Blazor (https://blazordev.it/articoli/)

**Introduzione, un po’ di storia**

Ricordate come funzionava (e funziona ancora…) **ASP.NET Web Forms**? Una pagina HTML, all’interno di una grande form, con un campo nascosto (il famoso **ViewState**) contenente una stringa crittografata rappresentante lo stato della pagina, cioè il valore corrente di tutti i controlli server nella pagina. Questa form, mandata al server ogni volta che l’interazione dell’utente richiedesse un aggiornamento dell’interfaccia, scatenava l’esecuzione del motore di ASP.NET Web Forms, che aggiornava l’HTML e il valore del ViewState in base agli handler del code behind.

Web Forms cercava solo di automatizzare questo processo, per nascondere i dettagli di funzionamento del protocollo ai programmatori Microsoft che venivano dal mondo Desktop, che erano abituati a ragionare in termini di Form, controlli e Handler di eventi nel code behind.

Questa astrazione portava con se però una grossa inefficenza in termini di scambio dati e di esperienza utente.

La cosa fu mitigata con l’arrivo dell’ **Update Panel**, che rendeva asincrono lo scambio utilizzando AJAX in maniera trasparente al programmatore, risolvendo il problema del refresh.

Quando questa astrazione è diventata ingombrante e abbiamo accettato l’idea di doverci scrivere l’HTML (anzichè generarlo dai controlli server) e controllare l’interazione con il server, utilizzando JavaScript dove necessario, siamo passati a **ASP.NET MVC**

A un certo punto però, per ragioni che avremmo compreso fino in fondo solo anni dopo, il legame tra ASP.NET e IIS è cominciato ad andare un po’ stretto.

Questo portò all’introduzione di [**OWIN**](http://owin.org/), uno standard aperto che definisce una interfaccia tra un web server .NET e una applicazione web.

Con la versione 4 di MVC, fornendo di fatto la possibilità di far girare applicazioni ASP.NET MVC su web server diversi da IIS.

 Ma la vera ragione che portò a un momento di rottura fu il fatto che ASP.NET non era stato creato pensando agli scenari Cloud, e Microsoft stava per puntare tutto sul suo Cloud: **Microsoft Azure**.

Nel 2016 fu rilasciata la prima versione di **ASP.NET Core**. Con l’occasione sono stati unificati i framework MVC e WEB API, abbandonando definitivamente Web Forms. Grazie al rilascio contestuale di **Kestrel**, un web server cross-platform compatibile OWIN, è possibile far girare .NET Core su Windows, Linux e MacOS, aprendo scenari incredibili. Il .NET Framework è ancora lì (per il momento), accanto a .NET Core, utilizzando le Standard Libraries per condividere il codice tra i due mondi.

Nello sviluppo web di solito è il client che apre una connessione al server, fa una richiesta HTTP e ottiene una risposta HTTP, con la quale la connessione viene chiusa. Spesso però abbiamo bisogno di sapere se lato server è cambiato qualcosa.

Possiamo sicuramente fare delle richieste periodiche in maniera automatica per verificare se qualcosa è cambiato, ma questa operazione, che viene chiamata **Polling**, è decisamente dispendiosa.

Una possibile alternativa è il **Long Polling**, che consiste nel mantenere attiva una connesione dal client al server in uno stato *Pending*: quando ci sarà un aggiornamento lato server, questo potrà essere comunicato utilizzando la connessione aperta. La risposta chiuderà la connessione, sarà quindi compito del client aprirne un’altra, ripetendo il processo. Sicuramente meno dispendioso del Polling, ma i browser moderni supportano delle valide alternative.

l’utilizzo dei **Server-Sent Event**, che ci permette di ricevere aggiornamenti dal server restando in ascolto su un particolare endpoint. In pratica registriamo una callback JavaScript utilizzando uno speciale oggetto chiamato **EventSource**, a quel punto il sever, utilizzando il content-type text/event-stream può invocare questa callback senza che ci siano attese lato client.

I Server-Sent Event sono però monodirezionali (dal server al client), se abbiamo bisogno di connessioni bidirezionali senza attese possiamo utilizzare la specificica **[WebSocket](https://tools.ietf.org/html/rfc6455)**.

Tutto bello, ma abbiamo bisogno di sapere se browser e server supportano la scelta che facciamo. Ed è proprio qui che entra in gioco **SignalR**: in maniera completamente trasparente al programmatore permette di utilizzare lo strumento migliore messo a disposizione da client e server.

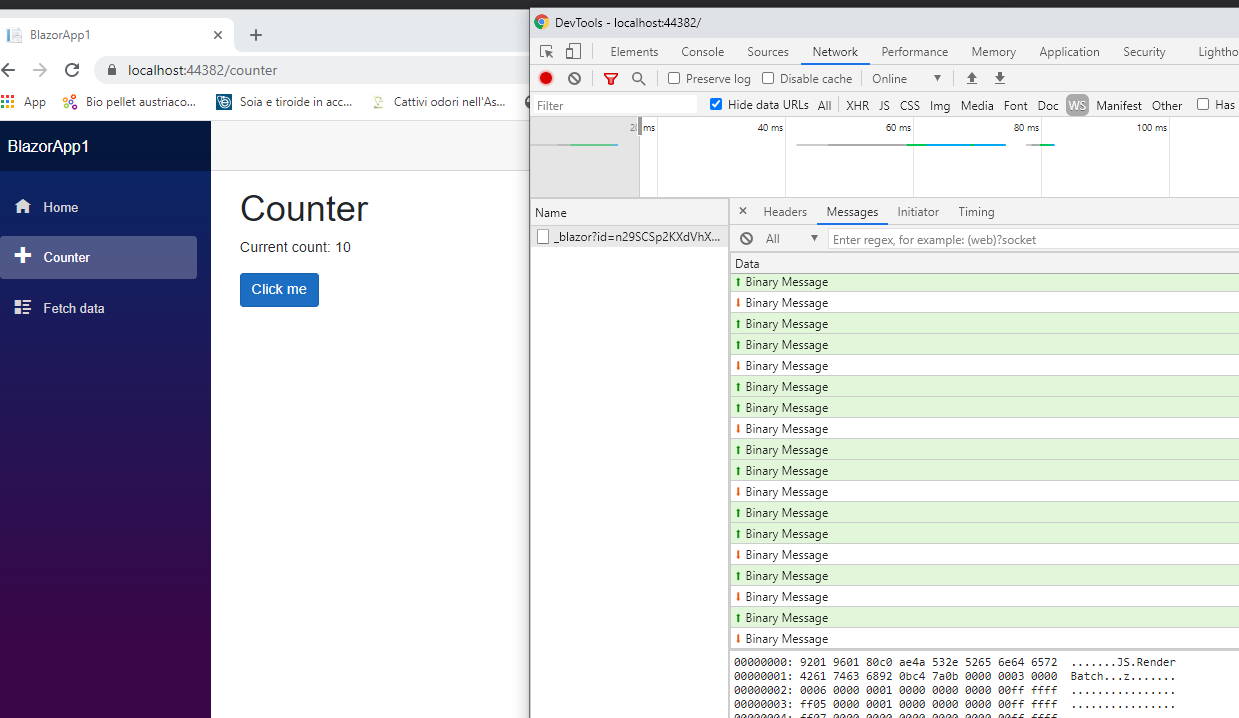
Questo significa che, se disponibile, SignalR utilizzerà le WebSocket, altrimenti proverà con i Server-Sent Events. Nel caso nessuno dei due sia disponibile, verrà utilizzato il Long Polling.

Il programmatore deve solo preoccuparsi di definire una o più classi **Hub**, che verranno utilizzate come endpoint per la comunicazione.

**Blazor server**

Gli aggiornamenti dell’interfaccia avvengono lato server, e vengono inviati al client utilizzando SignalR.

Se cliccate sul Filtro WS (WebSockets) del TAB Network, selezionate la WebSocket \_blazor, cliccate sulla voce di menu *Counter* e poi sul pulsante *Click me*, vedrete quanto segue:

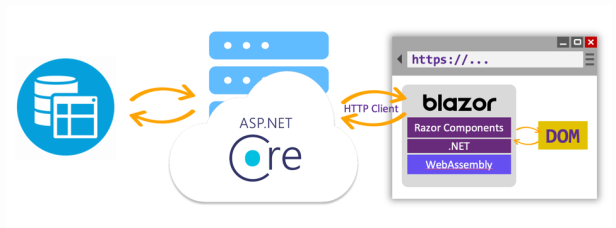


Nel Tab Messages, potete vedere lo scambio dati bidirezionale tra il client e il server: il click sul pulsante ha inviato tramite la WebSocket la richiesta al server, che ha aggiornato il valore di un contatore, rigenerato l’HTML, confrontato con l’HTML precedente e inviato il differenziale al client mediante la WebSocket. A quel punto il codice presente nello script blazor.server.js ha aggiornato il DOM del browser!

La sfida tecnologica si gioca però sul tavolo **WebAssembly**, e nel prossimo articolo vedremo come sia possibile far girare il .NET Core Framework direttamente nel browser, senza nessun plug-in aggiuntivo.

**Blazor WebAssembly**

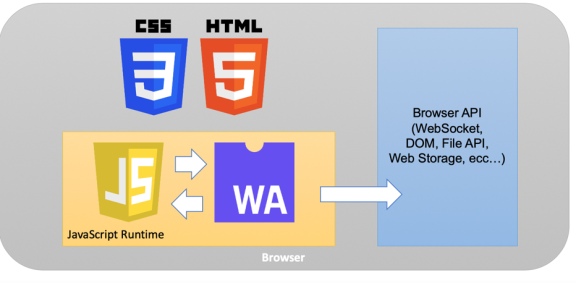
Dopo Blazor Server vediamo come funziona la versione WebAssembly, attualmente in preview ma già sperimentabile in attesa del rilascio di Maggio 2020.



Lo sviluppo web ci ha abituato a pensare che l’unico modo di eseguire codice nel browser sia JavaScript, ed è stato così fino all’arrivo della specifica **WebAssembly** e della sua adozione nei principali browser moderni.

