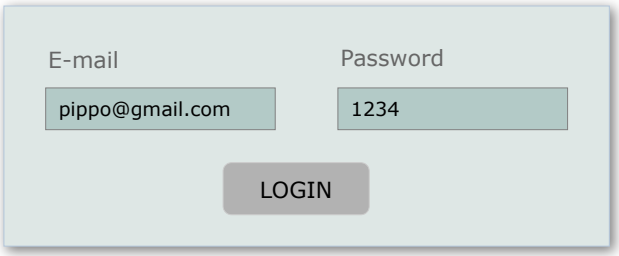
Ad esempio, la pagina **Login.html** del sito **www.superstore.it** consente all'utente di autenticarsi mediante un *form* HTML. I tag **<input>** per l'inserimento delle credenziali sono denominati **email** e **password**. Il tag **<form>** specifica **post** come valore dell'attributo **method** e **/login** come valore dell'attributo **action**.[5]

5 Si sorvola sugli aspetti legati alla sicurezza e si suppone che i dati, password compresa, siano inviati in chiaro.

Login.html

```
<form method="post" action="/login">
    <label>E-mail</label>
    <input name="email" type="text"/>

    <label>Password</label>
    <input name="email" type="password"/>

    <button>LOGIN</button>
</form>
```

Ipotetica interfaccia utente



Normalmente il valore la password non sarebbe visualizzato.

Supponendo che l'utente abbia inserito i valori mostrati a destra, quando l'utente clicca sul bottone, il browser invia la seguente richiesta:

Nota bene:

- L'URL della richiesta coincide con il valore dell'attributo **action** del *form*.
- Il *corpo* della richiesta è codificato come una *query string* (3.2.2); le chiavi sono i nomi dei *tag* di input, i valori sono quelli inseriti dall'utente.[6]
- L'*header* **Content-Type** specifica la modalità di codifica del *corpo*. Il valore indica che il *corpo* è codificato come una *query string*.
- L'*header* **Content-Length** specifica il valore 37, perché 37 è la lunghezza in byte del corpo.[7]

6 Il carattere **@** viene codificato come **%40**.
7 Il testo viene codificato nel sistema UTF8, dunque ad ognuno dei caratteri corrisponde un byte.

3.4 Risposte HTTP

A una richiesta HTTP corrisponde sempre una risposta HTTP, che è struttura in quattro parti:

Status line	È composta da tre elementi: • La versione del protocollo HTTP utilizzata. • Uno ***status code***, che indica il successo o il fallimento nell'elaborazione della richiesta. • Un testo che fornisce una descrizione sintetica dello *status code*.
Header	Un elenco di righe che fornisce informazioni sulla richiesta mediante coppie chiave-valore, separate dal carattere due punti.
Riga vuota	Separa l'intestazione della richiesta (*start line* più *header*) dal *corpo* della richiesta.
Corpo (opzionale)	Memorizza il contenuto della risposta che il *server* invia al *client*. Non tutte le risposte hanno un *corpo*.

Gli *status code* sono catalogati nel seguente modo:

- **1xx** (da 100 a 199): risposta informativa.
- **2xx**: successo. La richiesta è stata soddisfatta.
- **3xx**: redirezione. La richiesta è valida, ma, per completare l'operazione, il *client* viene rediretto a un altro URL. (I browser sono in grado di inviare automaticamente la richiesta al nuovo URL.)
- **4xx**: errore del *client*. La richiesta non è valida.
- **5xx**: errore del *server*. La richiesta non può essere soddisfatta per un problema nel *server*.

Seguono alcuni esempi di *status code* e del corrispondente testo descrittivo:

- **200 OK**: il corpo della risposta contiene i dati richiesti.
- **302 Found**: la risorsa richiesta è raggiungibile all'URL indicato nell'*header* **location**.
- **404 Not Found**: la risorsa richiesta non è stata trovata.
- **500 Internal Server Error**: la richiesta non è stata soddisfatta per un problema interno del *server*.

3.5 Restituire una pagina HTML

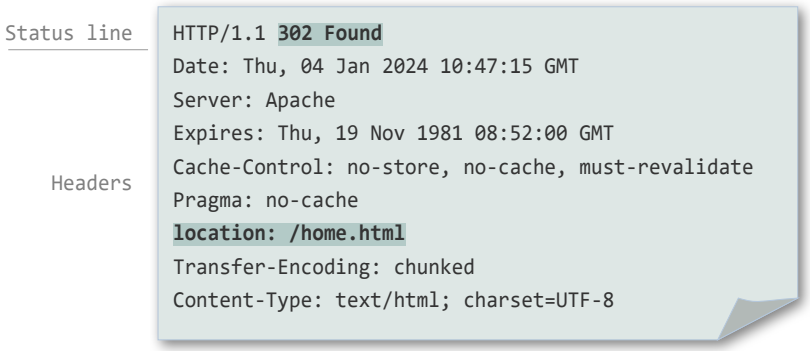
Supponi che la pagina **home.html** del sito **www.superstore.it** sia la seguente:

```
<html>
    <head>
        <link rel="stylesheet" href="stili.css">
    </head>
    <body>
        <h1>SUPERSTORE</h1>
    </body>
</html>
```

La risposta alla richiesta inviata in (3.2.1) sarebbe simile alla seguente:

3.6 Risposta a un POST

Supponi che l'utente effettui il login ipotizzato in (3.3), che il *server* autentichi con successo le credenziali inserite e dirotti l'utente alla home del sito. La risposta del *server* alla richiesta POST sarebbe la seguente:



Si tratta di un tipico pattern di elaborazione di un *form* HTML: i dati vengono elaborati e, se l'operazione ha successo, il *client* viene rediretto a un'altra pagina.

Il browser, ricevendo una risposta simile, invia automaticamente una richiesta GET alla pagina indicata nell'*header* **location** senza che sia necessario alcun intervento dell'utente

3.7 Cockie

Un *cockie* è una stringa di testo che *client* e *server* si scambiano nell'arco di più richieste allo scopo di superare la natura *stateless* (2.2.2) del protocollo HTTP.

Il *server* invia uno o più *cockie* nella risposta, utilizzando l'*header* **Set-Cockie**. Il *client* li re invia nelle richieste successive, utilizzando l'*header* **Cockie**. In questo modo è possibile memorizzare sui *cockie* delle informazioni che "sopravvivono" all'elaborazione di una singola richiesta. Di fatto: il *server* usa il *client* come meccanismo di memorizzazione dei dati.

I *cockie* vengono usati per vari scopi:

- Gestione della *sessione*: stato dell'utente (anonimo, autenticato), carrello acquisti, etc; in generale qualsiasi informazione che il *server* deve "ricordare" per gestire l'interazione con il *client* nell'arco di più richieste.
- Personalizzazioni e impostazioni relative a uno specifico utente.
- Tracciatura e analisi del comportamento dell'utente.

3.7.1 Tipo e durata dei cockie

I *cockie* possono essere di *sessione* o *permanenti*.

Nel primo caso sono gestiti in memoria; il browser li elimina quando viene chiuso. I *cockie permanenti*, invece, sono salvati su disco in modo da essere usati nelle sessioni successive. In queste caso, i *cockie* hanno una scadenza, oltre la quale vengono cancellati dal disco, poiché non sono più accettati dal *server*.

3.7.2 Esempio di cockie

Considera l'implementazione del *carrello acquisti* nel sito **www.superstore.it**. Occorre memorizzare i prodotti nel carrello in modo che siano utilizzabili nell'arco di più richieste, o addirittura più sessioni. A questo scopo si possono usare i *cockie*.

Supponi che l'utente acquisti il prodotto di codice 3, la risposta del browser potrebbe essere la seguente:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: idProdotto=3
...
```

Nelle richieste successive, il *client* invia il *cockie* ricevuto in precedenza, in modo che il *server* possa ricostruire il contenuto del *carrello acquisti* di quel particolare utente.

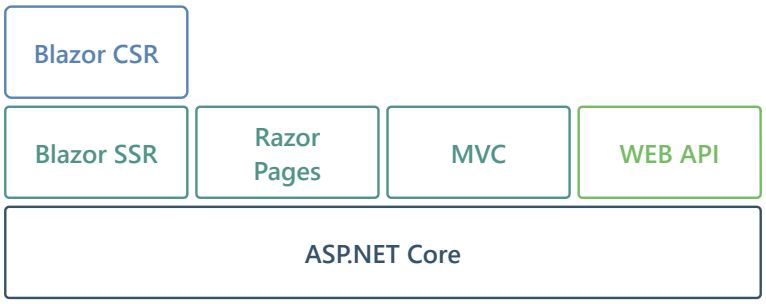
```
GET /prodotti HTTP/1.1
Host: www.superstore.it
Cockie: idProdotto=3
...
```

INTRODUZIONE A BLAZOR

Server Side Rendering

4 Panoramica generale

Blazor è una delle piattaforme per lo sviluppo di *applicazioni web* basata sul *framework* **ASP.NET Core**; le altre sono **ASP.NET Razor Pages**, **ASP.NET MVC** e **ASP.NET WEB API**.



Blazor è l'unica piattaforma *fullstack*: consente di implementare sia il SSR (2.2) che il CSR (2.3). Blazor offre inoltre un'implementazione del CSR chiamata *server interattivo*, basata su una connessione permanente tra il *client* e il *server*.(2.3.2)

Questo testo è dedicato a Blazor SSR, dunque allo sviluppo di *applicazioni web* basate sul modello richiesta-risposta del protocollo HTTP.

4.1 Componenti e pagine

In Blazor, al centro della programmazione c'è il *componente*, un file con estensione **.razor** che implementa una funzione dell'interfaccia utente.

Un *componente* è strutturato in tre parti, tutte opzionali: direttive, *rendering* e sezione `@code`.

```
Direttive
Rendering (HTML + C#)
@code {
   C#
}
```

La potenzialità dei *componenti* consiste nel fatto che possono essere usati come "mattoni" per costruire altri *componenti*. Un *componente* è utilizzabile nel codice HTML di un altro *componente* come se fosse un *tag*.

ComponenteB.razor

```
<h1>Componente B</h1>
<ComponenteA></ComponenteA>
```

ComponenteA.razor

```
<h3>Componente A</h3>
```

Blazor fornisce un nutrito insieme di *componenti predefiniti*; molti altri sono disponibili in rete.

4.1.1 Pagine

Le pagine sono *componenti* che hanno un indirizzo, più precisamente una *route*, dunque sono accessibili al *client* (si dice che sono *routable*, o *instradabili*) .La *route* di una pagina è una stringa preceduta dalla direttiva `@page`, che non deve per forza coincidere con il nome del file, anche se si può adottare questa scelta.

About.razor

```
@page "/about"

<h1>About</h1>
```

Supponendo che la pagina **About.razor** appartenga al sito **www.superstore.it**, sarebbe raggiungibile con l'URL: **http://www.superstore.it/about**, oppure, da un'altra pagina del sito, semplicemente con **/about**.

4.2 Generazione dinamica dell'HTML: *rendering*

Blazor, come tutte le piattaforme SSR, consente di generare dinamicamente il codice HTML. Si parla di ***rendering*** delle pagine.

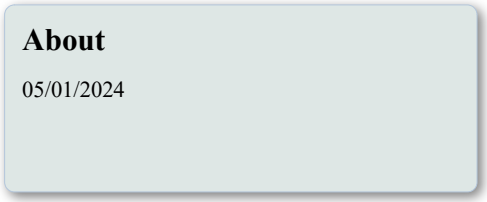
La tecnologia adottata è chiamata *razor* e utilizza il C# integrato con il codice HTML. Il codice C# deve essere prefissato dal carattere `@`.

Ad esempio, ecco come visualizzare la data odierna:

About.razor

```
@page "/about"

<h1>About</h1>

@DateTime.Now.ToShortDateString()
```

Output nel browser



L'esempio mostra un'***espressione implicita*** (6.1.1), il cui risultato viene collocato nello stesso punto della pagina in cui si trova l'espressione.

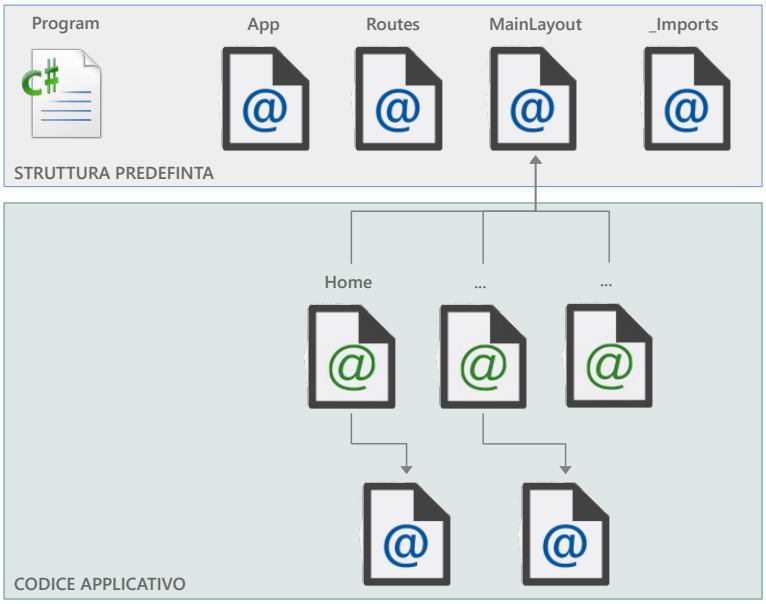
Nella sezione *rendering* si può fare riferimento a variabili definite nella sezione `@code`. Il seguente esempio produce lo stesso output di quello precedente:

```
@page "/about"