Ad altissimo livello, possiamo immaginare il browser come un contenitore in cui possiamo definire la nostra interfaccia utilizzando HTML e CSS, interagendo tramite JavaScript con le API del browser (DOM, WebSocket, File API, Web Storage, ecc.). JavaScript viene eseguito all’interno di un ambiente controllato, chiamato **JavaScript Runtime**, che fornisce **un unico thread** per ogni tab o dominio che viene visualizzato.

WebAssembly si piazza esattamente nello stesso runtime, condividendo quindi gioie e dolori di questo ambiente e la possibilità di interagire con JavaScript. Questo significa che WebAssembly ci permette di fare tutto quello che facciamo con JavaScript e, laddove ci serva integrare codice legacy o utilizzare librerie JavaScript già pronte, è possibile invocare una funzione JavaScript da un metodo C#. Sicuramente interessante anche la possibilità di fare il contrario, invocare cioè da JavaScript una funzione esposta da WebAssembly.

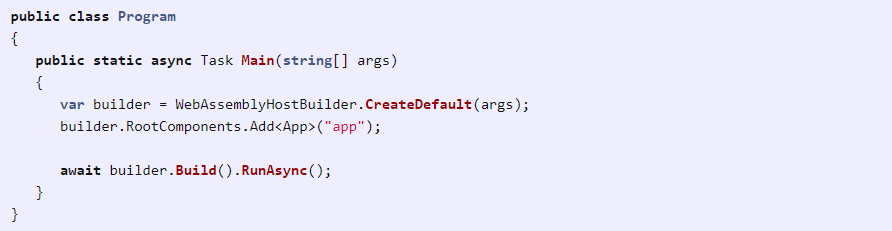


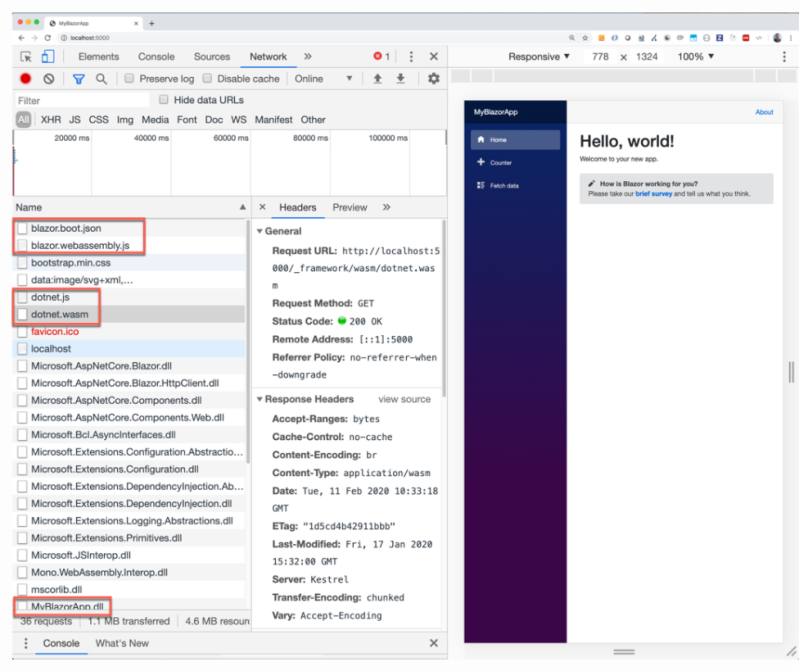
Lo standard WebAssembly, detto anche **WASM**, fornisce un formato binario nato con lo scopo di eseguire nel browser codice ad una velocità simile a quella del codice nativo. Blazor ci permette di eseguire nel browser codice **IL (Intermediate Language)**, generato a partire da C#.I file contenenti codice WebAssembly hanno estensione WASM.



Passiamo alla pratica! Creiamo un progetto Blazor WebAssembly e chiamiamolo MyBlazorApp con il comando dotnet new blazorwasm -o MyBlazorApp.

Quello che vediamo nel file Program.cs è la configurazione del web server che servirà i file e la registrazione del componente principale.

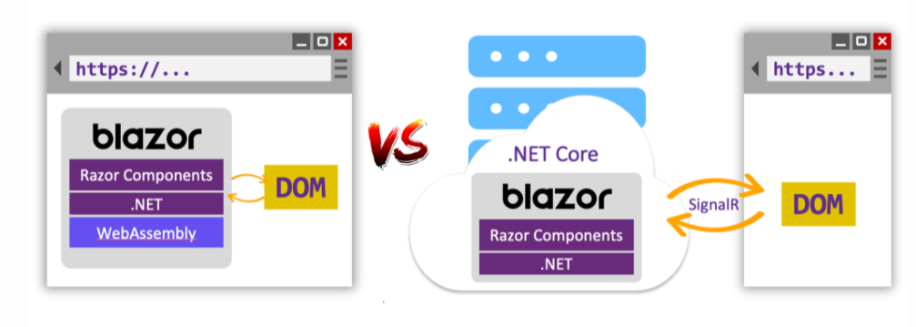




Sì, non ci sono errori, quelle che vedete sono delle DLL scaricate nel browser! Non lasciate venire alla mente ricordi molesti (non è Silverlight…), ma grazie a **dotnet.wasm** possiamo eseguire le librerie .NET standard del progetto, compreso il nostro MyBlazorApp.dll. Nel file blazor.boot.json trovate l’elenco degli assembly che devono essere caricati, utilizzato dallo script blazor.webassembly.js per inizializzare l’applicazione. L’applicazione sta girando nel client, niente WebSocket e niente back-end. Se pulite la sezione Network e cliccate sul counter dell’applicazione, vedrete che nessuna chiamata al server viene fatta. Se invece vi spostate su Fetch data, potete vedere la chiamata HTTP per il recupero dei dati visualizzati nella tabella, ma il rendering dell’HTML viene fatto client-side.

Nel prossimo articolo faremo un confronto tra i due, analizzando pro e contro di ogni versione e in quali contesti una delle due risponde meglio ai nostri requisiti.

**Quale Blazor sceglere?**



Per poter funzionare, la versione WASM richiede il supporto del browser, che ad oggi è garantita su Edge, Firefox, Safari, Opera e i principali browser mobile.



D’altro canto, l’utilizzo di SignalR in Blazor Server, *su numeri veramente importanti*, può creare un limite di scalabilità, che va gestito opportunamente. Il problema nasce perchè SignalR richiede che sia lo stesso processo server a gestire una specifica connessione, quindi la scalabilità di SignalR è legata a quella della singola macchina. Se aggiungiamo macchine per gestire più utenti, dobbiamo assicurarci che le richieste dei singoli utenti arrivino alla stessa macchina da cui la connessione è iniziata, cosa che può essere gestita, per esempio, con i sistemi di *affinity* dei Load Balancer.

Una delle principali caratteristiche di una Single Page Application è quella di elaborare sul client l’interfaccia utente,  In Blazor Server questa operazione non è possibile, perchè l’elaborazione avviene sul server e richiede una connessione stabile per il funzionamento.

In Blazor WebAssembly possiamo invece supportare gli **scenari off-line**, lavorare cioè in assenza di connessione. Il framework ci fornisce la possibilità di lavorare sul client e interrogare il server solo quando abbiamo bisogno di recuperare o inviare dati, utilizzando l’HTTP Client.

Supportare veramente uno scenario off-line però è una cosa più complessa, richiede la capacità di rilevare la presenza o meno della connessione e lavorare sui dati anche in assenza di essa.

i **dati sensibili,** Utilizzando Blazor Server tutto resta sul server e eventuali dati sensibili, come dei token di accesso a servizi terzi come quelli di Azure, non vengono proprio inviati al client.

La **dimensione** dei dati da scaricare in Blazor WebAssembly, al primo download, è molto più grande di quella necessaria per Blazor Server.

In una applicazione enterprise attendere qualche secondo per il primo download non è un grosso problema, mentre se decidessimo di usare Blazor per un sito o una applicazione pubblica, il primo download può impattare sull’esperienza utente. (Blazor server per siti? E balzor webassembly per app enterprise).

Strettamente collegata alla questione on-line/off-line c’è anche la questione **latenza**: dover richiamare il server per ogni operazione di aggiornamento dell’interfaccia può impattare sulle prestazioni e sull’esperienza utente in caso di connessione non stabile.

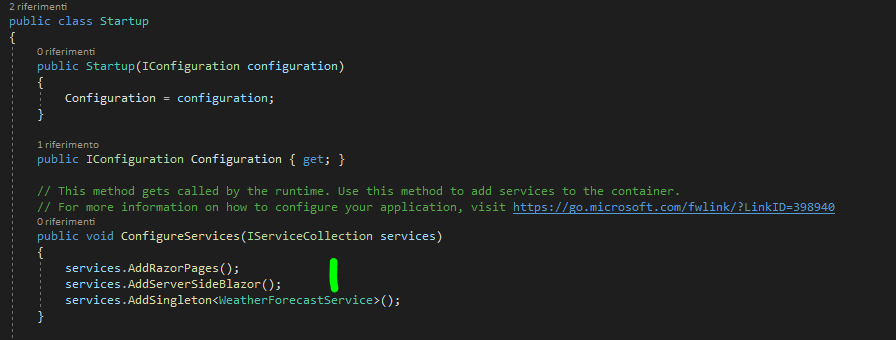
Con Blazor Server stiamo lavorando con una vera e propria applicazione ASP.NET Core, il tutto avviene sul server e quindi basta mettere un punto di interruzione nel codice per eseguire le fasi di debug della nostra applicazione.

Su questo Blazor WebAssembly non riesce, ad oggi, a tenere il passo. Il debug avviene nel browser, in un tab separato, abilitando opportunamente il tutto con dei comandi specifici.