<h1>About</h1>

@oggi

@code{
    string oggi = DateTime.Now.ToShortDateString();
}
```

4.3 Struttura generale di un'applicazione Blazor

Un'applicazione Blazor ha una struttura basata su quattro componenti predefiniti e sul file **Program.cs**, che contiene il codice di avvio.

L'unica pagina obbligatoria è **Home**, alla quale corrisponde la *route* `"/"`.



Segue una breve descrizione dei *componenti* predefiniti:

- **App**: rappresenta il punto di ingresso dell'applicazione; definisce la struttura generale delle pagine e referenzia i *fogli di stile*.
- **Routes**: abilita la funzione di *routing*, che consente ai *client* di accedere alle pagine, e referenzia il *componente* **MainLayout**.
- **MainLayout**: definisce il *layout* predefinito, applicato a tutte le pagine.
  È necessario modificare questo file per aggiungere funzioni implementate da tutte le pagine, come ad esempio un menù.
- **_Imports**: importa i *namespace* più utilizzati, in modo che non debbano essere importati da ogni *componente*.
  I *namespace* vengono importati nei *componenti* contenuti nella stessa cartella del file **_Imports**, oppure nelle sue sotto cartelle.

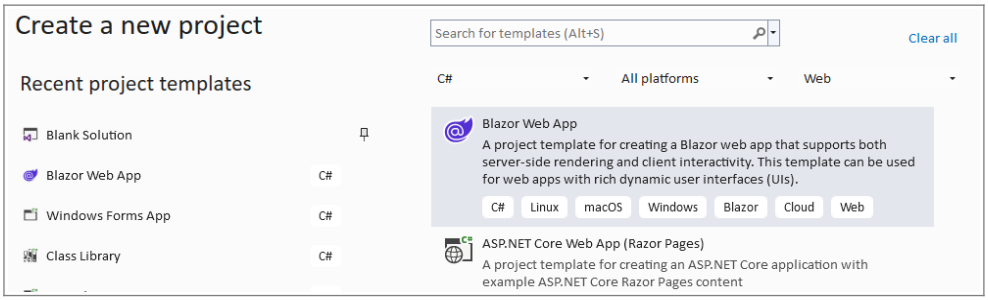
I *componenti* predefiniti sono generati automaticamente durante la creazione del progetto e sono memorizzati nella cartella **Components**.

Nelle applicazioni più semplici, l'unico *componente* da modificare è **MainLayout**.

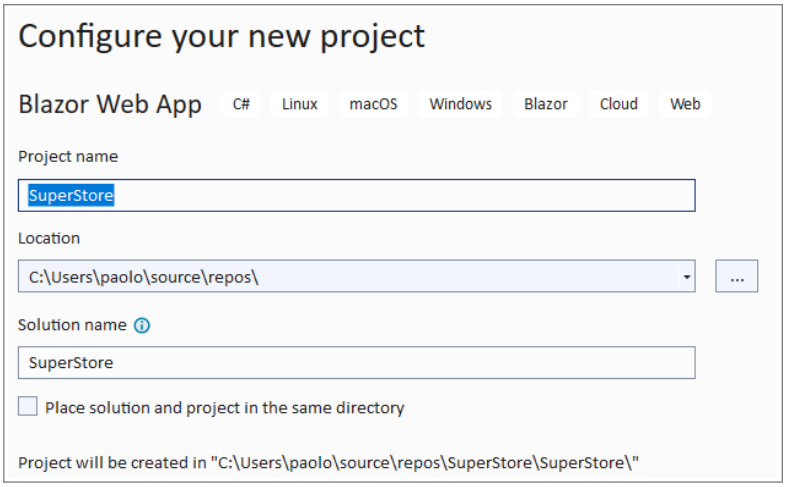
5 Creazione ed esecuzione di un'applicazione Blazor

La seguente procedura descrive la creazione di un'applicazione Blazor *Server-Side Rendering* vuota.

Alla schermata di richiesta del tipo di progetto da creare, occorre selezionare **Blazor Web App**. [8]

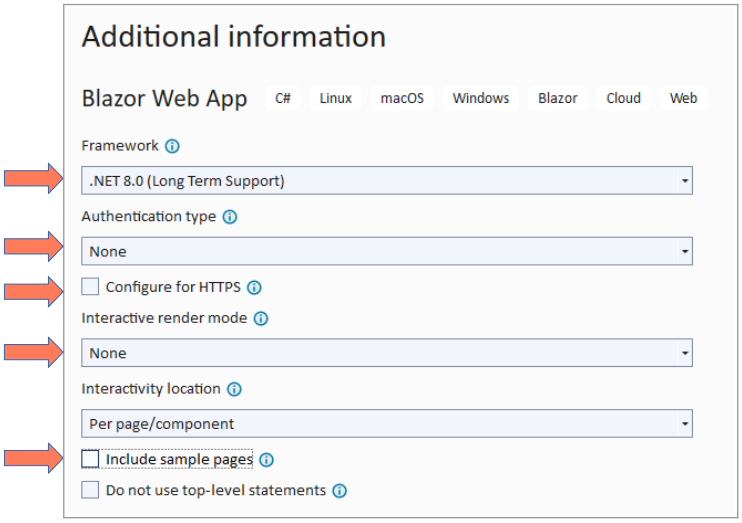


Dopodiché occorre nominare il progetto e specificare la sua collocazione.



Successivamente occorre selezionare le caratteristiche del progetto:

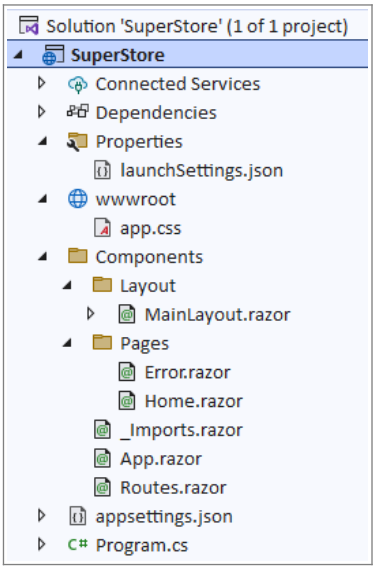
- .NET 8.
- Nessuna *autenticazione*.
- Non configurato per HTTPS.
- Nessuna interattività.
- Non includere pagine di esempio.



8 Nota bene: esistono altri tipi di progetti Blazor, quindi è importante selezionare quello indicato.

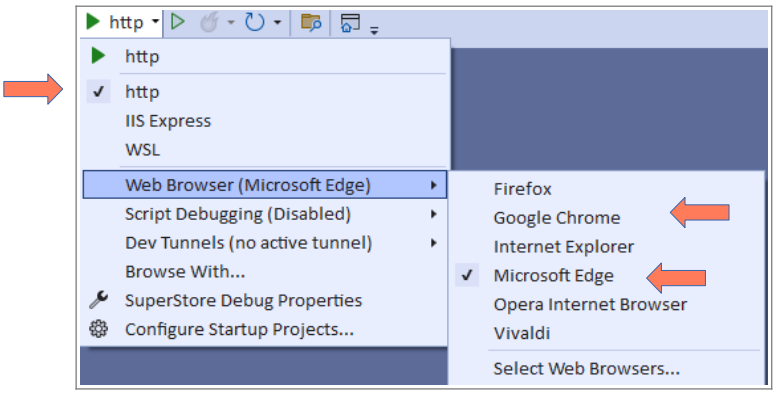
Visual Studio crea un progetto così strutturato:

- La cartella **Components** contiene tutti i *componenti*, tra i quali **App**, **Routes** e **_Imports**.
- La cartella **Layout** contiene il componente **MainLayout**. La cartella **Pages** contiene due pagine predefinite.
- La cartella **wwwroot** contiene le risorse statiche: *fogli di stile*, immagini, script, etc. Di default memorizza il *foglio di stile* **app.css**.
- La cartella **Properties** contiene il file delle impostazioni usate da Visual Studio per eseguire l'applicazione.
- Il file **appsettings.json** memorizza le impostazioni di configurazione dell'applicazione
- **Program.cs** contiene il codice di avvio e consente di configurare i servizi usati dall'applicazione.



5.1 Esecuzione dell'applicazione

Si esegue l'applicazione mediante i comandi consueti. Attraverso il comando **Debug** è possibile specificare il *web server*, IIS Express o Kestrel (**http**), e il *web browser,* Edge, Chrome, etc.



5.1.1 Configurare le impostazioni di avvio

Attraverso il comando **Properties** nel menù **Debug** è possibile intervenire sulle impostazioni d'esecuzione. Un'alternativa è quella di modificare direttamente il file **launchSettings.json**, dove queste impostazioni sono salvate.

Ad esempio, si può modificare la porta di ascolto del *web server* Kestrel (voce **http**), impostandola a **5000**:

```
"http": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "applicationUrl": "http://localhost:5000",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
```

6 Programmazione di una pagina

Le pagine sono il fulcro delle applicazioni Blazor, poiché forniscono i contenuti e implementano le funzioni accessibili ai *client*. Ogni pagina è raggiungibile mediante una (o più) *route*, specificata nella direttiva `@page`. (7)

```
@page "/"

Rendering (HTML + codice C#)

@code{

}
```

Le pagine producono – *renderizzano* – i contenuti dinamicamente, generandoli mediante codice C# integrato nel codice HTML, chiamato convenzionalmente codice *in-linea*.

6.1 Usare C# nelle pagine

Il codice C# può essere scritto in due collocazioni distinte della pagina: *in-linea*, integrato con il codice HTML, o all'interno della sezione `@code`.

Tutto ciò che si trova all'esterno della sezione `@code` è considerato HTML o testo in chiaro; dunque, l'uso di C# *in-linea* è sottoposto a dei vincoli. Esistono quattro modalità d'uso, tutte prefissate dal carattere `@`, tutte che non devono terminare con il carattere '`;`'.

- *Espressioni implicite*.
- *Espressioni esplicite*.
- Costrutti di controllo.
- Blocchi.

6.1.1 Espressioni implicite

Le *espressioni implicite* producono un valore: costanti, variabili, espressioni di qualunque tipo, chiamate a metodi e proprietà. La *chiamata di metodo* deve riferirsi a un metodo che restituisce un valore.

Ad esempio:

```
<h3>@DateTime.Now</h3>

<h3>@Directory.GetCurrentDirectory()</h3>
```

Le *espressioni implicite* sono così chiamate perché il compilatore deduce la loro terminazione lessicale. Non devono contenere spazi, con l'eccezione dalla *chiamata di metodo*, nella quale la parentesi chiusa finale termina l'espressione.

6.1.2 Espressioni esplicite

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

```
<h3>Settimana precedente: @( DateTime.Now - TimeSpan.FromDays(7) )</h3>
```

Sono così chiamate perché la loro terminazione lessicale è indicata esplicitamente dalla parentesi di chiusura.

6.1.3 Costrutti di controllo

`if`, `for`, `foreach`, `while`, etc. Ad esempio:

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

6.1.4 Blocchi di codice

Un *blocco* è rappresentato da una coppia di parentesi graffe; la graffa d'apertura deve essere preceduta da `@` nella stessa riga di codice. Dunque:

Corretto	Scorretto

`@{`
 	
`}`
	`@`
	`{`
	
	`}`

Il codice all'interno del *blocco* è uguale al *codice di primo livello* delle **Console App**. Le variabili sono accessibili dopo la loro dichiarazione, anche all'esterno del blocco, ma non nella sezione `@code`.

Nel seguente *blocco*, la variabile `somma` memorizza la somma degli elementi di un vettore. Successivamente, la variabile è usata come *espressione implicita* in un *tag* HTML.

```
@{
    int[] numeri = { 10, 20, 30 };
    int somma = 0;
    foreach (var x in numeri)
    {
        somma += x;
    }
}
<h3>@somma</h3>
```

6.2 Sezione @code

La sezione `@code` equivale al corpo di una classe: può definire variabili globali, metodi, eventi, altre classi e *record*, ma non *istruzioni di primo livello*. In genere è all'interno di questa sezione che viene collocato il codice di caricamento ed elaborazione dei dati.

Non esistono regole stringenti sulla collocazione della sezione `@code` all'interno della pagina, ma è buona norma collocarla alla fine.

Nel seguente esempio, in `@code` viene definito un vettore di stringhe, visualizzato mediante un elenco di paragrafi:

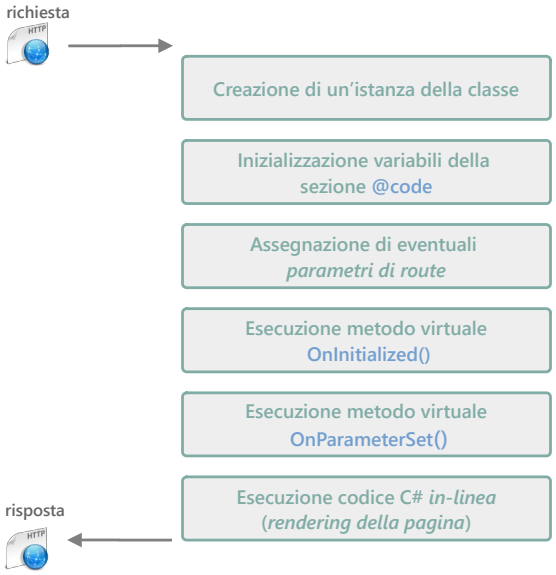
```
@foreach (string nome in nomi)
{
    <p>@nome</p>
}
```

```
@code{
    string[] nomi = {"Rossi, Andrea", "Bianchi, Sara", "Verdi, Filippo"};
}
```

Nota bene: il vettore `nomi` è usato prima della sua dichiarazione, ma non è un errore, perché `nomi` è a tutti gli effetti un campo di classe e la sua inizializzazione avviene prima dell'esecuzione del codice *in-linea*.

6.3 Ciclo di elaborazione della pagina

Nella fase di compilazione le pagine sono tradotte in classi che derivano dalla classe `ComponentBase`. Durante l'esecuzione, la richiesta di una pagina produce l'esecuzione delle seguenti fasi[9]:



Di particolare importanza è il *metodo virtuale* `OnInitialized()`, che è l'equivalente del *gestore d'evento* `Load()` delle applicazioni *windows.forms*: in questo metodo viene collocato il codice di inizializzazione e di caricamento dei dati della pagina.