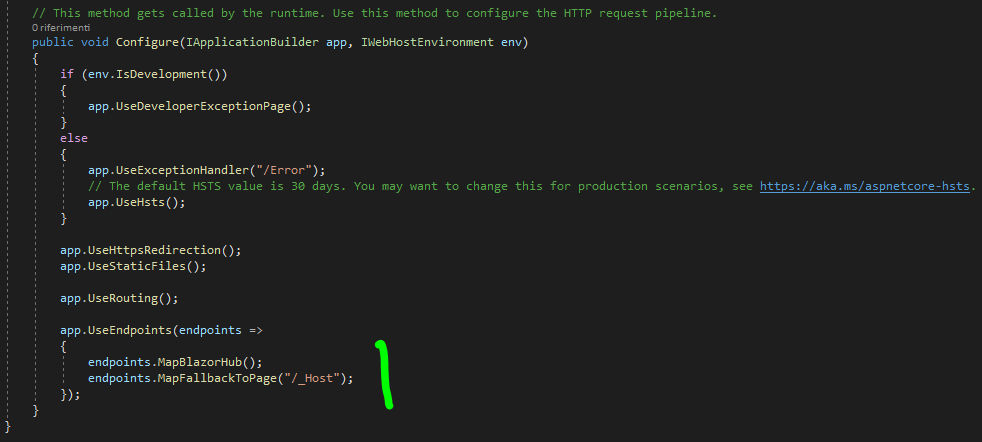
**Crea una SPA: struttura**

L’idea è quella di guidarvi nella realizzazione di una applicazione enterprise con Blazor, partendo da una classica CRUD.

La configurazione di Blazor Server viene fatta nella classe Startup, dove nel metodo ConfigureServices viene aggiunto il supporto alle pagine Razor (services.AddRazorPages()) e ai componenti Blazor (services.AddServerSideBlazor()).



Nel metodo Configure, vengono poi impostati gli endpoints di Blazor: l’Hub SignalR (endpoints.MapBlazorHub()) e l’indirizzo verso il quale navigare in caso di pagina non trovata (endpoints.MapFallbackToPage("/\_Host")).



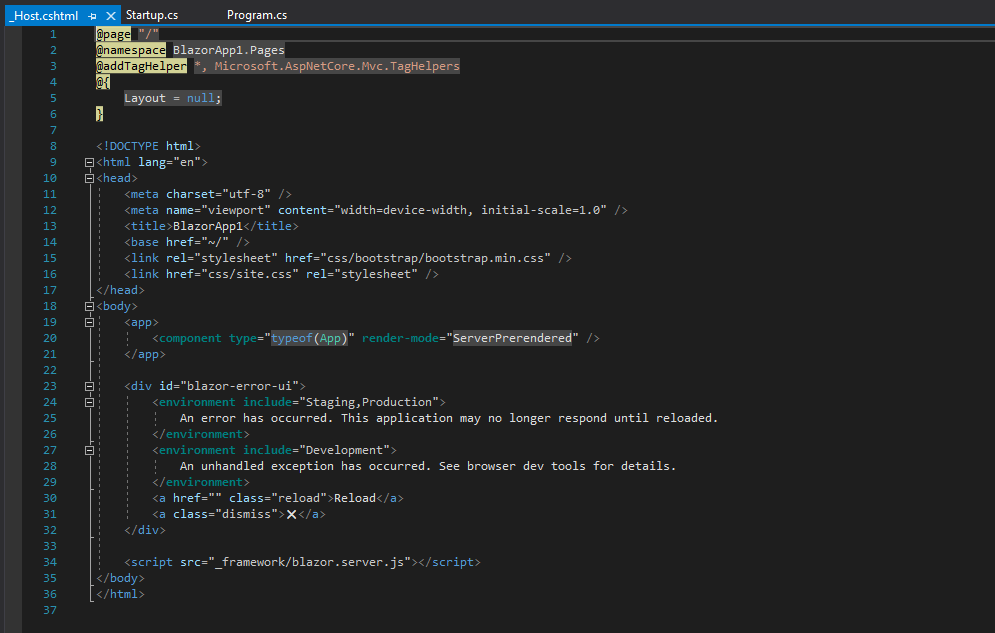
in Blazor Server la definizione dell’interfaccia parte da una singola pagina (da cui il nome Single Page Application), chiamata solitamente \_Host.html, che contiene l’HTML di base e un primo componente.

Come potete vedere, niente di diverso da una classica pagina HTML, fatta eccezione per alcuni elementi. Subito sotto il body potete vedere l’utilizzo di un componente Blazor, denominato appe corrispondente al file App.razor presente nel progetto.

Questo componente sarà elaborato lato server (render-mode="ServerPrerendered") e sarà la radice dell’albero di componenti che costituirà la nostra interfaccia.

Sotto questo componente c’è un blocco opzionale che in caso di errore mostra un messaggio che, in fase di staging o produzione, invita a ricaricare la pagina, nel caso invece siate in fase di sviluppo, indica di verificare gli errori nella console del browser.

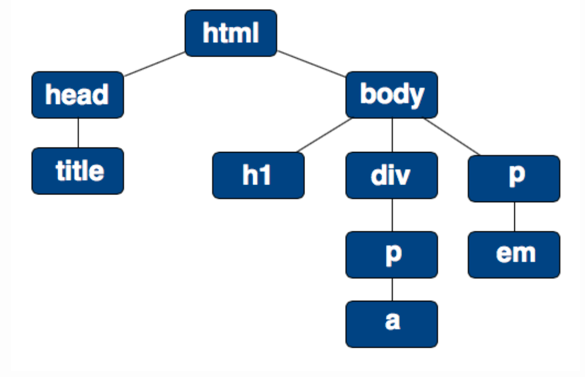
Il file js scarica nel browser la componente client di Blazor Server, responsibile della comunicazione con il server mediante SignalR.



**Crea una SPA: componenti**

Un componente conterrà la definizione del pezzo di interfaccia che lo interessa, l’HTML nel caso di applicazioni Web, e definirà il codice per gestire questo pezzo di interfaccia, limitandosi a visualizzare i dati richiesti e catturare l’input dell’utente.

Come abbiamo visto negli articoli precedenti, sia in Blazor Server che Blazor WebAssembly, il rendering della UI parte da una singola pagina HTML in cui c’è un primo componente.



Una buona strategia, in caso non si sia sicuri di come dimensionare un componente, è quella di partire dal crearne uno e poi rifattorizzarlo in più componenti man mano che cresce.

Tutti i framework di front-end, compreso Blazor, ci permettono di creare componenti parametrizzabili, in modo da non dover avere nessun riferimento esplicito al componente che lo ospita, ma solo ricevere da esso **dati in input** ed emettere **eventi in output** a cui potrà eventualmente sottoscriversi.

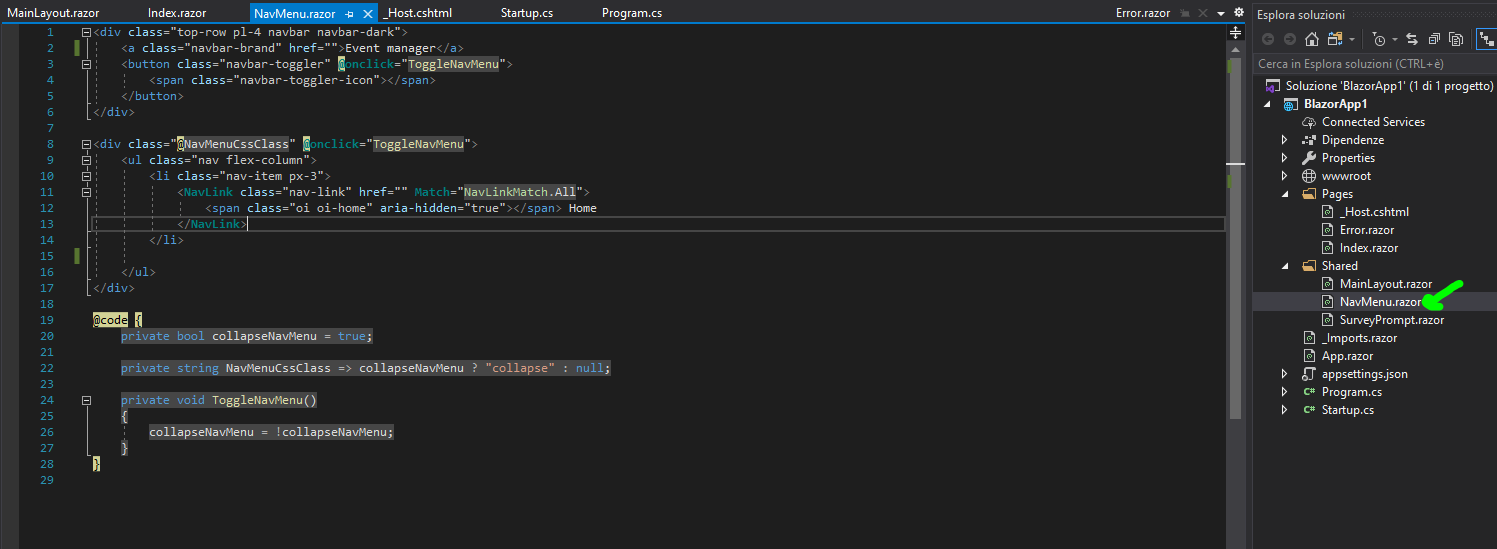
Passiamo alla pratica e vediamo come creare un componente in Blazor.

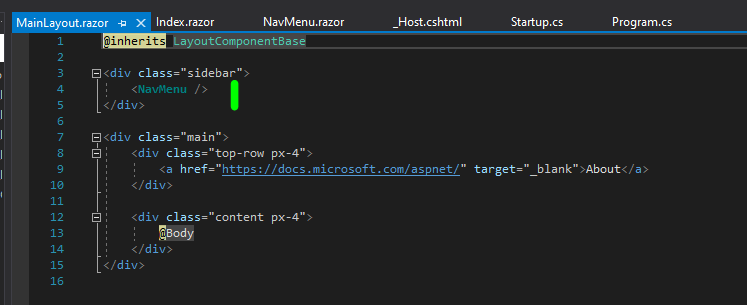
Vedremo che cosa sono le pagine in Blazor (cartella pages) per il momento immaginiamoli come degli speciali componenti contenitori che rappresentano una pagina della nostra applicazione.

Tecnicamente in Blazor un componente è un file con estensione .razor, che contiene il markup, le direttive Razor per la gestione del markup e il binding dei dati

Il nome del file definisce anche il **selettore** con cui possiamo chiedere al framework di istanziare per noi il componente e renderizzarlo nel punto in cui abbiamo utilizzato il selettore.

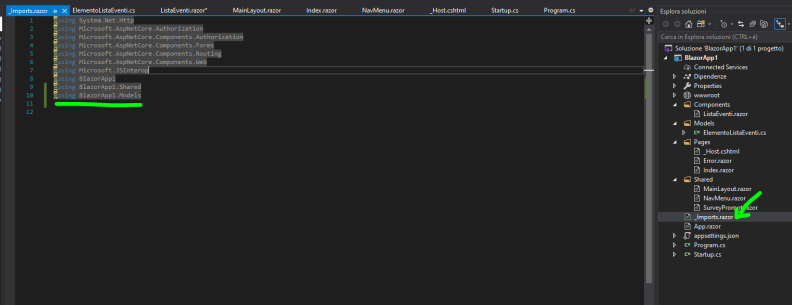
l file NavMenu.razor, che abbiamo già visto nell’articolo precedente, è un ottimo esempio: contiene il markup (HTML e classi CSS per la definizione della classica [Navbar](https://getbootstrap.com/docs/4.0/components/navbar/) di Bootstrap), le direttiva e le istruzioni Razor (quelle con il simbolo @ per capirci) e il blocco @code dove è definito il codice del componente. Per poter utilizzare questo componente, nel file MainLayout.razor troviamo il selettore <NavMenu />:

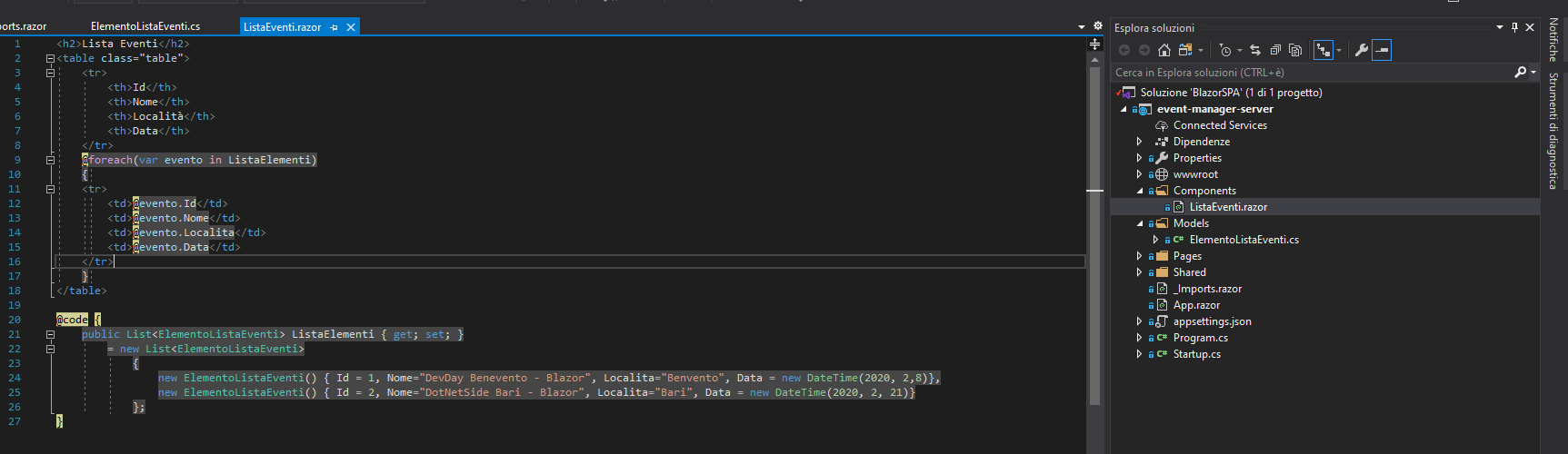


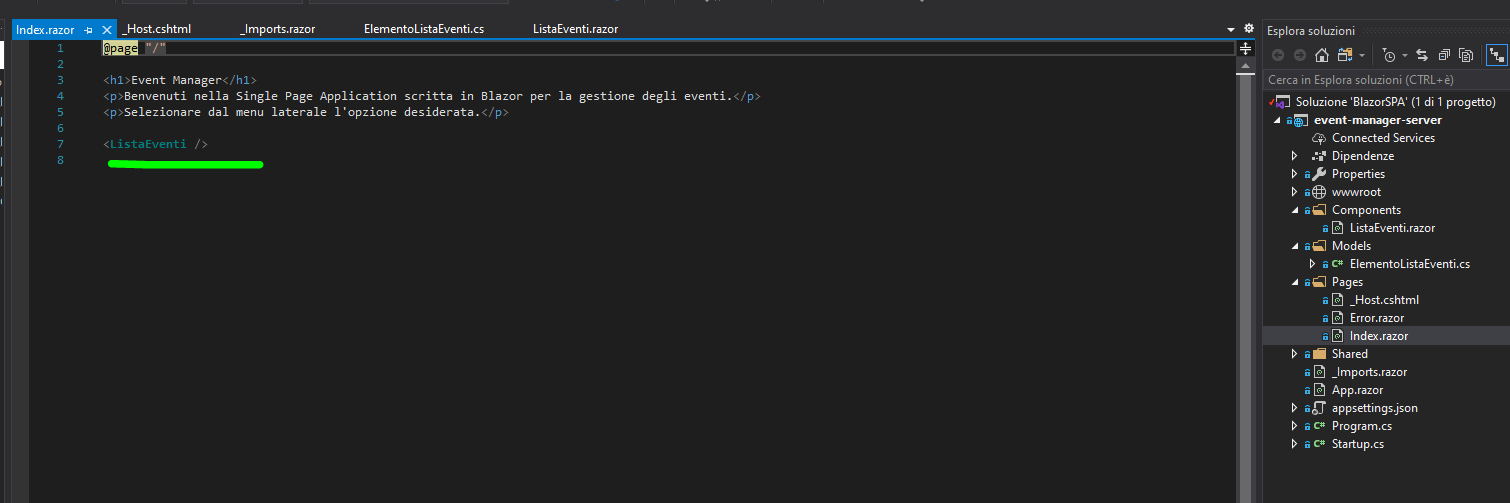


potremmo realizzare un componente che visualizza la lista degli eventi in formato tabellare.

In questo file è possibile aggiungere i namespace globali:







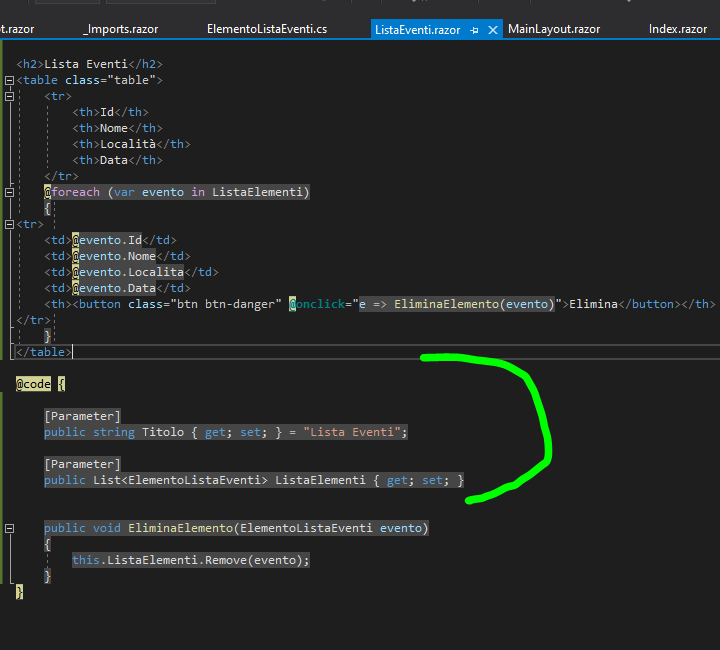
**Creare una SPA: componenti riutilizzabili**

la parametrizzazione non garantisce necessariamente il disaccoppiamento tra due componenti, che potrebbero usare i parametri per passarsi un riferimento alla propria istanza.

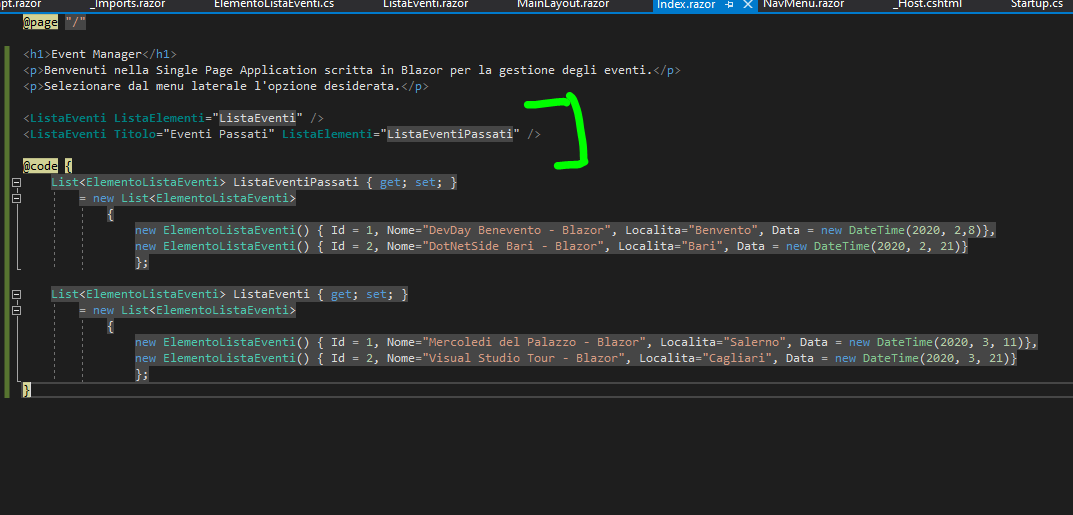
Potremo passare this (ossia il padre) al componente figlio, in modo da poter rendere possibile la comunicazione, ma questo determina che se vogliamo riutilizzare il figlio ci dobbiamo portare dietro il padre. Come risolviamo allora il problema? Basta progettare i nostri componenti per essere trattati come delle **Black Box** che prendono **in Input i dati** con cui devono lavorare e forniscano **in Output eventi** a cui sottoscriversi per poter **reagire** all’interazione dell’utente. In questo modo è il componente stesso a **esporre il contratto** di comunicazione, rendendosi autonomo dal suo utilizzatore. Vediamo come questo può essere fatto in Blazor.