```
@page "route"

Rendering

@code{
    protected override void OnInitialized()
    {
        Inizializzazione e caricamento dati
    }
}
```

Nota bene: essendo `OnInitialized()` un *metodo virtuale*, occorre ridefinirlo mediante la parola chiave `override`.

9 Per semplicità, lo schema omette l'esecuzione dei *metodi asincroni* `OnInitializatedAsync()` e `OnParameterSetAsync()`.

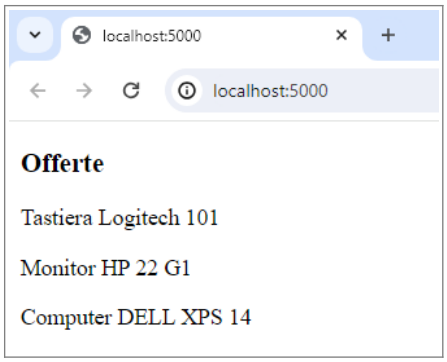
6.4 Un semplice esempio: visualizzare un elenco di dati

Supponi che la home del sito debba visualizzare un elenco di prodotti in offerta; l'elenco è memorizzato nel file di testo **Offerte.txt**, collocato nella cartella **Dati**: [10]

```
Tastiera Logitech 101
Monitor HP 22 G1
Computer DELL XPS 14
```

La funzione potrebbe essere implementata usando codice C# *in-linea*, ma non sarebbe una buona soluzione. L'approccio migliore è caricare il file in `OnInitialized()`, separando il codice che gestisce i dati dal codice di *rendering* della pagina.

```
@page "/"
<h3>Offerte</h3>
@foreach(var prodotto in offerte)
{
    <p>@prodotto</p>
}
@code{
    string[] offerte;
    protected override void OnInitialized()
    {
        offerte = File.ReadAllLines("Dati/Offerte.txt");
    }
}
```

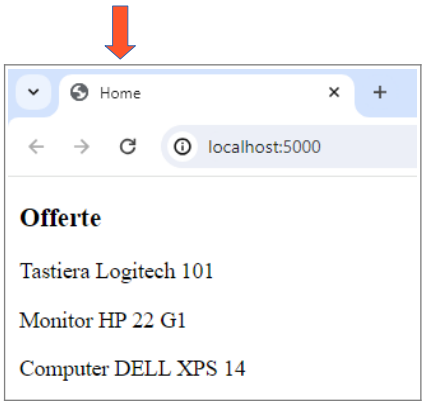


Il metodo `OnInitialized()` carica i titoli nel vettore `offerte`, i cui elementi sono visualizzati mediante codice *in-linea*.

6.5 Impostare il titolo della pagina

Il *componente* predefinito **PageTitle** consente di specificare il titolo della pagina, visualizzato nella *tab* del browser.

```
@page "/"
<PageTitle>Home</PageTitle>
<h3>Offerte</h3>
...
```



10 Non è necessario configurare il file perché venga copiato nella *cartella di output*.

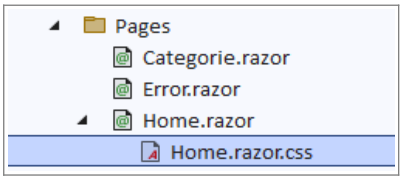
6.6 CSS isolation: applicare regole di stile al singolo componente

Esiste una funzione, chiamata *CSS isolation*, che consente di applicare stili CSS al singolo *componente* con la garanzia che le *regole di stile* non collidano con quelle definite in altri *fogli di stile*.

Per utilizzare questa funzione è sufficiente definire un *foglio di stile* con lo stesso nome del *componente* e collocarlo nella stessa cartella.

Ad esempio, supponi di voler modificare la grandezza del font usato nella visualizzazione delle offerte nella pagina home. (6.4) Si può definire una nuova *classe di stile* nel file **app.css**, oppure definire una regola che si applica ai soli paragrafi del file **Home.razor**.

Nella cartella **Pages** si crea il file **Home.razor.css**, all'interno del quale si definisce la *regola di stile*:




```
p {
    font-size: larger
}
```

Si tratta di una funzione utile, ma della quale non si dovrebbe abusare. In generale è usata per stilizzare in modo specifico i *componenti* d'uso generale, utilizzati in varie pagine.

> ### Modifica degli stili e aggiornamento automatico dell'interfaccia
>
> Lo sviluppo di applicazioni Blazor fruisce di una funzione di Visual Studio che consente di aggiornare automaticamente l'output nel browser semplicemente salvando le modifiche, siano esse all'HTML, al codice C# o agli stili.
>
> Questa funzione può fallire nel caso di *fogli di stile*, perché il browser li memorizza nella propria cache e dunque non ricarica gli stili modificati, ma quelli precedentemente memorizzati.
>
> Ciò vale particolarmente per i *fogli di stile* definiti con la tecnica *CSS isolation*.

7 Routing

Il *routing* è una funzione fondamentale di tutte le *applicazioni web*, poiché consente di "navigare" tra le pagine dell'applicazione.

In Blazor, ogni pagina specifica una *route* mediante la direttiva `@page`. Esistono dei vincoli:

- Deve esistere una pagina che rappresenta la home dell'applicazione e specifica la *route* `"/"`.
- Due pagine non possono avere la stessa *route*. Durante l'esecuzione, il tentativo di usare una *route* definita da due o più pagine produce un crash dell'applicazione.

Le *route* sono stringhe *case insensitive*; dunque le *route* `"/about"` e `"/ABOUT"` sono considerate uguali.

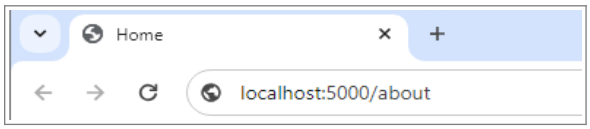
7.1 Route semplici (o senza parametri)

Una *route* è una stringa che inizia con `/` e può essere composta da più *segmenti*, separati dal carattere `/`.

Sono esempi di *route*: `"/"`, `"/about"` e `"/prodotti/dettaglio"` e `"/utenti-sito"`.

Lato client si accede a una pagina usando un URL, mediante un *hyperlink* oppure digitandolo nella barra degli indirizzi del browser.

```
<a href="/about">About</a>
```



7.2 Definire un menù per l'accesso alle pagine dell'applicazione

Supponi di avere tre pagine, *home*, *categorie* e *prodotti*, raggiungibili mediante un menù. Quest'ultimo, per essere visibile in tutte le pagine, deve essere inserito nel *componente* `MainLayout`.

Ogni *link* del menù referenzia la *route* della pagina corrispondente.

```
@inherits LayoutComponentBase
<div>
    <h1>LA MIA BIBLIOTECA</h1>
    <menu>
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/categorie">Categorie</a></li>
            <li><a href="/prodotti">Prodotti</a></li>
        </ul>
    </menu>
    <div>
        @Body
    </div>
</div>
```

```
@page "/"
<PageTitle>Home</PageTitle>
<img src="/Home.jpg" class="center"/>
```

```
@page "/categorie"
<PageTitle>Categorie prodotti</PageTitle>
<h1>Categorie prodotti</h1>
```

```
@page "/prodotti"
<PageTitle>Catalogo prodotti</PageTitle>
<h1>Catalogo prodotti</h1>
```

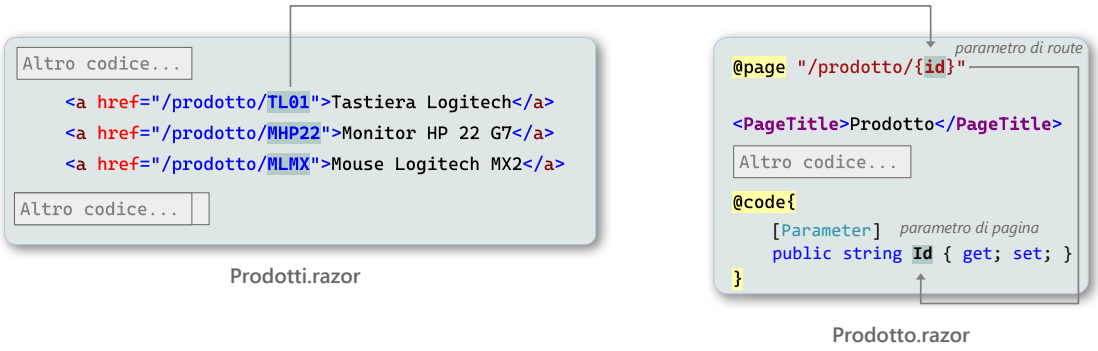
7.3 Passare dei valori a una pagina: route con parametri

Una *route parametrica* consente di passare dei dati alla pagina. La *route* specifica uno o più identificatori, chiamati ***parametri di route***, ai quali sono fatti corrispondere i valori specificati nell'URL di richiesta della pagina.

Nella sezipne `@code`, la pagina deve definire dei ***parametri di pagina*** con lo stesso nome dei *parametri di route* specificati. Un *parametro di pagina* è una proprietà pubblica decorata con l'attributo `[Parameter]`.

Durante l'elaborazione della richiesta, i valori specificati nell'URL sono assegnati ai *parametri di pagina* con lo stesso nome del *parametro di route* corrispondente.

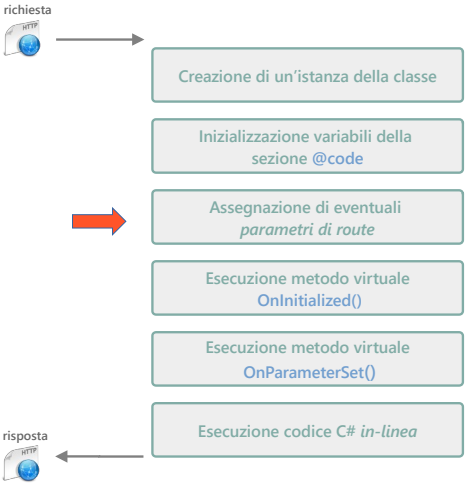
La figura sottostante ipotizza una pagina contenente i *link* relativi a tre prodotti, ognuno dei quali specifica nell'URL il codice del prodotto. La pagina a destra specifica un *parametro di route* di nome **id** e un *parametro di pagina* con lo stesso nome (la corrispondenza è *case insensitive*.)



Supponi che l'utente clicchi sul primo prodotto:

- Il *client* invia la richiesta **GET /prodotto/TL01**.
- Nell'elaborare la pagina (6.3), Blazor estrae il valore **TL01** dall'URL e lo assegna alla proprietà `Id`.

Nota bene: l'assegnamento ai *parametri di pagina* avviene prima dell'esecuzione del metodo `OnInitialized()`.



7.3.1 Parametri di route

Un *parametro di route* è un identificatore racchiuso tra parentesi graffe al quale deve corrispondere un *parametro di pagina*, cioè una proprietà pubblica con lo stesso nome.

Una *route* può avere più *parametri*:

```
/clienti/{nome}/{cognome}
```

Un *parametro di route* può essere specificato tra due *segmenti* della *route*, anche se, in genere, non esiste alcun motivo per farlo.

```
/clienti/{nomecliente}/italia
```

7.3.2 URL corrispondenti a una route parametrica

L'URL usato per accedere a una *route parametrica* deve specificare i valori corrispondenti ai *parametri di route* senza parantesi graffe.

Se la struttura dell'URL non corrisponde alla struttura della *route*, l'applicazione risponderà con un errore di pagina non trovata: **404 Not Found**.

Route	URL	Corrispondenza
`/clienti/{nome}`	`/clienti/filippo`	Trovata
	`/clienti/1000`	Trovata
	`/clienti`	Non trovata
`/clienti/{nome}/{id}`	`/clienti/filippo`	Non trovata
	`/clienti/filippo/rossi`	Trovata
	`/clienti/andrea/20`	Trovata

7.4 Parametri di route tipizzati: vincoli di tipo

I *parametri di route* e i corrispondenti *parametri di pagina* devono essere dello stesso tipo e, di default, si intendono `string`. Per usare un tipo diverso, occorre specificarlo mediante un ***vincolo di tipo*** dopo il nome del parametro:

`/pagina/{parametro:tipo}`

Considera le due pagine schematizzate in (7.3) e supponi che i codici dei prodotti siano interi. In questo caso è utile definire il *parametro di route* e il *parametro di pagina* di tipo `int`.

Prodotti.razor

```
...
<a href="/prodotto/1">Tastiera Logitech</a>
<a href="/prodotto/2">Monitor HP 22 G7</a>
<a href="/prodotto/3">Mouse Logitech MX2</a>
...
```

Prodotto.razor

```
@page "/prodotto/{id:int}"
...
@code{
    [Parameter]
    public int Id { get; set; }
}
```

L'URL usato per la *route* deve specificare un valore convertibile nel tipo del *parametro*, altrimenti viene restituito un errore di pagina non trovata: **404 Not Found**. (Non viene sollevato un errore di conversione.)

7.5 Parametri opzionali

Esistono situazioni nelle quali i dati forniti alla pagina sono opzionali; in loro assenza, la pagina restituisce un contenuto predefinito. Esistono due strade per affrontare questi scenari:

- Definire due (o più) *route* nella pagina, con e senza *parametri*.
- Definire una *route* con *parametri opzionali*.

Di seguito considero la seconda possibilità.

7.5.1 Parametri opzionali

Un *parametro di route opzionale* è caratterizzato dall'uso del carattere **?**.

`/pagina/{parametro?}` `/pagina/{parametro:tipo?}`

In questo caso, la pagina è raggiungibile attraverso due URL distinti, il primo che definisce soltanto la parte fissa della *route*, il secondo che definisce anche il valore del parametro o dei parametri.

Nel primo caso, ai *parametri di pagina* viene assegnato il valore di default.

Supponi di implementare la pagina **Prodotti**, che ha la funzione di visualizzare tutti i prodotti in catalogo, oppure soltanto quelli appartenenti a una certa categoria. In questo caso, il *parametro di route* corrispondente al codice del prodotto è opzionale, perché la pagina potrebbe essere richiesta allo scopo di ottenere tutti i prodotti.