In un componente Blazor i parametri sono delle proprietà pubbliche decorate con l’attributo [Parameter], senza differenza tra parametri di input e parametri di output in termini dichiarativi.

Andiamo alla pratica, passiamo il parametro titolo e lista eventi al componente di prima:



E lo chiamiamo così sulla pagina ospitante:



Dopodiché aggiungiamo un evento per cancellare una riga:



Niente di complesso, la direttiva @onclick ci permette di andare a catturare il click del mouse e di invocare un metodo, che in questo caso è una **lambda expression**, perchè abbiamo bisogno di passare un parametro. L’evento di click ci fornisce un oggetto MouseEventArgs (e =>) con le informazioni sull’evento scatenante l’azione.  in questo caso non lo utilizziamo ma potrebbe tornare utile in scenari differenti (volendo possiamo sostiture e => con () =>). Eseguendo il codice vedremo funzionare il nostro pulsante.

Posso creare in alternativa un altro parametro del componente, utilizzando il tipo EventCallback<T>, una struttura offerta dal framework che permette al componene di sollevare un evento e al suo utilizzatore (il componente padre) di sottoscriversi al verificarsi dello stesso:

E’ consuetudine utilizzare la nomenclatura On<nome evento> per definire gli eventi del componente, nel nostro caso OnElimina

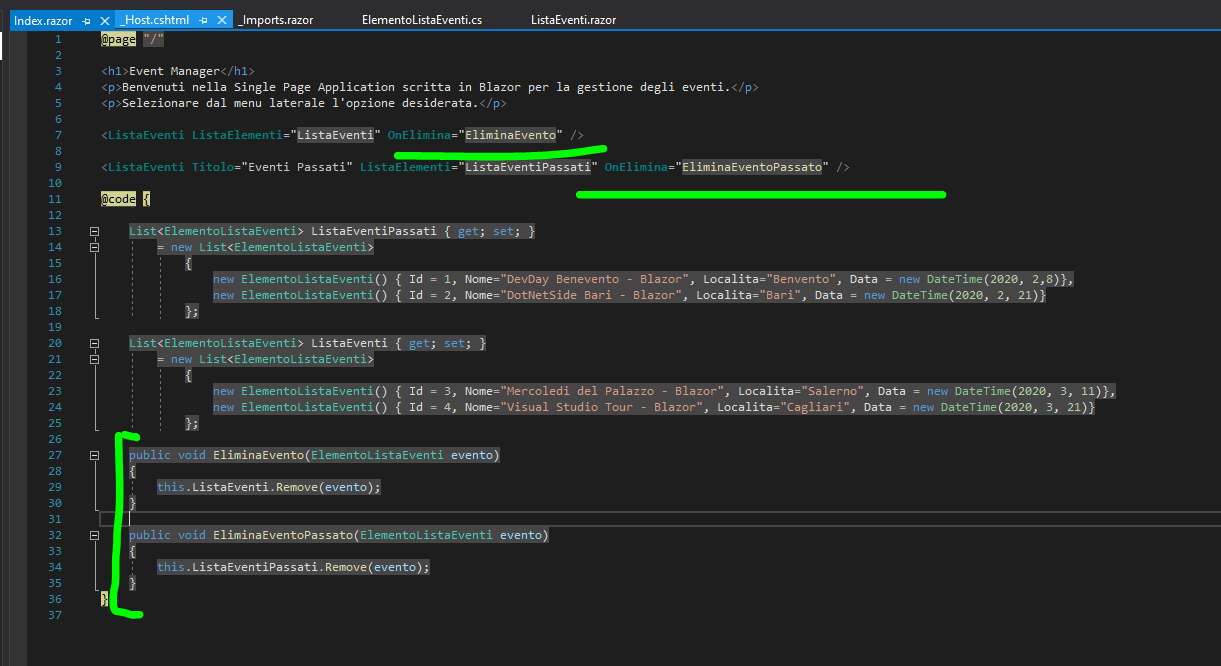
Il tipo EventCallback<T> ci espone il metodo InvokeAsync(T) grazie al quale al click del pulsante possiamo sollevare l’evento definito, in questo modo tutti i sottoscrittori saranno invocati:

Eseguendo il codice vedremo che il risultato non cambia, ma dal punto di vista della parametrizzazione il nostro componente ListaEventi è sempre più vicino a un generico elemento lista che possiamo riutilizzare in più contesti:

qui il componente:



Qui il client (come si può notare il gestore eventi è spostato sul client e non è interno al componente come l’esempio di prima:



**Creare una SPA: pagine e navigazione**

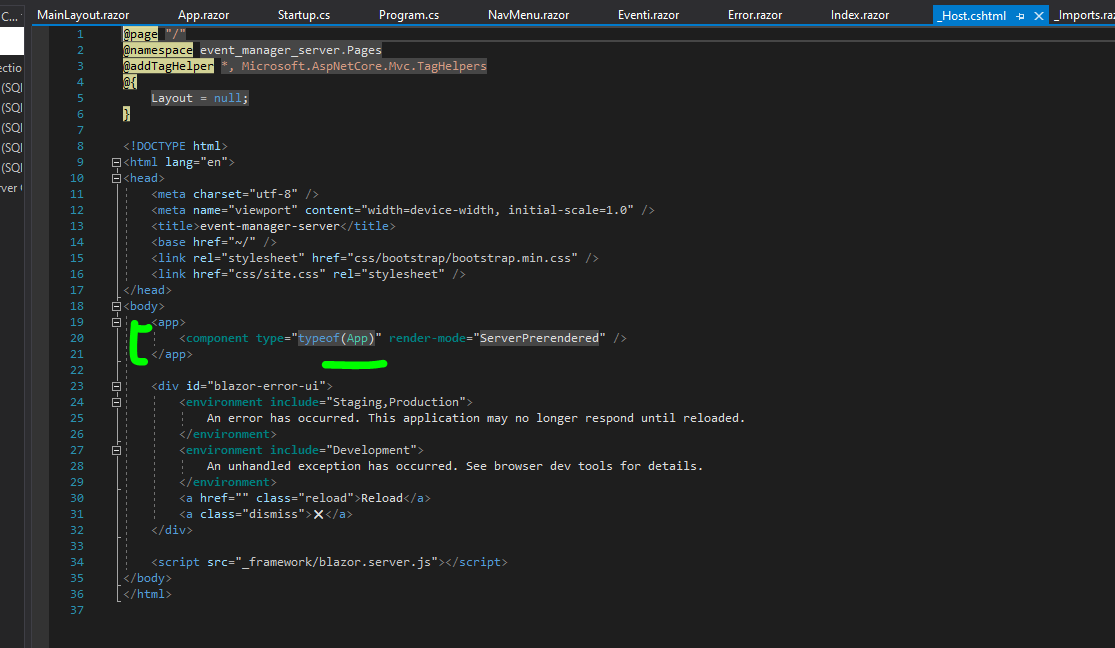
In una Single Page Application la navigazione non avviene lato server (abbiamo una sola pagina, quindi continuerebbe a restituirci la stessa), ma lato client, dinamicamente nel browser. Questo è possibile grazie a uno speciale componente della nostra applicazione che si occupa di gestire il meccanismo di **routing**.

In Blazor, così come nella maggior parte dei framework di front-end per Single Page Application, **una pagina è un componente destinatario di una navigazione**.

Questo significa che da qualche parte c’è una configurazione che mette in relazione due elementi principali: il patha cui vogliamo arrivare e il componenteche vogliamo visualizzare. In framework come Angular questa configurazione va esplicitata nel modulo di routing, in Blazor è molto più semplice: basta aggiungere nel componente desiderato una direttiva @page "<path>" per spiegare al gestore della navigazione che, quando l’utente vuole navigare verso quel percorso, deve visualizzare il componente che contiene la direttiva corrispondente.

Guardiamo ora il motore di routing:

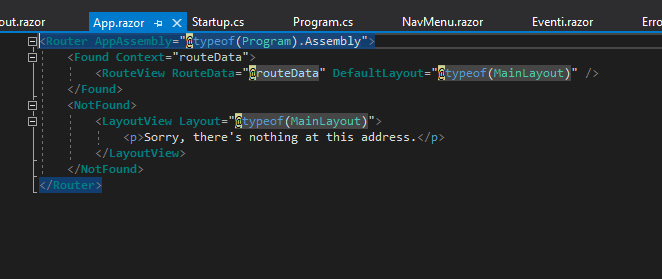
 il nostro albero di elementi parte dal componente <app />



App in realtà è il componente principale e contiene il componente router Ha senso che sia qui, in modo che possa intercettare tutti gli eventi di navigazione e impedire che venga fatta una richiesta HTTP al server, gestendo localmente la richiesta dell’utente, ’attributo AppAssembly indica al motore di routing in quale assembly .NET debba cercare le pagine, in modo che possa rilevare le direttive @page e registrare le coppie path/componente.

A questo punto, quando l’utente cercherà di navigare verso un certo percorso, possono succedere due cose: il path ricercato viene trovato oppure no.