```
@page "/prodotti/{id:int?}"

<PageTitle>Prodotti</PageTitle>

Visualizza il contenuto della lista prodotti

@code{
    [Parameter]
    public int Id { get; set; }

    List<Prodotti> prodotti;
    protected override void OnInitialized()
    {
        if (Id == 0)   verifica se è stato fornito un valore per il parametro
        {
            carica tutti i prodotti
        }
        else
        {
            carica i prodotti della categoria specificata
        }
    }
}
```

Nota bene: se l'URL di richiesta della pagina non specifica il parametro, il *parametro di pagina* corrispondente viene inizializzato al valore predefinito, che nel caso del tipo `int` è zero.

7.5.2 Uso di parametri di pagina nullabili

In alcuni scenari, le verifica del valore di default del *parametro di pagina* non consente di stabilire se alla pagina è stato passato un parametro oppure no. In questi casi è necessario che il *parametro di pagina* sia *nullabile*.

```
@page "/prodotti/{id:int?}"

...

@code{
    [Parameter]
    public int? Id { get; set; }

    List<Prodotti> prodotti;
    protected override void OnInitialized()
    {
        if (Id is null)   verifica se è stato fornito un valore per il parametro
        ...

}
```

Nota bene: nel metodo `OnInitialized()` viene stabilito se il parametro è stato fornito oppure no verificando se è nullo.

7.6 Query string e parametri di query

L'uso di *parametri di route* rappresenta il meccanismo principale per passare dei dati alla pagina, ma non è l'unico: è possibile usare anche le *query string* (3.2.2).

Considera nuovamente l'esempio in (7.4), che implementa la selezione di un prodotto in un elenco. Lo stesso risultato può essere ottenuto con l'uso di *query string* nella pagina **Prodotti** e l'uso di un ***parametro di query*** nella pagina **Prodotto**.

Prodotti.razor

```
...
<a href="/prodotto-qr?id=1">Tastiera Logitech</a>
<a href="/prodotto-qr?id=2">Monitor HP 22 G7</a>
<a href="/prodotto-qr?id=3">Mouse Logitech MX2</a>
...
```

Prodotto.razor

```
@page "/prodotto"
...
@code{
    [SupplyParameterFromQuery]
    int Id { get; set; }
}
```

Un *parametro di query* è una proprietà decorata con l'attributo `[SupplyParameterFromQuery]`. Durante la chiamata della pagina, alla proprietà viene assegnato il valore corrispondente specificato nell'URL di richiesta della pagina. Il confronto della chiave usata nell'URL e il *parametro di query* è *case insensitive*.

Vi sono delle differenze tra l'uso di *parametri di pagina* e di *parametry di query*:

Parametri di pagina	Parametri di query
La *route* di pagina deve specificare dei *parametri di ruote* corrispondenti.	La *route* di pagina non deve specificare dei parametri.
La mancata corrispondenza tra l'URL e i *parametri di route* produce la risposta **404: pagina non trovata.**	La mancata corrispondenza tra l'URL e i *parametri di query* non produce un errore; ai *parametri di query* non viene assegnato alcun valore.
I *parametri di pagina* devono essere proprietà pubbliche.	I *parametri di query* non devono essere necessariamente proprietà pubbliche.

In generale è preferibile l'uso di *parametri di route*, ma esistono scenari, uno dei quali è proposto in (12.3), nei quali occorre usare i *parametri di query*.

8 Form

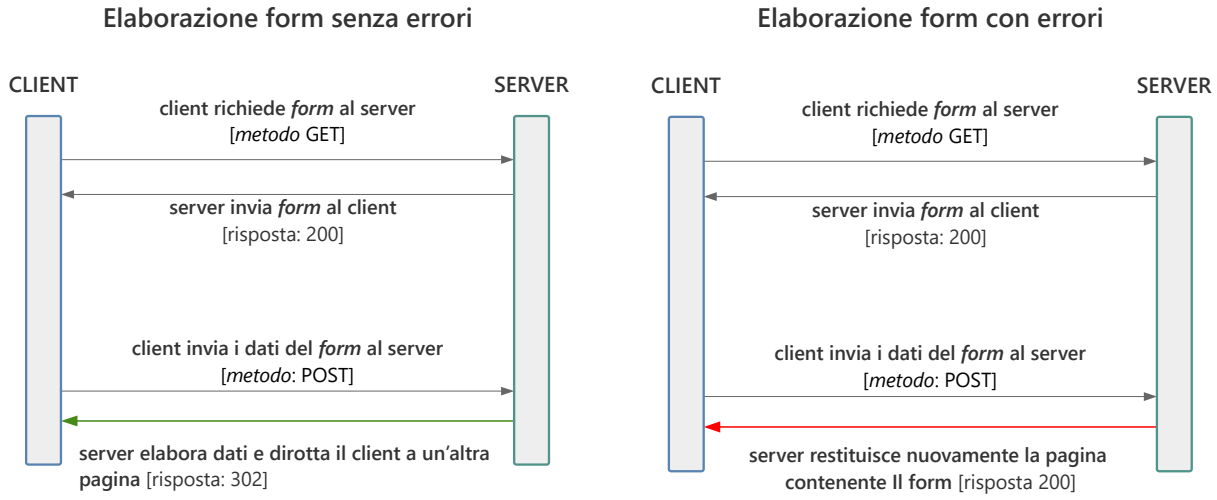
I *form* HTML consentono all'utente di inserire dei dati da processare nel *server* e di eseguire delle operazioni che modificano dello stato dell'applicazione. (3.6)

8.1 Elaborazione di un form

L'uso di un *form* suppone una comunicazione *client-server* che rispecchia lo schema *get→post→redirect:*

- Il *client* ottiene la pagina con il *form*.
- L'utente inserisce i dati e conferma. Il *client* esegue il POST dei dati.
- Il *server* elabora i dati ricevuti e dirotta il *client* su un'altra pagina.

In caso di errori nei dati inseriti, il *server* restituisce nuovamente la pagina HTML contenente il *form*.



In Blazor, la pagina contenente il form è la stessa che ne elabora i dati inseriti. Dunque, l'attributo **action** del *form* specifica la pagina stessa.

8.2 Form Blazor

Blazor supporta l'uso dei *tag* **<form>** e **<input>**, etc, ma fornisce anche dei *componenti* specializzati:

- `EditForm`: è il componente principale e implementa il *tag* **<form>**.
- `InputCheckBox`: input di un valore booleano.
- `InputDate`: input di una data.
- `InputFile`: *upload* di un file.
- `InputNumber`: input di un valore numerico.
- `InputRadio`: input di un valore preso da un insieme predefinito.
- `InputSelect`: *dropdownlist* (corrisponde a un *listbox* o un *combobox*).
- `InputText`: input di una stringa.
- `InputTextArea`: input di un testo multi-linea.

Una pagina che implementa un *form* rispecchia la seguente struttura:[11]

```
@page "/route"

<EditForm FormName="nome-form" Model="model" OnSubmit="Submit">

    <InputXXX @bind-Value="model.Proprietà" ></InputXXX>

    ...

    <button>testo</button>
</EditForm>

@code{
    [SupplyParameterFromForm]
    TipoModel model { get; set; }
    protected override void OnInitialized()
    {
        if (model is null)
        {
            model = new TipoModel();
        }
    }
    void Submit()
    {
        elabora model
    }
}
```

Il *form* specifica un cosiddetto ***model***, cioè una proprietà che memorizza i dati inseriti dall'utente; la proprietà è decorata con l'attributo `[SupplyParameterFromForm]`.

Oltre al *model*, il componente `EditForm` deve specificare un nome e il metodo da eseguire per gestire il *submit*. (Il metodo può avere un nome qualsiasi; non deve chiamarsi necessariamente `Submit()`.)

I *componenti* di input sono collegati alla proprietà del *model* mediante la direttiva `@bind-Value`; durante l'elaborazione del *form*, i dati inseriti nei *componenti* di input sono assegnati ai campi corrispondenti della proprietà.

Nota bene: il *componente* `EditForm` non specifica l'attributo *action*; questo è generato automaticamente e specifica la *route* della pagina contenente il *form*.

8.2.1 Elaborazione del form

Come mostra la figura in (8.1), il *form* viene elaborato in due fasi. Quando la pagina viene richiesta (*metodo* GET), il *model* è `null`, dunque il metodo `OnInitialized()` lo crea, eventualmente inizializzandolo con dei valori predefiniti.

Quando l'utente conferma il *form*, eseguendo il *submit*, il *client* esegue un POST che porta con se i dati inseriti; questi sono assegnati al *model*. Successivamente viene eseguito nuovamente `OnInitialized()`, il quale non esegue alcuna operazione. Infine viene eseguito il metodo `Submit()`, che elabora il *model* in accordo alla funzione del *form*.

In conclusione, il codice contenuto della pagina viene eseguito perlomeno due volte; la prima durante il GET, la seconda durante il POST. (8.7)

11 Una pagina potrebbe definire più form, ma è uno scenario che non viene preso in considerazione.

8.3 Esempio di un form

Supponi di dover implementare l'inserimento di una nuova categoria di prodotti. Le categorie sono definite dal *record* `Categoria`, che rappresenta dunque il *model*:

```
public record Categoria
{
    public int CategoriaId { get; set; }
    public string Nome { get; set; }
}
```

La pagina di inserimento può essere implementata nel seguente modo. (A destra viene mostrato il tag *form* generato automaticamente in corrispondenza del componente `EditForm`.):

```
@page "/nuovacategoria"

<PageTitle>Nuova categoria</PageTitle>

<h3>Nuova categoria</h3>
<EditForm FormName="categoria"
          Model="Categoria"
          OnSubmit="Submit">

    <InputText @bind-Value="Categoria.Nome"></InputText>
    <button>Crea</button>

</EditForm>

@code{
    [SupplyParameterFromForm]
    Categoria Categoria { get; set; }

    protected override void OnInitialized()
    {
        if (Categoria is null)
        {
            Categoria = new Categoria();
        }
    }

    void Submit()
    {
        elabora categoria
    }
}
```

```
<form method="post"
      action="/nuovacategoria">
```

Nota bene: mediante `@bind-Value`, il *componente* `InputText` viene associato alla proprietà `Nome` del *model*.

8.4 Dirottare il client a un altra pagina: navigation manager

Dopo aver elaborato i dati del *form*, il *server* deve dirottare il *client* a un'altra pagina. A questo scopo occorre utilizzare il `NavigationManager`, un componente che è accessibile mediante la direttiva `@inject`:

```
@page "/nuovacategoria"
@inject NavigationManager nav
...
```

Dopo questa definizione, è possibile usare il metodo `NavigateTo()` per navigare a un'altra pagina specificandone la *route*:

```
...
@code{
    [SupplyParameterFromForm]
    Categoria Categoria { get; set; }

    protected override void OnInitialized()
    {
        if (Categoria is null)
        {
            Categoria = new Categoria();
        }
    }

    void Submit()
    {
        elabora categoria

        nav.NavigateTo("/categorie");  dirotta il client alla pagina /categorie
    }
}
```

8.5 Upload di un file

Attraverso un *form* è possibile implementare la funzione di *upload* di uno o più file. È una funzione standard del protocollo HTTP ed è utilizzabile mediante il *tag* **<input type="file">**, o il *componente* `InputFile`.

Per utilizzare questa funzione è necessario che il *form* definisca l'attributo `enctype` e che sia usato un oggetto di tipo `IFormFile` o `IFormFileCollection` per ricevere il file o i file inviati.

Partendo dal precedente esempio, supponi che ad ogni categoria sia associata un'immagine, inviata dal *client* durante l'inserimento. L'obiettivo è verificare se l'immagine è stata inviata e, in caso positivo, salvarla nella cartella **Immagini** in **wwwroot**.

Segue il *form* e la dichiarazione dei parametri necessari per memorizzare i dati inviati dal *client*.

```
...
<EditForm FormName="form" Model="Categoria" OnSubmit="Submit" enctype="multipart/form-data" >

    <InputText @bind-Value="Categoria.Nome"></InputText>

    <InputFile name="File"></InputFile>

    <button type="submit">Submit</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    public Categoria Categoria { get; set; }

    [SupplyParameterFromForm]
    IFormFile File { get; set; }
    ...
```

Alcune note:

- Il *componente* `InputFile` non usa la direttiva `@bind-Value`, ma l'attributo `name`. È necessario adottare questa soluzione per aggirare un bug che affligge questo *componente*.
- Nella sezione `@code`, la proprietà `File` è definita separatamente e non come proprietà della classe che rappresenta il *model*. Si tratta di un'opzione che può essere adottata per qualsiasi *componente* di input.

Segue l'implementazione parziale del metodo `Submit()`. Il codice verifica che il file sia stato inviato, crea il percorso di destinazione e un `FileStream` per salvarlo.

```
    [SupplyParameterFromForm]
    IFormFile File { get; set; }

    private void Submit()
    {
        if (File != null)  il file è stato inviato?
        {
            var nomeFile = Path.Combine("wwwroot/Immagini", File.FileName);
            var fs = new FileStream(nomeFile, FileMode.Create);
            File.CopyTo(fs);   scrive il file su disco
            fs.Close();
        }
        ...
    }
}
```

8.6 Form di modifica dei dati

L'uso di un *form* per la modifica dei dati si svolge in due fasi:

- Il *client* seleziona i dati da modificare, ad esempio mediante un *link* che referenzia un *id*. Il *server* invia al *client* il *form* popolato con i dati richiesti.
- L'utente modifica i dati e conferma il *form*. Il *server* riceve i nuovi dati e li aggiorna sul database.

Durante questo processo, che vede due richieste HTTP, occorre tenere traccia dell'*id* corrispondente ai dati. È un problema tipico delle applicazioni *state-less* (2.2.2): tenere traccia dei dati che si stanno modificando.

Una soluzione è quella di preservare *l'id* dei dati tra le due richieste utilizzando un *tag* di input "nascosto" per memorizzare *l'id*. In Blazor questa tecnica non è necessaria, perché il componente `EditForm` memorizza l'URL usato per ottenere la pagina, compreso anche l'*id* corrispondente al *parametro di route*.

Considera l'implementazione di un *form* per la modifica di una categoria.

```
@page "/editcategoria/{id:int}"

<PageTitle>Modifica categoria</PageTitle>
<h3>Modifica categoria</h3>

<EditForm FormName="edit-categoria" Model="categoria" OnSubmit="Submit">

    <InputText @bind-Value="categoria.Nome" placeholder="Inserisci nome"></InputText>
        <button>Modifica</button>
</EditForm>
```

```
@code {
    [Parameter]
    public int Id { get; set; }

    [SupplyParameterFromForm]
    Categoria categoria { get; set; }

    StoreContext db = new();
    protected override void OnInitialized()
    {
        if (categoria is null)
        {
            categoria = db.TrovaCategoria(Id);
        }
    }
    ...
}
```

Supponi che l'utente selezioni la categoria di *id* 1, il cui valore viene assegnato al *parametro di pagina* `Id`. Nel restituire la pagina al client, Blazor converte il *componente* `EditForm` nel seguente tag *form*:

```
<form method="post" action="/editcategoria/1"> ... </form>
```

Quando viene eseguito il POST del *form*, il *model* contiene il nome modificato dall'utente e al *parametro di pagina* `Id` viene assegnato nuovamente il valore 1. Quindi, il *server* conosce l'*id* della categoria da aggiornare con il nuovo nome.

```
...
@code {
    [Parameter]
    public int Id { get; set; }

    [SupplyParameterFromForm]
    Categoria categoria { get; set; }

    StoreContext db = new();
    ...
    void Submit()
    {
        categoria.CategoriaId = Id;
        db.AggiornaCategoria(categoria);
        db.SaveChanges();
    }
}
```

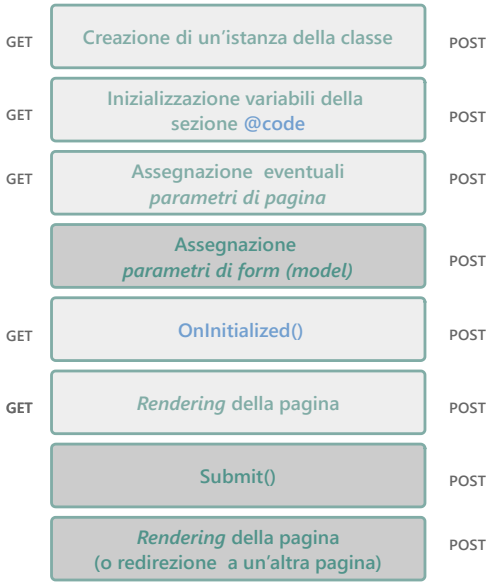
In conclsione, Blazor usa la *route* della pagina, memorizzata nell'attributo *action* del *form*, per memorizzare un dato – l'*id* della categoria – che occorre preservare tra la richiesta GET e la richiesta POST.

8.7 Idiosincrasia dei form Blazor

In Blazor SSR, i *form* soffrono di una problematica che in alcuni scenari produce un'esecuzione inefficiente o addirittura un crash dell'applicazione.

Il problema deriva dal fatto che la pagina subisce due chiamate, GET e POST, che richiedono due elaborazioni distinte, ma:

1. Non esiste un modo diretto per stabilire se la chiamata è un GET o un POST.
2. Il metodo che gestisce il *submit* viene eseguito dopo `OnInitialized()` e l'esecuzione del codice C# *in-linea*. In pratica: viene eseguito il *rendering* anche nella fase di POST.



Negli scenari come quello mostrato in (8.3) ciò non crea alcun problema, ma la situazione è diversa se il form visualizza dei dati oltre a consentire il loro inserimento.

Il seguente esempio mostra un *form* per l'inserimento di un nuovo prodotto. L'utente deve inserire il nome del prodotto e selezionare la categoria di appartenenza. L'elenco delle categorie è caricato dal database e visualizzato mediante un *dropdownlist*.

```
@page "/nuovoprodotto"

<PageTitle>Nuovo prodotto</PageTitle>

<h3>Nuovo prodotto</h3>

<EditForm FormName="prodotto" Model="Model" OnSubmit="Submit">

    <InputText @bind-Value="model.Nome"></InputText> <br />

    <InputSelect @bind-Value="model.CategoriaId">

        @foreach (var cat in model.Categorie)        solleva un eccezione durante il POST
        {
            <option value="@cat.CategoriaId">@cat.Nome</option>
        }
    </InputSelect> <br />

    <input type="submit" value="Crea"/>

</EditForm>

@code{

    [SupplyParameterFromForm]
    ProdottoModel Model {get; set; }

    StoreContext db = new();

    protected override void OnInitialized()
    {
        if (Model is null)
        {
```

```
            Model = new ProdottoModel()
            {
                Categorie = db.Categorie.ToList()   carica categorie dal database
            };
        }
    }

    void Submit()
    {
         elabora model (aggiunge il prodotto al database)
    }

    public class ProdottoModel
    {
        public string Nome { get; set; }
        public int CategoriaId { get; set; }
        public List<Categoria> Categorie { get; set; }
    }
}
```

La riga evidenziata nella pagina pagina precedente è destinata a produrre un'eccezione durante il POST del *form*. Per comprenderne il motivo occorre analizzare il codice alla luce dello schema a destra, che riproduce l'elaborazione della pagina durante il POST.

1. Viene creata un'istanza del *model* e assegnati i dati inseriti, il nome del prodotto e l'*id* della categoria, alle proprietà corrispondenti.
2. Viene eseguito `OnInitialized()`; poiché il *model* esiste già, non viene creato; dunque non viene caricata la lista delle categorie.
3. Viene eseguito il *rendering* della pagina, poiché la proprietà `model.Categorie` è `null`, viene sollevata un'eccezione.

Il problema è rappresentato dal punto 3: non sarebbe necessario eseguire il *rendering* della pagina, perché nel POST occorre inserire un prodotto nel database e dirottare il *client* a un'altra pagina.

Purtroppo, Blazor non fa alcuna distinzione tra GET e POST, quindi esegue comunque eseguite tutte le fasi di elaborazione della pagina.

8.7.1 Soluzione semplice (ma inefficiente)

Una semplice soluzione è quella di caricare sempre la lista delle categorie, anche durante il POST.

```
protected override void OnInitialized()
{
    StoreContext db = new();
    if (Model is null)
    {
        Model = new ProdottoModel();
    }
    model.Categorie = db.Categorie.ToList();  eseguita sia nel GET che nel POST della pagina
}
```

Si tratta di una soluzione inefficiente, perché carica dei dati dal database nonostante non sia necessario visualizzarli, dato che il client viene dirottato a un'altra pagina.

8.7.2 Bypassare il *rendering* del form

È la soluzione più efficiente e può essere realizzata verificando se l'elaborazione della pagina è un POST. Esistono vari modi per farlo; una soluzione ad hoc è quella di definire una proprietà che restituisce `true` se il *model* è stato creato, ma non contiene le categorie.

```
[SupplyParameterFromForm]
ProdottoModel Model {get; set; }

public bool Post => model != null && model.Categorie == null; true se è un POST

...
```

Dopodiché si usa questa proprietà per condizionare il *rendering* del contenuto del *form*.

```
<EditForm FormName="prodotto" Model="Model" OnSubmit="Submit">
    @if (Post == false)
    {
        <InputText @bind-Value="model.Nome" placeholder="Inserisci nome"></InputText>
        <br />
        <InputSelect @bind-Value="model.CategoriaId">
            @foreach (var cat in model.Categorie)
            {
                <option value="@cat.CategoriaId">@cat.Nome</option>
            }
        </InputSelect>
        <br />
        <input type="submit" value="Crea" />
    }
</EditForm>
```

(Nota bene: la `if` condiziona il *rendering* del contenuto del *form*, non il *form* stesso, come sarebbe più naturale. Durante il POST, il *form* in sé deve essere *renderizzato*, anche se privo di contenuto.)

8.7.3 Attivare il *rendering* dopo aver gestito il submit del form

Se l'utente inserisce dei dati errati, il *server* restituisce nuovamente la pagina in modo che possano essere reinseriti. In questo caso occorre eseguire il codice di *rendering* e, dunque, caricare la lista delle categorie.

Di seguito ipotizzo che l'unico errore possa essere l'inserimento di un nome vuoto.

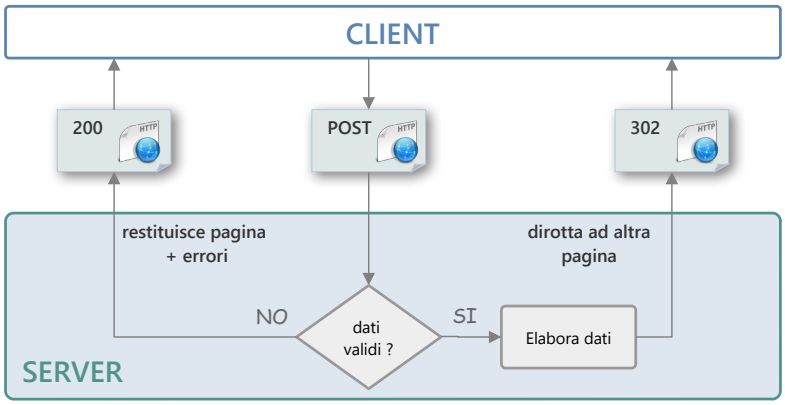
```
void Submit()
{
    if (string.IsNullOrWhiteSpace(model.Nome))
    {
        model.Categorie = db.Categorie.ToList();  ora la proprietà Post restituisce false
    }
    else
    {
        nav.NavigateTo("/prodotti");
    }
}
```

9 Validare i dati di un form

La *validazione dati* è una funzione fondamentale di ogni applicazione, a maggior ragione delle *applicazioni distribuite*. Validare i dati significa verificare che siano conformi a certi vincoli; in caso contrario l'applicazione deve richiedere all'utente di reinserirli.

Nelle *applicazioni web*, la validazione dati può essere eseguita nel *client*, nel *server* o in entrambi. La validazione *lato client* è più efficiente, perché evita un *round trip* tra *client* e *server*. Blazor SSR implementa la validazione *lato server*.

Questa rispecchia il seguente schema:



Il *client* esegue un POST con i dati inseriti dall'utente. Il *server* verifica i dati: se sono validi li elabora e dirotta il *client* a un'altra pagina; se non sono validi restituisce la pagina al *client* con l'aggiunta di uno o più messaggi di errore.

Blazor definisce dei *componenti* predefiniti per gestire questa procedura, ma è possibile implementarla autonomamente. In entrambi i casi occorre:

1. Verificare la validità dei valori in sé: correttezza di tipo, di formato, di intervallo, etc.
2. Verificare il rispetto delle regole dell'applicazione (*business logic*): unicità di codici, chiavi e *nickname*, lunghezza e formato delle password, etc.
3. Generare dei messaggi d'errore diretti all'utente.

> **Blazor *Data annotation validation***
>
> Blazor definisce dei *componenti* di validazione e di visualizzazione degli errori basati sull'uso di ***data annotation***. Un'*annotazione dati* è un attributo che decora una proprietà del *model* e consente di applicare dei vincoli sui valori che memorizza.
>
> Purtroppo questa funzione non gestisce correttamente la validazione di *proprietà nullabili* di tipo valore: date, numeri, valori booleani, etc.

9.1 Validare la registrazione di un utente

Supponi di dover implementare il *form* di registrazione degli utenti di un sito. L'utente deve inserire l'e-mail, la password e la conferma della password. Dunque, occorre il seguente *model*:

```
public record RegistraUtenteModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfermaPassword { get; set; }
}
```

Si vogliono applicare le seguenti regole di validazione: tutti i campi devono essere non vuoti; `Password` e `ConfermaPassword` devono essere uguali.

Segue un'implementazione (che omette l'accesso al database).

```
@page "/registra"
@inject NavigationManager nav

<PageTitle>Registra utente</PageTitle>
<h3>Registrazione utente</h3>

<EditForm FormName="registra" Model="Model" OnSubmit="Submit">

    <InputText @bind-Value="Model.Email" placeholder="Email"></InputText><br />
    <InputText @bind-Value="Model.Password" placeholder="Password"></InputText><br />
    <InputText @bind-Value="Model.ConfermaPassword" placeholder="Conferma password">
    </InputText><br />

    <button>Registra</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    RegistraUtenteModel Model { get; set; } = new();

    void Submit()
    {
        if (Model.Email == "" || Model.Password == "" ||
            Model.Password != Model.ConfermaPassword)
        {
            return;
        }

        ... verifica se l'utente esiste già nel database
        ...in caso negativo, inserisce l'utente

        nav.NavigateTo("/");
    }
    Definizione model...
}
```

Quello mostrato è un approccio incompleto, perché non produce i messaggi di errore corrispondenti ai dati non validi. Inoltre non viene verificata la correttezza della e-mail.

9.2 Generare i messaggi di errore

L'obiettivo è quello di associare a ogni *componente* di input un messaggio di errore in caso di input non valido.

Segue un'implementazione in grado di visualizzare soltanto un errore per campo di input, basata su un dizionario, `erroriInput`, contenente i messaggi di errore associati ai componenti di input. I nomi delle proprietà del *model* fungono da chiave.

Un metodo, `ErroreInput()`, riceve una chiave e restituisce il messaggio di errore corrispondente, oppure una stringa vuota se per quel *componente* di input non ci sono errori.