Nel caso venga trovato la sezione <Found> del router naviga verso il componente corrispondente passandogli i dati di navigazione (RouteData="@routeData") e indicando un **Layout** (DefaultLayout="@typeof(MainLayout)").:

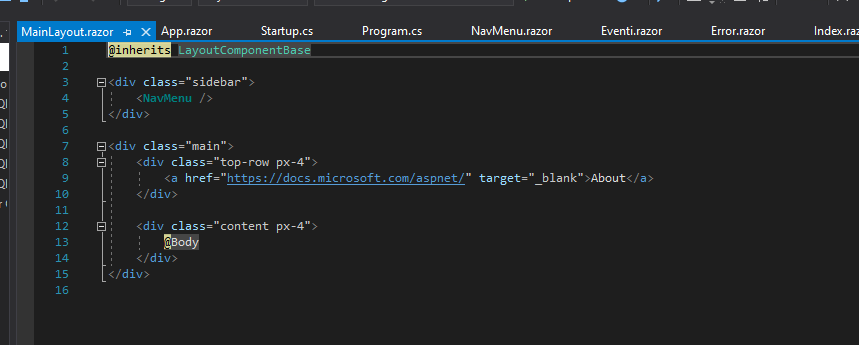


Ma che cos’è un **Layout**? Se avete avuto esperienze con ASP.NET WebForms, stiamo parlando dello stesso concetto di **MasterPage**, se invece venite da ASP.NET MVC è il corrispondente Layout di Razor che ben conoscete. Tecnicamente si tratta di un componente che troviamo nella cartella Shared, nel file, MainLayout.razor:

Un file di layout estende una classe astratta base LayoutComponentBasee definisce il markup della struttura della nostra applicazione che non cambia durante la navigazione, indicando con il segnaposto @Body il punto esatto in cui vogliamo che venga renderizzato il componente destinatario della navigazione, o il messaggio di cortesia del componente NotFound in caso nessun componente risponda al percorso richiesto.

Come potete vedere nella sezione sidebar (<div class="sidebar"><NavMenu /></div>) viene utilizzato un componente separato che rappresenta il nostro menu di navigazione, che trovate nel file NavMenu.razor della cartella Shared. Andiamo ad aggiungere al nostro menu la voce Eventi,

Qui potremmo utilizzare direttamente un elemento HTML di tipo anchor (<a href="/eventi" />), ma il componente <NavLink></NavLink> del framework gestisce per noi l’applicazione della classe active sull’elemento nel caso in cui il percorso di navigazione concida esattamente (Match="NavLinkMatch.All") con il valore di href.



**Creare una SPA: gestione form**

<https://blazordev.it/blazor/spa/2020/03/03/spa-gestione-form.html>

**Creare una SPA: integrazione back end con blazor**

<https://blazordev.it/blazor/spa/2020/03/06/spa-integrazione-backend.html>

**Interoperabilità con JavaScript**

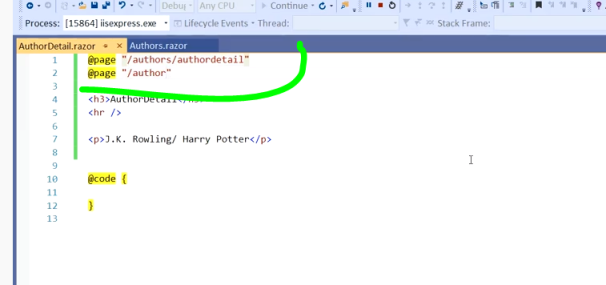
https://blazordev.it/blazor/advanced/2020/03/13/interoperabilita-javascript.html

**Chiamate .NET da JavaScript**

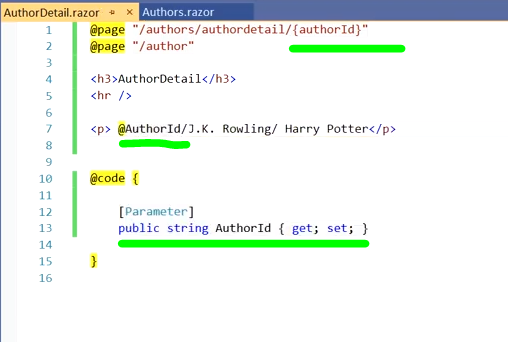
https://blazordev.it/blazor/advanced/2020/03/13/interoperabilita-javascript.html

**Routing**

Possiamo avere multiple routing:



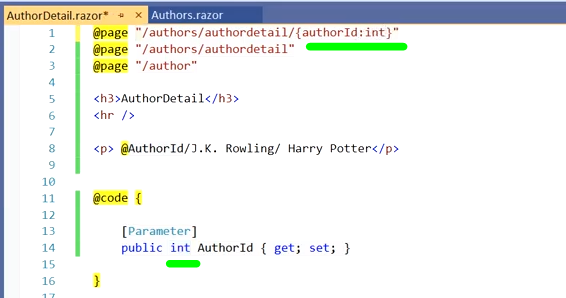
Routing parametrico:



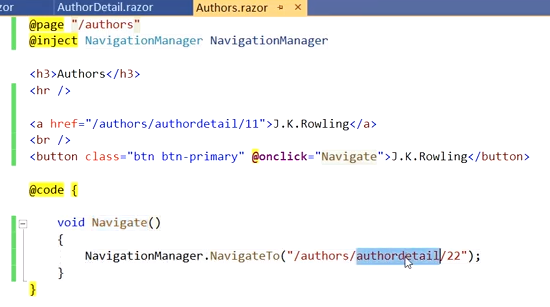
Per gestire il parametro facoltativo aggiungere il routing senza parametro:



Passare un parametro che non sia stringa:



Uso del navigationmanager per navigare:



Dependency injection



Transient crea una istanza ogni volta che viene chiamato

Se vogliamo che crei una sola volta creiamo un singleton

L’addscoped è simile al singleton con la differenza che se si fa refrsh della pagina viene istanziato un nuovo componente 8cosa che col singleton non avviene)



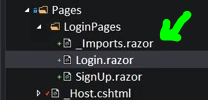
Iniettando il servizio non c’è bisogno di inizializzarlo:

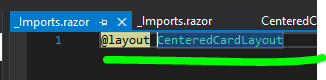


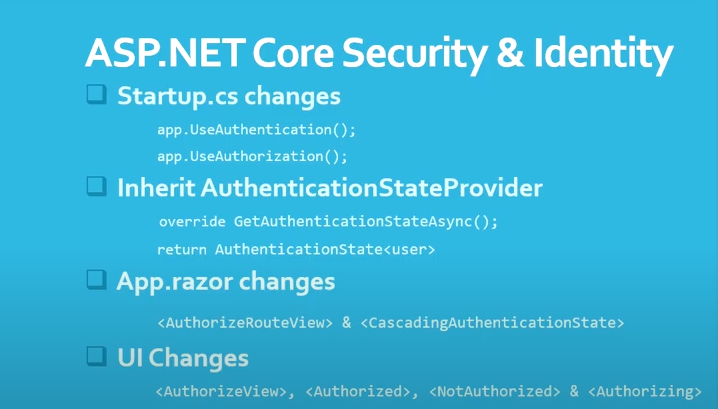
**Autentication**

Creare layout login page:

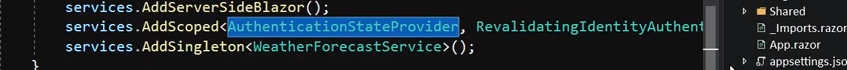
per dire che ogni componente della cartella utilizzerà uno specifico componente basta inserire:



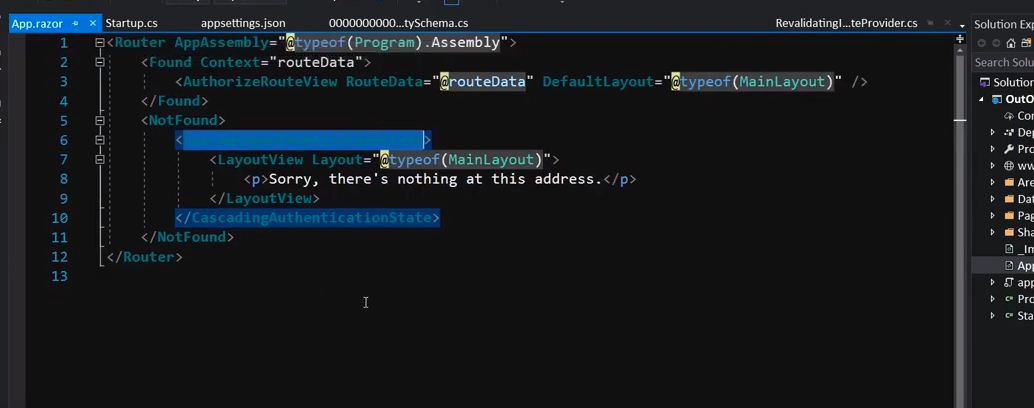




Se creiamo un progetto con autenticazione individuale in bazor server vengono inserite le pagine razor.



Quando avviene l’autenticazione viene chiamata la seconda classe qui su, mentre la prima viene usata per controllare se l’utente è validato o no.