```
...
<EditForm FormName="registra" Model="Model" OnSubmit="Submit">

    <InputText @bind-Value="Model.Email" placeholder="Email"></InputText>
    <span class="validation-message">@ErroreInput("email")</span><br />

    <InputText @bind-Value="Model.Password" placeholder="Password"></InputText>
    <span class="validation-message">@ErroreInput("password")</span><br />

    <InputText @bind-Value="Model.ConfermaPassword" placeholder="Conferma password"></InputText>
    <span class="validation-message">@ErroreInput("conferma-password")</span><br />

    <button>Registra</button>

</EditForm>
```

Il metodo `Submit()` verifica il *model* e aggiunge al dizionario i messaggi di errore:

```
...
void Submit()
{
    if (Model.Email == "")
    {
        erroriInput.Add("email", "Il campo 'Email' è vuoto");
    }
    else if (!ValidaEmail(Model.Email))
    {
        erroriInput.Add("email", "Il formato della email non è valido");
    }
```

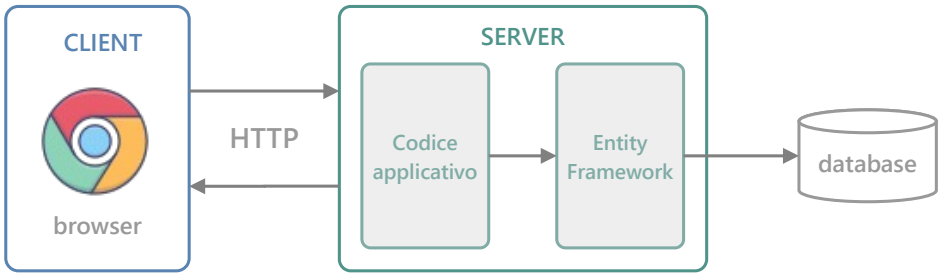
```
    if (Model.Password == "")
    {
        erroriInput.Add("password", "Il campo 'Password' è vuoto");
    }

    if (Model.Password != Model.ConfermaPassword)
    {
        erroriInput.Add("conferma-password", "'Password' e 'Conferma password' non coincidono");
    }
    ...
}

public bool ValidaEmail(string email)
{
    var pattern = @"^[a-zA-Z0-9.!#$%&'*+-/=?^_`{|}~]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$";

    var regex = new Regex(pattern);
    return regex.IsMatch(email);
}
```

Una soluzione più sofisticata, in grado di visualizzare più messaggi di errore per componente di input, è mostrata in (13.2).

10 Uso e configurazione di Entity Framework

Il funzionamento di Entity Framework è indipendente dal tipo di applicazione nel quale viene usato; ciò detto, l'uso in una *applicazione web* richiede un approfondimento.

Innanzitutto si tratta di una tecnologia *lato server*; nelle applicazioni che implementano il CSR (2.3), il *front-end* non si interfaccia direttamente con Entity Framework.

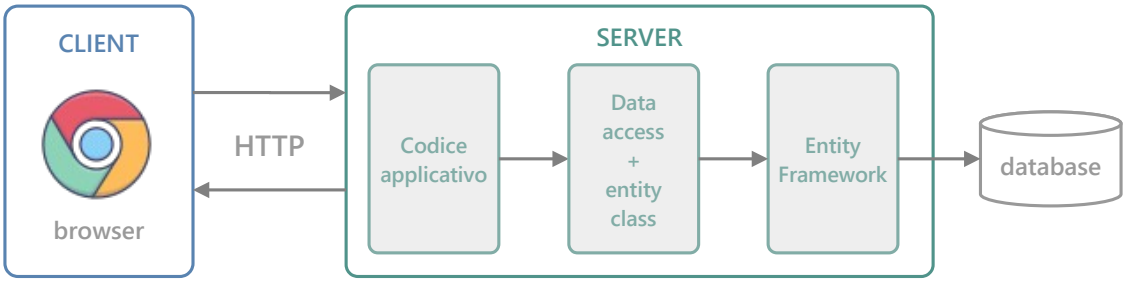


In secondo luogo, le *applicazioni web* che implementano il SSR (2.2) rappresentano uno scenario nel quale l'*oggetto context* che esegue le operazioni sul database viene creato e "buttato" ad ogni richiesta, dunque non tiene traccia delle *entità* caricate in memoria (vedi tutorial Entity Framework: **Uso di EF in scenari N-Tier**").

Infine, è importante considerare la quantità di dati coinvolti nelle operazioni, poiché questi sono ottenuti dal *server* e viaggiano su una connessione di rete per raggiungere il *client*. Una *query* che producesse centinaia o addirittura migliaia di record provocherebbe un tempo di risposta elevato, probabilmente inaccettabile in alcuni scenari.

10.1 Uso di Entity Framework in una *class library*

È lo scenario più comune: implementare il *data access* in una *class library*, in modo che sia logicamente indipendente dal codice applicativo.



Di seguito suppongo l'esistenza di una *class library* che definisca il *data access* e le *entity class* per l'accesso al database **Biblioteca**.

È utile importare il *namespace* della *class library* una volta per tutte nel componente **_Imports.razor**. (4.3)

```
@using System.Net.Http
@using System.Net.Http.Json
...
@using Biblioteca.DataAccess;
```

La *class library* definisce la *classe context* `BibliotecaContext`:

```
public class BibliotecaContext : DbContext
{
    Configurazione della stringa di connessione e dell'associazione Autori-Libri...

    public DbSet<Libro> Libri { get; set; }
    public DbSet<Autore> Autori { get; set; }
    public DbSet<Genere> Generi { get; set; }
    public DbSet<Tesserato> Tesserati { get; set; }
    public DbSet<Prestito> Prestiti { get; set; }
}
```

10.2 Creazione e uso di un oggetto context

Nelle pagine che restituiscono dei contenuti al *client*, l'uso di EF è estremamente semplice.

La seguente pagina visualizza l'elenco dei generi letterari:

```
@page "/generi"

<PageTitle>Generi</PageTitle>
<h3>Generi letterari</h3>
<hr />

@foreach (var g in db.Generi)
{
    <p> <a href="/catalogo/@g.GenereId">@g.Nome</a></p>
}

@code{
    BibliotecaContext db = new();
}
```

10.3 Modifica di un record

La modifica dei dati rappresenta uno *scenario disconnesso* sull'uso di EF: l'*oggetto context* che carica i dati non è lo stesso oggetto che li aggiorna sul database.

Segue un *form* che implementa in modo scorretto la modifica del nome di un genere letterario. (8.6)

```
@page "/modifica-genere/{id:int}"
@inject NavigationManager nav

<h3>Modifica genere</h3>
<hr />

<EditForm FormName="modifica-genere" Model="Genere" OnSubmit="Submit">
    <input type="hidden" name="Genere.GenereId" @bind-value="Genere.GenereId"/>
    <InputText @bind-Value="Genere.Nome"></InputText>
    <button>Modifica</button>
</EditForm>
```

```
@code {

    [Parameter]
    public int Id { get; set; }

    [SupplyParameterFromForm]
    Genere Genere { get; set; }  non è tracciato dall'oggetto context db

    BibliotecaContext db = new();
    protected override void OnInitialized()
    {
        if (Genere is null)
        {
            Genere = db.Generi.Find(Id);
        }
    }

    void Submit()
    {
        db.SaveChanges(); non esegue alcun aggiornamento nel database!

        nav.NavigateTo("/generi");
    }
}
```

La soluzione si basa sull'idea che la proprietà `Genere`, precedentemente caricata dal database, sia riconosciuta come modificata da EF; dunque, l'esecuzione del metodo `SaveChanges()` dovrebbe produrre l'istruzione SQL UPDATE necessaria per salvarla sul database. Il codice funzionerebbe se fosse collocato in un'*applicazione desktop*, ma non funziona in un'*applicazione web*.

Il problema è che l'oggetto *context* `db` usato per caricare il genere dal database non è lo stesso oggetto *context* usato per salvarlo. Dunque: la proprietà `Genere` creata durante il POST rappresenta un oggetto *detached*, sul quale EF non ha alcuna informazione e per il quale non genera un'istruzione SQL.

La soluzione consiste nel comunicare a EF che la proprietà `Genere` referenzia un oggetto modificato, per il quale occorre generare l'istruzione UPDATE. Un modo per ottenere questo risultato è impostare direttamente lo stato *modificato* della proprietà:

```
...
void Submit()
{
    var entry = db.Entry(Genere);
    entry.State = Microsoft.EntityFrameworkCore.EntityState.Modified;
    db.SaveChanges();
    nav.NavigateTo("/generi");
}
```

10.4 Configurare l'oggetto *context* (e l'applicazione)

Il file **Program.cs** contiene il codice di configurazione e di avvio dell'applicazione. È qui che vengono stabilite e configurate le funzioni necessarie ai *componenti* dell'applicazione.

Di default, **Program.cs** ha la seguente struttura:

```
var builder = WebApplication.CreateBuilder(args);
                                                    Aggiunge i servizi
builder.Services.AddRazorComponents();
-------------------------------------------------------------------------
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
}
                                                    Stabilisce le funzioni da usare
app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>();
-------------------------------------------------------------------------
                                                    Avvia l'applicazione
app.Run();
```

In questo file è possibile istruire Blazor a creare l'oggetto *context* in modo che sia passato automaticamente ai *componenti* che ne richiedono l'uso. Ciò consente di stabilire esternamente alla classe *context* i parametri del suo funzionamento: il *provider* e la *stringa di connessione*.

```
using Microsoft.EntityFrameworkCore;
using Biblioteca.DataAccess;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<BibliotecaContext>(
    options => options.UseSqlServer(@"Data Source=(localdb)\MSSQLLocalDB..."));

...

app.Run();
```

10.4.1 Modifica della classe context

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

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

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {...}

    public DbSet<Libro> Libri { get; set; }
    public DbSet<Autore> Autori { get; set; }
    ...
}
```

Nota bene: il metodo `OnConfiguring()` non è più necessario.

10.4.2 “Iniezione” dell’oggetto context nei componenti

Queste modifiche fanno sì che sia Blazor a creare automaticamente gli oggetti *context*. I *componenti* che ne richiedono l’uso dovranno utilizzare la direttiva `@inject` (8.4):

Ad esempio, ecco come deve essere modificata la pagina **Generi**:

Creazione “manuale” dell’oggetto context

```
@page "/generi"

<PageTitle>Generi</PageTitle>

<h3>Generi letterari</h3>
<hr />
@foreach (var g in db.Generi)
{
    <p>
    <a ref="/catalogo/@g.GenereId">@g.Nome</a>
    </p>
}
@code{
     BibliotecaContext db = new();
}
```

“Iniezione” dell’oggetto context

```
@page "/generi"

@inject BibliotecaContext db

<PageTitle>Generi</PageTitle>

<h3>Generi letterari</h3>
<hr />
@foreach (var g in db.Generi)
{
    <p>
    <a ref="/catalogo/@g.GenereId">@g.Nome</a>
    </p>
}
```

10.5 Utilizzare il file delle impostazioni “appsettings.json”

ASP.NET implementa un sistema di configurazione che può utilizzare diverse sorgenti: file JSON, XML, testo, database, *linea di comando*, etc. La sorgente impiegata più comunemente è il file **appsettings.json**, generato automaticamente durante la creazione del progetto. (5)

In questo file è possibile definire vari parametri di configurazione, compresa la *stringa di connessione* del database:[12]

```
{
  "ConnectionStrings": {
    "stringaConnessione": "Data Source=(localdb)\\MSSQLLocalDB; ..."
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

In **Program.cs** esistono vari modi per accedere alle impostazioni di configurazione. Per ottenere la *stringa di connessione* il modo più semplice è l’uso del metodo `GetConnectionString()` dell’oggetto `Configuration`:

12 È possibile definire più di una *stringa di connessione*.

```
...

var builder = WebApplication.CreateBuilder(args);

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

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

...
app.Run();
```

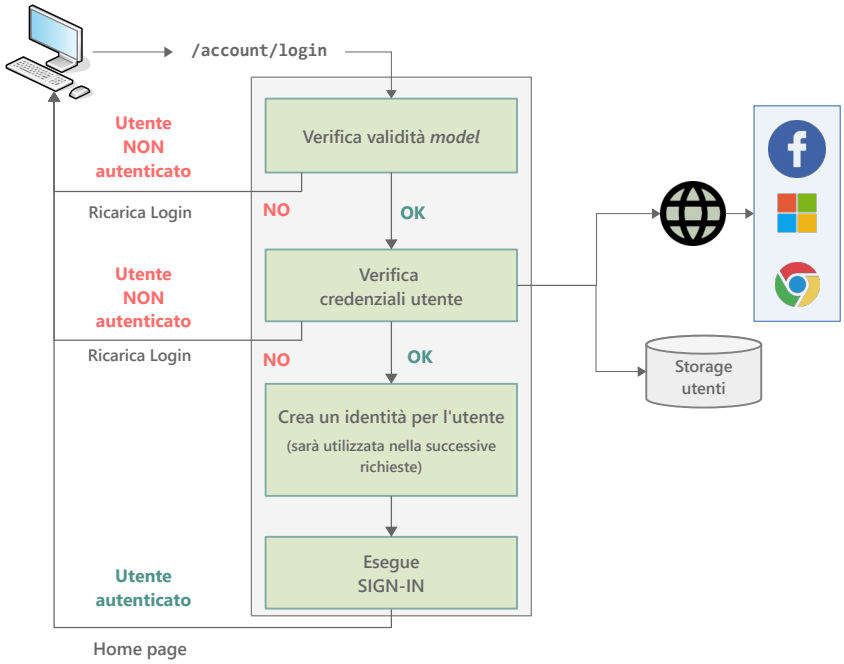
11 Autenticazione

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

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

11.1 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. Dopo la verifica delle credenziali, l'applicazione crea un'*identità* per l'utente e con essa esegue il *sign-in*, completando l'autenticazione.

Dopo che l'utente è stato autenticato, ogni successiva richiesta del *client* farà riferimento all'*identità* creata e sarà riconosciuta dall'applicazione come proveniente da quell'utente in particolare.

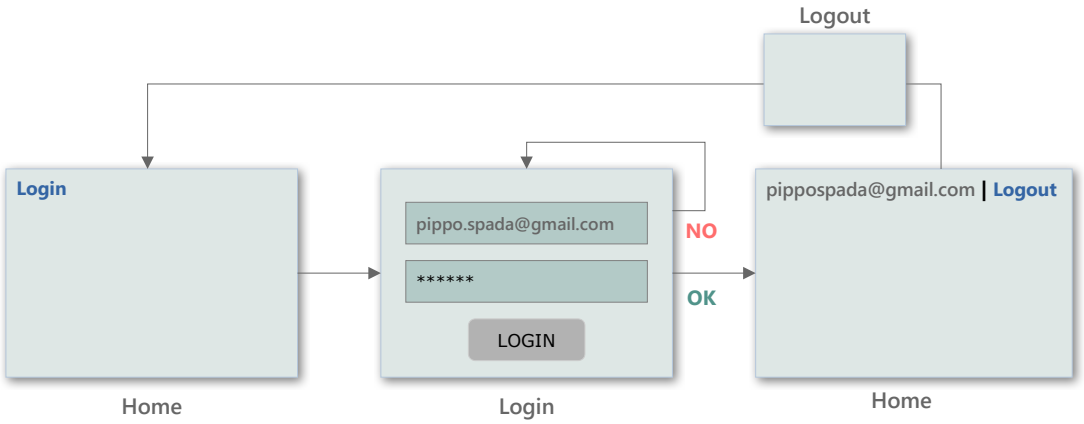
In caso di errori nei dati inseriti o di credenziali non valide, l'utente sarà invitato a correggersi.

11.2 Implementazione del processo di autenticazione

Blazor fornisce un'infrastruttura molto sofisticata, che comprende funzioni di registrazione degli utenti, invio di e-mail di conferma, uso di servizi esterni, etc. Di seguito mi limito a mostrare un'implementazione di base, che richiede:

- La configurazione in **Program.cs** del servizio di autenticazione.
- L'accesso all'oggetto `HttpContex`; questo oggetto fornisce le informazioni sulla richiesta in corso e consente di creare l'*identità* dell'utente e autenticarlo.

L'applicazione è composta da tre pagine: **Home**, **Login** e **Logout**, con quest'ultima che non visualizza alcun contenuto.



La **Home** visualizza una tra due alternative:

- Se l'utente è anonimo: un *link* alla pagina **Login**.
- Se l'utente è già autenticato: l'e-mail dell'utente e un *link* alla pagina **Logout**.

La pagina **Login** richiede le credenziali dell'utente ed esegue l'autenticazione. Se questa ha successo, il *client* viene dirottato alla **Home**, in caso contrario l'utente dovrà re inserire le credenziali.

La pagina **Logout** rende l'utente nuovamente anonimo e dirotta il *client* alla **Home**.

11.2.1 Credenziali degli utenti

Le credenziali degli utenti sono memorizzate nella tabella **Utenti** di un database. Alla tabella corrisponde la classe `Account`:

```
public class Account
{
    public int AccountId { get; set; }          // -> AccountId  INT IDENTITY   NOT NULL
    public string Email { get; set; }           // -> Email      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}";
}
```

11.3 Configurare il servizio di autenticazione

La funzione di autenticazione dev'essere configurata, altrimenti il processo sopra descritto non può essere eseguito. Nel codice che segue, la prima istruzione evidenziata configura il servizio di autenticazione basato su *cockie* (3.7), mentre la seconda istruzione ne stabilisce l'uso nella applicazione.

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

builder.Services.AddRazorComponents();

builder.Services.AddAuthentication().AddCookie();  autenticazione basata sui cockie
...
app.UseAuthentication();

app.UseAntiforgery();
app.MapRazorComponents<App>();

app.Run();
```

(Nota bene: l'istruzione `app.UseAuthentication()` deve essere collocata prima di `app.UseAntiforgery()`, altrimenti il processo di autenticazione/autorizzazione potrebbe non funzionare correttamente.)

Il metodo `AddCookie()` esiste in più versioni e consente di personalizzare il servizio, specificando i parametri del suo funzionamento, tra i quali:

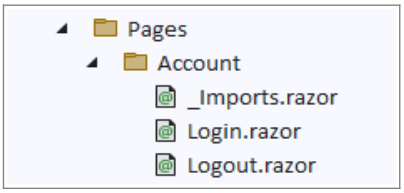
- La durata del *cockie*: alla scadenza, l'utente ritorna automaticamente anonimo.
- La *route* delle pagine di *login* e *logout*. Di default sono: **/account/login** e **/account/logout**.

11.4 Pagine di login e logout

La pagina **Login** definisce un *form* che richiede le credenziali dell'utente – email e password – e le usa per autenticarlo. La pagina **Logout** restituisce l'utente allo stato di anonimo. Di default devono avere le *route* **/account/login** e **/account/logout**.

Entrambe devono usare l'oggetto `HttpContext` della richiesta e importare alcuni *namespace*. A questo scopo è utile collocarle nella cartella **Account** insieme a un file **_Imports.razor** che importa i *namespace* richiesti:

Struttura cartelle



_Imports.razor

```
@using Microsoft.AspNetCore.Authentication
@using Microsoft.AspNetCore.Authentication.Cookies
@using System.Security.Claims
```

11.4.1 Pagina Login

La pagina chiede all'utente di inserire l'email e la password.

```
@page "/account/login"
@inject NavigationManager nav
<PageTitle>Login</PageTitle>
```

```
<EditForm Model="model" OnSubmit="Submit" FormName="login">
    <InputText @bind-Value="model.Email"/>
    <InputText type="password" @bind-Value="model.Password"/>
    <button>LOGIN</button>
</EditForm>
```

Il metodo `Submit()` svolge tre compiti:

1. Verifica le credenziali inserite dall'utente. In caso negativo, viene restituita la stessa pagina, dando modo all'utente di correggersi.
2. Crea un'*identità* per l'utente.
3. Usa l'*identità* creata per eseguire il *sign-in*, cioè l'autenticazione vera e propria. Ogni successiva richiesta proveniente dall'utente sarà associata all'*identità* creata.

```
@code{

    [CascadingParameter]
    HttpContext Context { get; set; }

    [SupplyParameterFromForm]
    private LoginModel model { get; set; } = new();

    UtentiContext db = new UtentiContext();

    void Submit()
    {
        var account = db.Utenti.SingleOrDefault(u => u.Email == model.Email &&
                                                     u.Password == model.Password);
        Verifica credenziali ...                                                    1

        string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
        var identity = new ClaimsIdentity(scheme);
        identity.AddClaim(new Claim(ClaimTypes.Name, account.Email));               2
        var principal = new ClaimsPrincipal(identity);

        Context.SignInAsync(scheme, principal).Wait;                                3

        nav.NavigateTo("/");
    }

    class LoginModel
    {
        public string Email { get; set; } = "";
        public string Password { get; set; } = "";
    }
}
```

(Il metodo `Context.SignInAsync()` è asincrono e restituisce un *task*; la chiamata al metodo `Wait()` sospende l'esecuzione fintantoché il task non è completato. Non è il modo corretto per eseguire un *metodo asincrono*; adotto questa modalità per semplificare il codice.)

11.4.2 Identità e attestazioni (claim)

Un'*identità* (`ClaimsIdentity`) si basa su una o più *attestazioni* (`Claim`), cioè 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 punto 2) creo una *identità* basata su una sola *attestazione*, rappresentata dall'email dell'utente, che sarà utilizzata nella **Home** per identificare l'utente.

11.4.3 Accesso all'oggetto HttpContext

Di default, i *componenti* Blazor non hanno l'accesso all'oggetto `HttpContext` generato per ogni richiesta; a questo scopo è necessario ottenerlo mediante un ***parametro a cascata***, cioè una proprietà decorata con l'attributo `[CascadingParameter]`. È compito di Blazor assegnare al parametro l'oggetto `HttpContext` della richiesta.

11.4.4 Pagina logout

La pagina di **Logout** ha la funzione di eseguire il *sign-out* dell'utente. Questo elimina l'*identità* creata durante il *sign-in* e dunque gli restituisce lo stato di utente anonimo.

```
@page "/account/logout"
@inject NavigationManager nav

@code{
    [CascadingParameter]
    HttpContext Context { get; set; }

    protected override void OnInitialized()
    {
        string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
        Context.SignOutAsync(scheme).Wait();
        nav.NavigateTo("/");
    }
}
```

(Il metodo `Context.SignOutAsync()` è asincrono e restituisce un *task*; la chiamata al metodo `Wait()` sospende l'esecuzione fintantoché il task non è completato. Non è il modo corretto per eseguire un *metodo asincrono*; adotto questa modalità per semplificare il codice.)

11.5 Pagina Home

Il contenuto della **Home** dipende dallo stato dell'utente, anonimo o autenticato (11.2). A tale scopo esiste la proprietà: `Context.User.Identity.IsAuthenticated`, che restituisce `true` se l'utente è autenticato.

```
@page "/"

<PageTitle>Home</PageTitle>

<div>
    @if (Autenticato())
    {
        <span>@Nome() | </span>
```

```
        <a href="/account/logout">Logout</a>
    }
    else
    {
        <a href="/account/login">Login</a>
    }
</div>

@code{
    [CascadingParameter]
    HttpContext Context { get; set; }

    string Nome() => Context.User.Identity.Name;
    bool Autenticato() => Context.User.Identity.IsAuthenticated;
}
```

Nota bene:

- Anche nella **Home** occorre dichiarare il *parametro a cascata* `Context`, necessario per conoscere lo stato dell'utente.
- La proprietà `Context.User.Identity.Name` restituisce il valore assegnato alla chiave `Claims.Name` nel punto 2) del codice che esegue il *sign-in* dell'utente. Nell'esempio è l'email.

12 Autorizzazione

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

L'*autorizzazione* 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 ad accedere a determinate risorse in base all'età.
- **Basata sui *ruoli***: l'utente è autorizzato in base al ruolo che gli è stato assegnato nell'applicazione.

  Ad esempio, in una biblioteca i tesserati potranno 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.

12.1 Abilitare la funzione di autorizzazione

La funzione di *autorizzazione* deve essere attivata in **Program.cs**:

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

builder.Services.AddRazorComponents();

builder.Services.AddAuthentication().AddCookie();

builder.Services.AddAuthorization();

var app = builder.Build();

...

app.UseAuthentication();

app.UseAuthorization();

app.UseAntiforgery();
app.MapRazorComponents<App>();

app.Run();
```

Nota bene: l'*autorizzazione* deve essere configurata dopo l'*autenticazione*.

12.2 Autorizzazione semplice

Nell'*autorizzazione semplice* si usa l'attributo `[Authorize]` per stabilire quali pagine sono accessibili/negate agli utenti: soltanto gli utenti autenticati possono accedere a pagine decorate con `[Authorize]`.

Questo attributo è definito nel *namespace* `Microsoft.AspNetCore.Authorization`, il quale deve essere importato nella pagina (oppure aggiunto al file **_Imports.razor**).

Supponi di aggiungere all'applicazione la pagina **Autenticati**, accessibile soltanto a quegli utenti che hanno effettuato il *login*.

```
@page "/autenticati"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
<h3>Autenticati</h3>
```

12.2.1 Redirezione automatica alla pagina login

Se un utente anonimo tenta di accedere a una pagina decorata con `[Authorize]`, cliccando su un *link* o digitando l'URL nella barra degli indirizzi, sarà rediretto automaticamente alla pagina **Login**. Se questa non esiste viene restituito un errore 404.

12.2.2 Autorizzare il logout

Per coerenza è opportuno negare l'accesso alla pagina **Logout** agli utenti anonimi.

```
@page "/account/logout"
@inject NavigationManager nav
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]

...
```

12.3 Reindirizzare l'utente autenticato alla risorsa richiesta

Considera questo scenario: un utente anonimo tenta di accedere alla pagina **Autenticati**, viene rediretto alla pagina **Login** ed esegue correttamente il *login*; l'applicazione dovrebbe dirottarlo automaticamente alla pagina **Autenticati**, che aveva richiesto inizialmente, non alla pagina **Home**.

A questo scopo, nel dirottare l'utente anonimo alla pagina **Login**, Blazor aggiunge alla *route* **/account/login** una *query string* contenente la pagina che aveva richiesto. (7.6)

Dunque, se l'utente richiede la *route* **/autenticati**, viene dirottato a **Login** con il seguente URL[13]:

/Account/Login?ReturnUrl=/autenticati

Nella pagina **Login** si può accedere al *parametro di query* **ReturnUrl** e dirottare l'utente alla *route* specificata. (7.6)

```
...
@code{
    [CascadingParameter]
    HttpContext Context { get; set; }

    [SupplyParameterFromForm]
    LoginModel model { get; set; } = new();

    [SupplyParameterFromQuery]
    string ReturnUrl { get; set; }

    void Submit()
    {
```

13 Non esattamente: la barra che precede **autenticati** viene codificata con **%2F**.

```
        Verifica credenziali ...