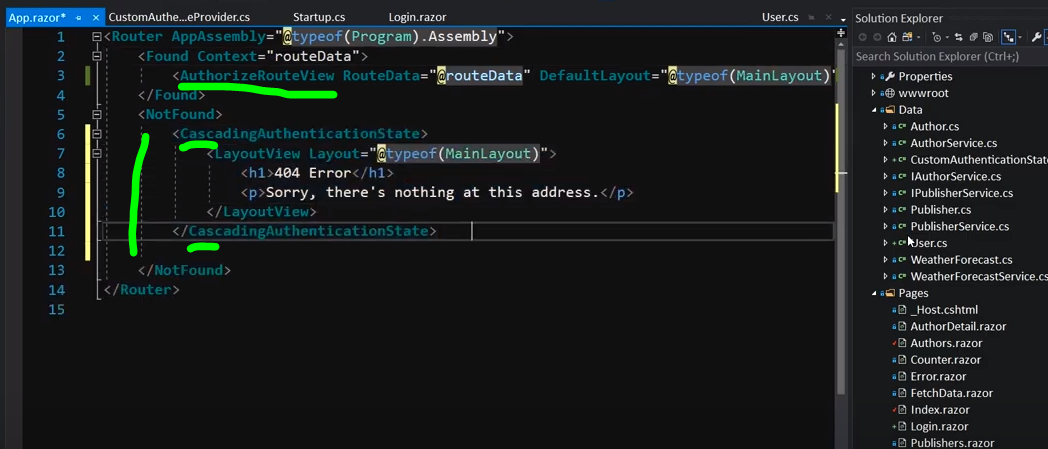


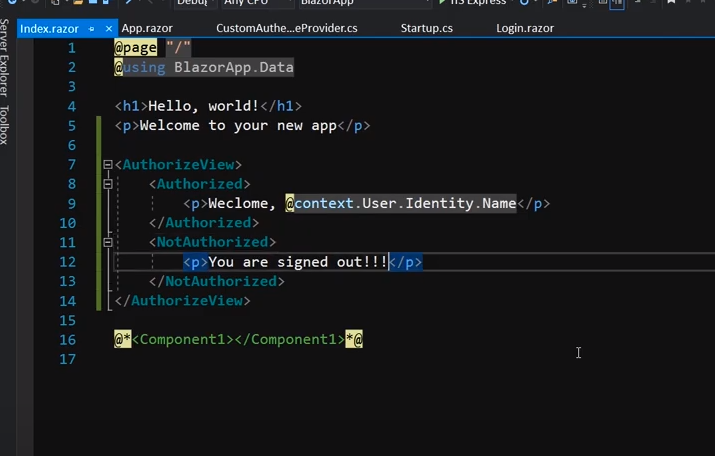
Prima creiamo il layout per la login page

Ora creiamo un progetto senza autenticazione ed aggiungiamola successivamente…

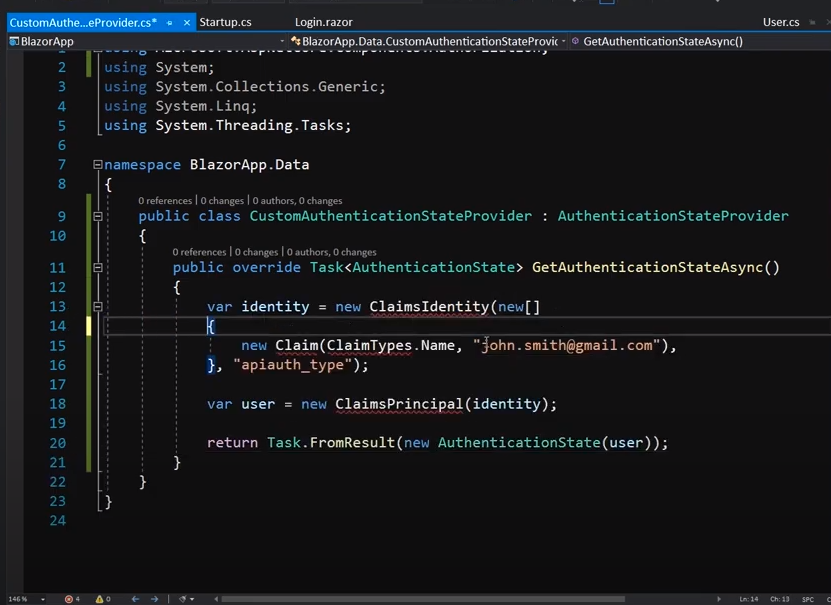


Bisogna ora dire che ci si aspetta come parametro a cascata il CustomAuthenticationStateProvider

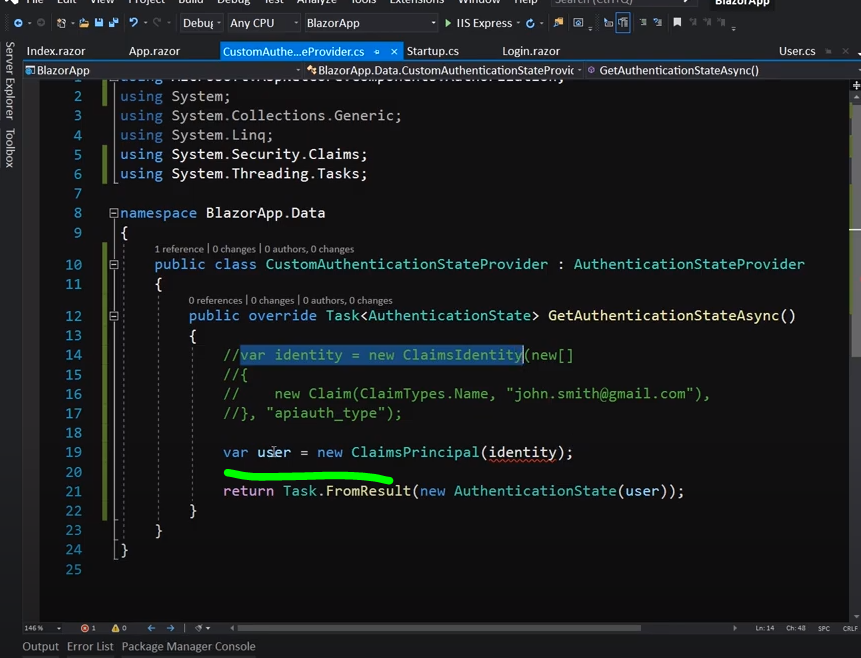


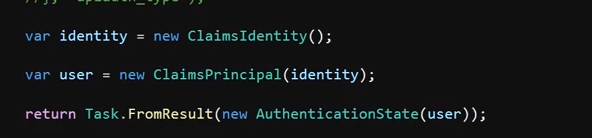


Qui l’utente verrà validato immediatamente al lancio dell’applicazione come john



In queto modo l’utente non verrà validato:

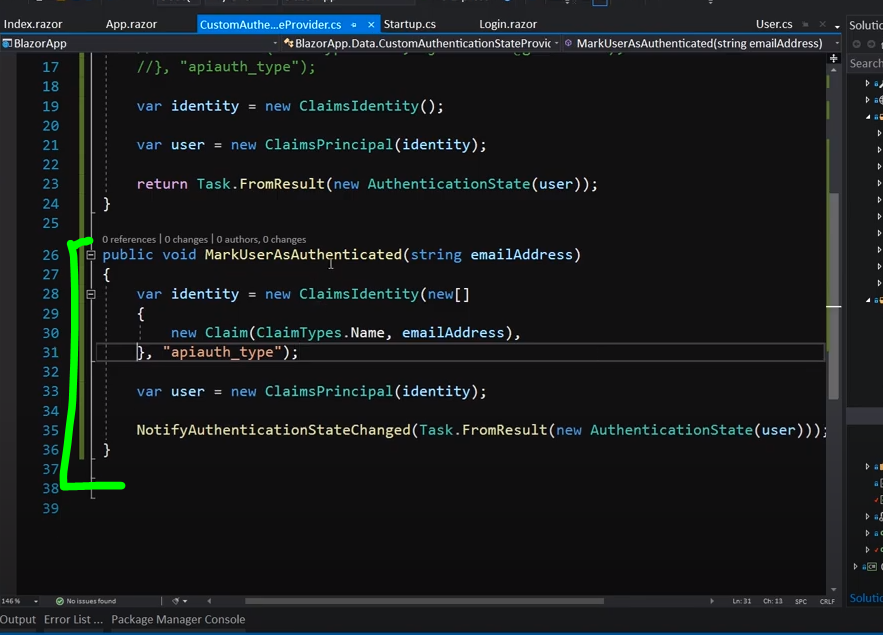


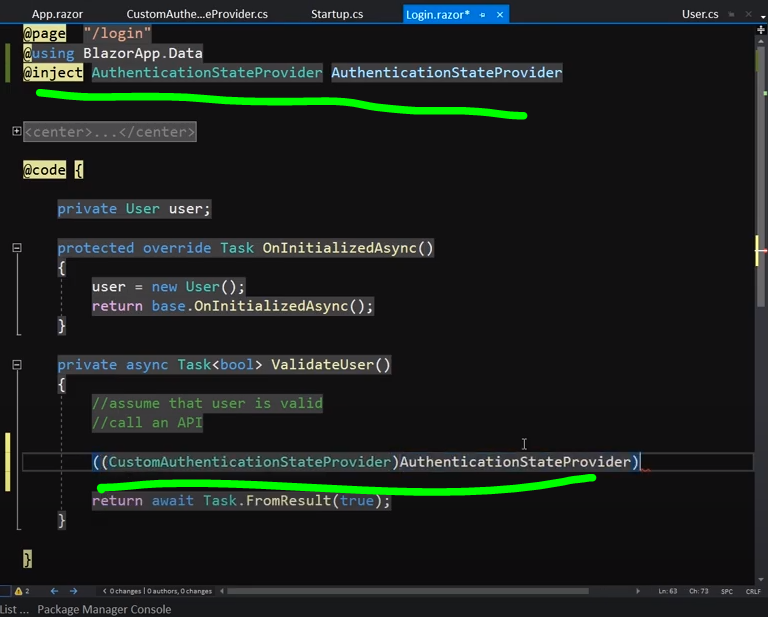


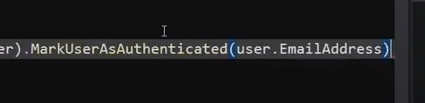
Occorre ora andare nella login, iniettare lo state provider e cambiare lo stato del provider quando è loggato.

Come facciamo?