        Crea identità ed esegue il sign-in...
        if (ReturnUrl != null)
        {
            nav.NavigateTo(ReturnUrl);
        }
        else
        {
            nav.NavigateTo("/");
        }
    }
    ...
}
```

12.4 Autorizzazione basata su ruoli

L'*autorizzazione* basata su ruoli parte dalla premessa che gli utenti appartengano a uno o più ruoli, utilizzati per concedere o negare l'accesso a determinate funzioni.

Ad esempio, agli utenti di un registro scolastico 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 dall'esempio precedente, supponi di aggiungere 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 Email { get; set; }           // -> Email      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}";
}
```

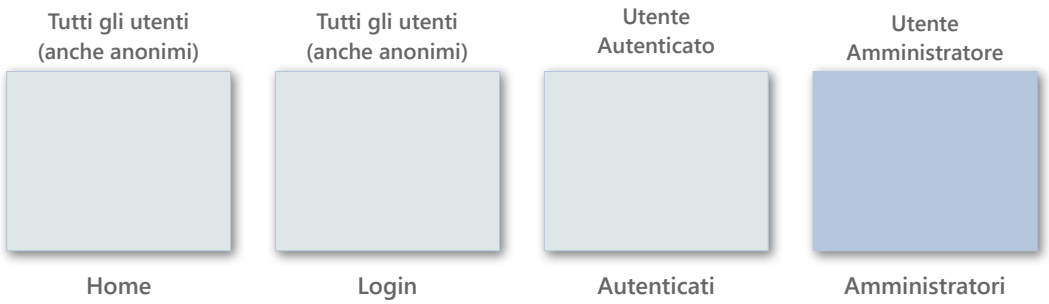
Durante il processo di autenticazione, è necessario che all'*identità* dell'utente sia assegnato il suo *ruolo*. Questo, avente il valore `"Amministratore"` o `"Utente"`, può essere usato nelle altre pagine per concedere o negare l'accesso.

```
@page "/account/login"
@inject NavigationManager nav

...

@code {

    ...
    UtentiContext db = new UtentiContext();
```

```
    void Submit()
    {
        var account = db.Utenti.SingleOrDefault(u => u.Email == model.Email &&
                                                     u.Password == model.Password);
        Verifica credenziali ...

        string scheme = CookieAuthenticationDefaults.AuthenticationScheme;
        var identity = new ClaimsIdentity(scheme);
        identity.AddClaim(new Claim(ClaimTypes.Name, account.Email));

        identity.AddClaim(new Claim(ClaimTypes.Role, account.Ruolo));

        var principal = new ClaimsPrincipal(identity);

        Context.SignInAsync(scheme, principal).Wait;
        nav.NavigateTo("/");
    }
}
```

12.4.1 Regolare l'accesso in base al ruolo dell'utente

Supponi che all'applicazione sia aggiunta la pagina **Amministratori**, accessibile soltanto agli utenti con *ruolo* amministrativo. La figura riepiloga le pagine e gli utenti che possono accedervi (viene omessa la pagina **Logout**, che non visualizza alcun contenuto):

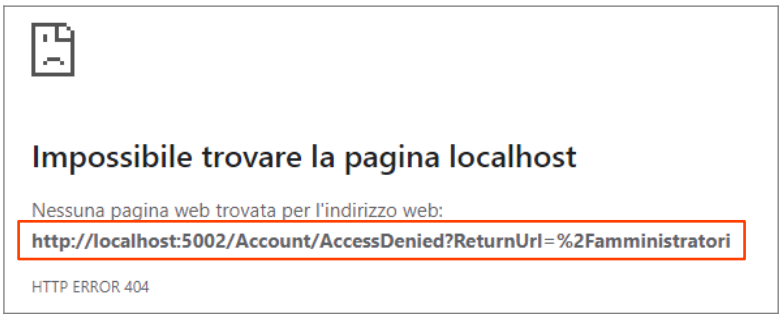


Per restringere l'accesso alla pagina **Amministratori**, occorre specificare il *ruolo* richiesto nell'attributo `[Authorize]`:

```
@page "/amministratori"
@using Microsoft.AspNetCore.Authorization

@attribute [Authorize(Roles ="Amministratore")]

<h3>Amministratore</h3>
```

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

Il tentativo di un utente autenticato, ma non amministratore, di accedere alla pagina **Amministratori** produce la seguente risposta del *server*:



L'URL riportato mostra che il *server* tenta di dirottare il *client* alla pagina **/Account/AccessDenied**, che però non esiste; dunque viene restituito l'errore 404.

L'URL contiene anche una *query string* che assegna alla chiave `ReturnUrl` la pagina richiesta dall'utente.

12.5 Realizzare la pagina di "accesso negato"

È opportuno realizzare la pagina **AccessDenied**, in modo da visualizzare un messaggio d'errore appropriato e invitare l'utente ad eseguire il *login* con un account amministrativo.

```
@page "/account/accessdenied"

<h3>ACCESSO NEGATO</h3>
<h4>Si può accedere a questa pagina solo come 'amministratore'</h4>
<a href="/account/login?ReturnUrl=@ReturnUrl">Login</a>


@code{
    [SupplyParameterFromQuery]
    string ReturnUrl { get; set; }
}
```

La pagina **AccessDenied** definisce un *parametro di query* per memorizzare il `ResultUrl` che gli viene passato automaticamente da ASP.NET. La *query string* viene usata nel *link* che indirizza alla pagina **Login.** (12.3) In questo modo, se e quando l'utente si autentica come amministratore, viene reindirizzato automaticamente alla pagina **Amministratori**.

13 Componenti

Un *componente* incapsula un elemento dell'interfaccia utente. Se volessimo fare un'analogia con le applicazioni *windows.forms*, le pagine corrispondono alle finestre, i *componenti* corrispondono ai controlli: `TextBox`, `ListBox`, etc.

I *componenti*, come le pagine, possono definire dei parametri, cioè proprietà pubbliche decorate con l'attributo `[Parameter]`. Nel codice che usa il componente, si passa il valore a un parametro utilizzando la sintassi degli attributi HTML.

Ad esempio, a sinistra è definito il *componente* `Titolo`, che definisce il parametro `Testo`. A destra il componente viene usato nella home.

```
<h1>@Testo</h1>

@code {
    [Parameter] public string Testo { get; set; }
}
```

```
@page "/"

<Titolo Testo="Home del sito"></Titolo>
```

Possiamo far rientrare i *componenti* in due categorie:

- *Componenti* che implementano una funzione da utilizzare in una specifica applicazione. Sono realizzati per semplificare la struttura delle pagine.
- *Componenti* che implementano una funzione generale, utilizzabile in molte applicazioni. A questa categoria appartengono i *componenti predefiniti* e moltissimi altri *componenti* che è possibile trovare su Internet.

Di seguito fornisco un esempio di entrambe le categorie.

13.1 Visualizzare lo stato dell'utente: anonimo, autenticato

Nell'esempio in (11.5) la **Home** incorpora la visualizzazione dello stato dell'utente – anonimo o autenticato - compresi i *link* alle pagine **Login** e **Logout**

```
@page "/"
<PageTitle>Home</PageTitle>
<div>
    @if (Autenticato())
    {
        <span>@Nome() | </span>
        <a href="/account/logout">Logout</a>
    }
    else
    {
        <a href="/account/login">Login</a>
    }
</div>

@code{
    [CascadingParameter]
    HttpContext Context { get; set; }
```

```
    string Nome() => Context.User.Identity.Name;
    bool Autenticato() => Context.User.Identity.IsAuthenticated;
}
```

Il codice evidenziato incapsula una funzione indipendente dal resto della pagina; può essere utile incapsulare questa funzione in un *componente*: **StatoUtenteView.razor**.

Ciò fatto, la pagina **Home** si riduce all'uso del *componente*:

```
@page "/"

<PageTitle>Home</PageTitle>

<StatoUtenteView></StatoUtenteView>
```

Questa soluzione semplifica la pagina, applica il principio di incapsulamento e facilita il riuso della funzione di visualizzazione dello stato dell'utente in altre applicazioni.

13.2 Validazione: visualizzazione dei messaggi di errore

In (9.2) viene mostrata una soluzione al problema di visualizzazione de messaggi di errore nella validazione di un *form*. Si tratta di una soluzione limitata e ad hoc: non sono gestiti i messaggi generali (che non riguardano un campo di input in particolare) e per ogni campo di input è possibile visualizzare un solo messaggio. È utile adottare una soluzione più generale e versatile, basata su due elementi:

- Una classe in grado di memorizzare più messaggi corrispondenti alla stessa chiave.
- Un *componente* che semplifica la visualizzazione dei messaggi.

13.2.1 Classe StoreMessaggi

La classe `StoreMessaggi` rappresenta un dizionario che ammette chiavi duplicate. L'obiettivo è quello di associare più messaggi allo stesso campo di input. Una chiave stringa vuota identifica i messaggi generali, cioè non riferiti a un campo di input in particolare.

```
public class StoreMessaggi
{
    SortedList<string, List<string>> errori = new(StringComparer.OrdinalIgnoreCase);

    public void Clear()
    {
        errori.Clear();
    }
    public void Add(string chiave, string messaggio)
    {
        List<string> messaggi;
        if (!errori.TryGetValue(chiave, out messaggi))
        {
            messaggi = new();
        }
        messaggi.Add(messaggio);
        errori[chiave] = messaggi;
    }
```

```
    public void Add(string messaggio) => Add("", messaggio);
    public IEnumerable<string> Get(bool soloGenerali = false)
    {
        if (soloGenerali)
            return errori.Where(kv => kv.Key == "").SelectMany(kv => kv.Value);
        return errori.Values.SelectMany(v => v);
    }

    public IEnumerable<string> Get(string chiave)
    {
        if (errori.TryGetValue(chiave, out List<string> messaggi))
            foreach (var msg in messaggi)
            {
                yield return msg;
            }
        else
            yield return "";
    }
}
```

La classe fornisce due metodi `Get()`, i quali restituiscono i messaggi generali, i messaggi riferiti a uno specifico campo di input o tutti i messaggi memorizzati.

13.2.2 Componente di visualizzazione dei messaggi

Il *componente* **MessaggioPer** semplifica la visualizzazione dei messaggi di errore. Definisce due parametri:

- `Messaggi`: rappresenta lo *store dei messaggi*.
- `Campo`: rappresenta il nome del campo di input del quale visualizzare i messaggi. Se non viene assegnato, implica la visualizzazione dei messaggi generali.

Il *componente* ridefinisce il metodo `OnParameterSet()`, eseguito dopo che sono stati assegnati i parametri; il metodo ottiene i messaggi da visualizzare in base al valore assegnato al parametro `Campo`.

```
@if (errori.Count > 0)
{
    <div>
        @foreach (var msg in errori)
        {
            <span class="validation-message">@msg</span><br />
        }
    </div>
}
@code{
    [Parameter] public StoreMessaggi Messaggi { get; set; }

    [Parameter] public string Campo { get; set; } = "";

    List<string> errori = new();

    protected override void OnParametersSet()
    {
```

```
        errori = Messaggi is null
        ? Enumerable.Empty<string>().ToList()   non ci sono messaggi da visualizzare
        : Messaggi.Get(Campo).ToList();
    }
}
```

(Nota bene: il codice che ottiene la lista dei messaggi non può essere collocato in `OnInitialized()`, perché il metodo viene eseguito quando il parametro `Messaggio` non contiene ancora alcun messaggio.)

13.2.3 Uso del componente

Il componente **MessaggioPer** consente di semplificare la visualizzazione dei messaggi di errore:

```
@page "/registra"
@using System.Text.RegularExpressions
@inject NavigationManager nav

<PageTitle>Registra utente</PageTitle>
<h3>Registrazione utente</h3>

<EditForm FormName="registra" Model="Model" OnSubmit="Submit">

    <MessaggioPer Messaggi="messaggi"></MessaggioPer>

    <InputText @bind-Value="Model.Email" placeholder="Email"></InputText>
    <MessaggioPer Campo="Email" Messaggi="messaggi"></MessaggioPer>

    <InputText @bind-Value="Model.Password" placeholder="Password"></InputText>
    <MessaggioPer Campo="Password" Messaggi="messaggi"></MessaggioPer>

    <InputText @bind-Value="Model.ConfermaPassword" placeholder="Conferma password"></InputText>
    <MessaggioPer Campo="ConfermaPassword" Messaggi="messaggi"></MessaggioPer>

    <button>Registra</button>

</EditForm>
```

La validazione del *model* avviene in due fasi. Il metodo `VerificaValiditàModel()` verifica la validità in sé dei campi: valori inseriti, validità della email, password e conferma password uguali. Successivamente viene verificato se la email inserita è già stata registrata in precedenza.

In caso di errori viene aggiunto un messaggio allo *store messaggi*:

```
@code {
    [SupplyParameterFromForm]
    public RegistraUtenteModel Model { get; set; } = new();

    StoreContext db = new();

    StoreMessaggi messaggi = new();


    bool VerificaValiditàModel()
    {
```

```
        if (Model.Email == "")
        {
            messaggi.Add("Email", "Il campo 'Email' è vuoto");
        }
        else if (!ValidaEmail(Model.Email))
        {
            messaggi.Add("Email", "Il formato della email non è valido");
        }
        if (Model.Password == "")
        {
            messaggi.Add("Password", "Il campo 'Password' è vuoto");
        }
        if (Model.Password != Model.ConfermaPassword)
        {
            messaggi.Add("ConfermaPassword", "'Password' e 'Conferma password' non coincidono");
        }
        return messaggi.Count == 0;
    }

    void Submit()
    {
        if (!VerificaValiditàModel())
            return;

        if (db.EsisteUtente(Model.Email))
        {
            messaggi.Add($"L'email '{Model.Email}' è già registrata");
        }

        if (messaggi.Count == 0)
        {
            db.Utenti.Add(new Utente { Email = Model.Email, Password = Model.Password });
            db.SaveChanges();
            nav.NavigateTo("/");
        }
    }
}
```

Validazione e componenti predefiniti

Blazor fornisce dei *componenti* predefiniti, **DataAnnotationsValidator**, **ValidationMessage** e **ValidationSummary**, che automatizzano il processo precedentemente descritto mediante l'uso di *annotazioni* sul *model*.

Purtroppo questi *componenti* non sono in grado di gestire correttamente la validazione di *tipi valore nullabili*., come `int?`, `DateTime?`, etc.