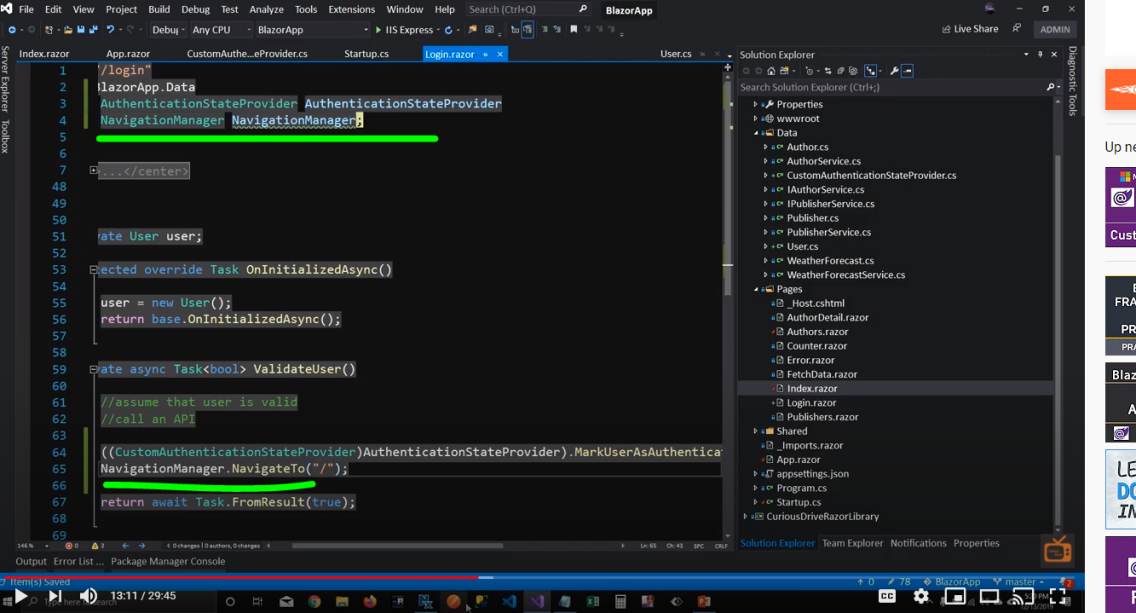
Creiamo nel customstateprovider:





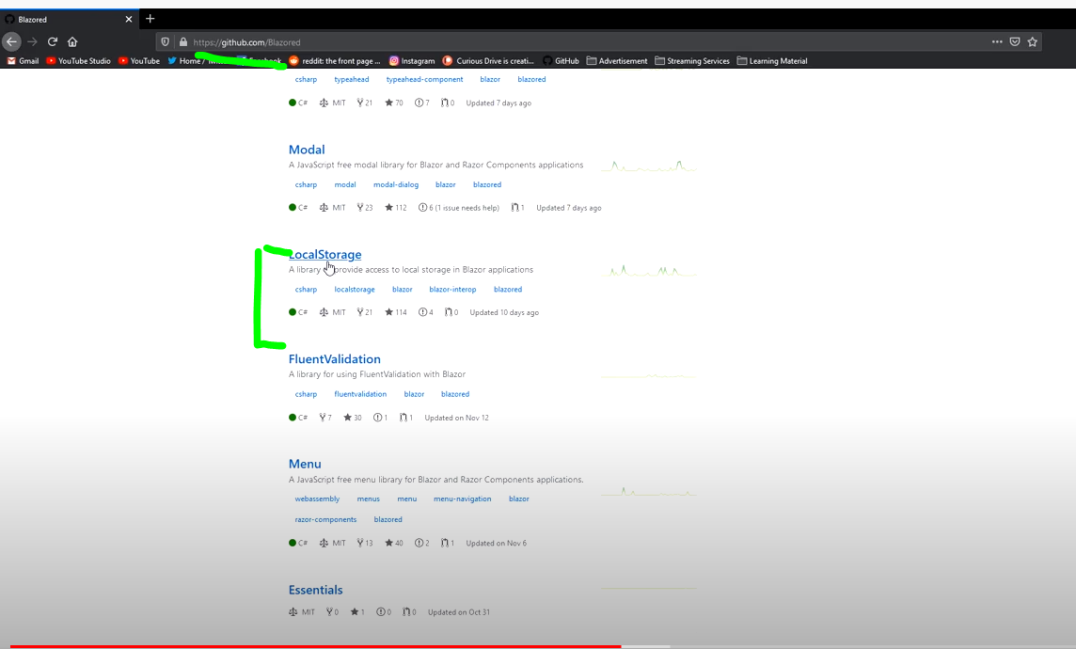


Dopodiché navighiamo alla index page:

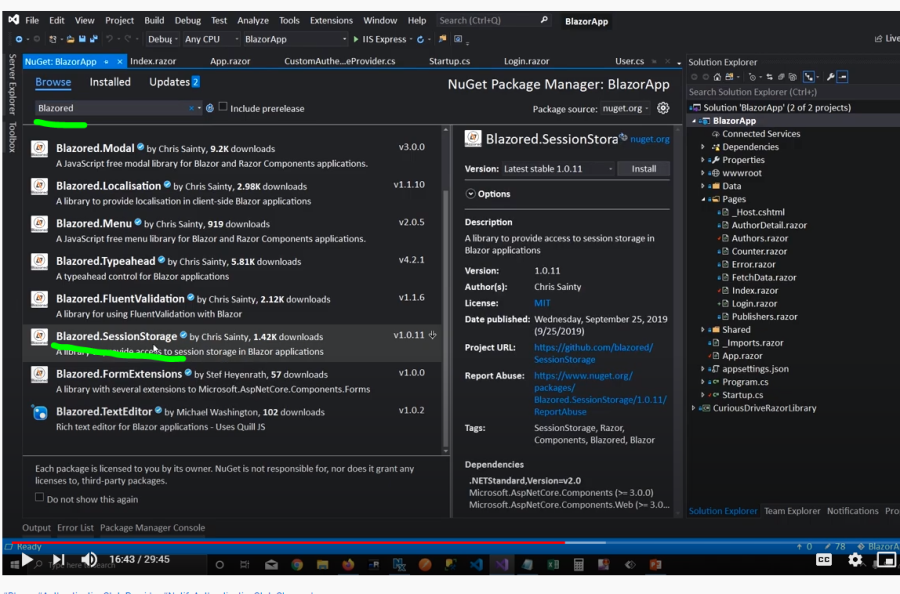


Il problema è che dopo autenticazione se si fa refresh non si è più loggati, questo perché dobbiamo mantenere lo stato dell’utente loggato tra markeduserasautenticated e getautenticationstate.

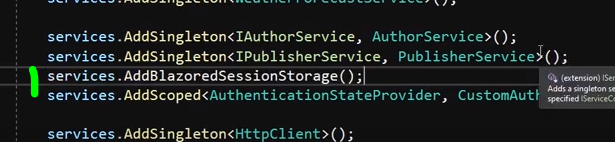
Per farlo usiamo localstorage:

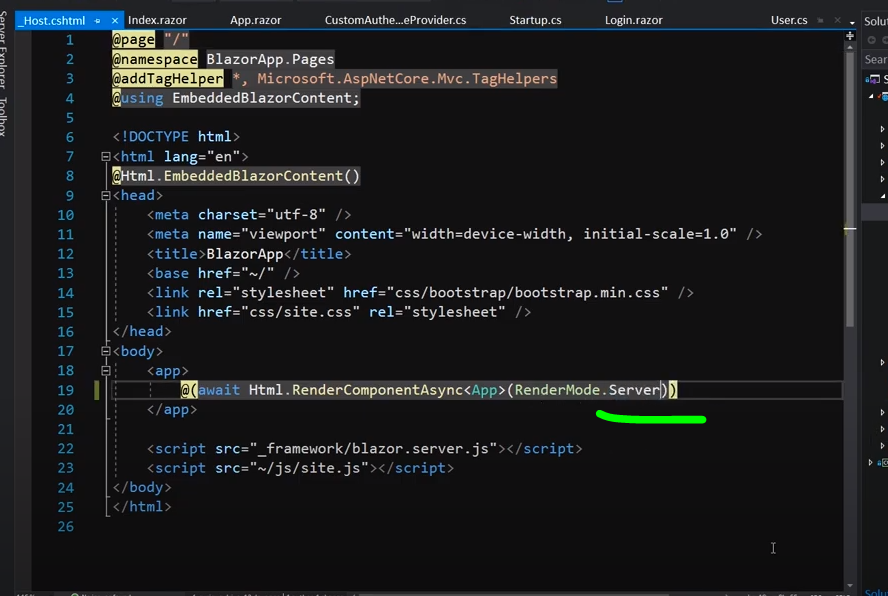


Esiste anche il sessionstorage



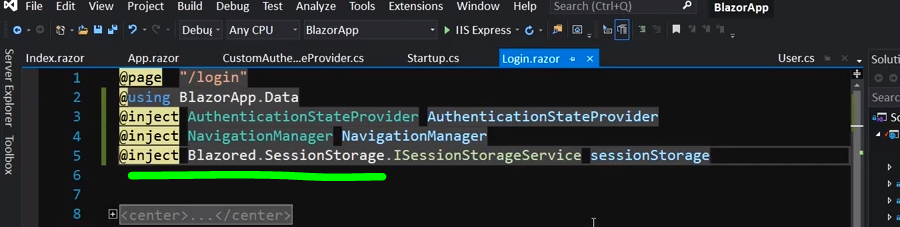
Aggiungo alla startup page:



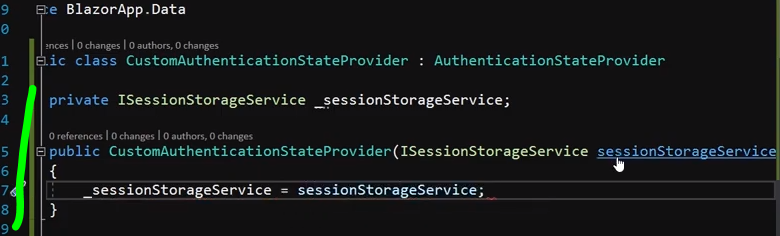


Questo perché sessionstorage usa javascript code è la pagina deve essere renderizzata completamente.

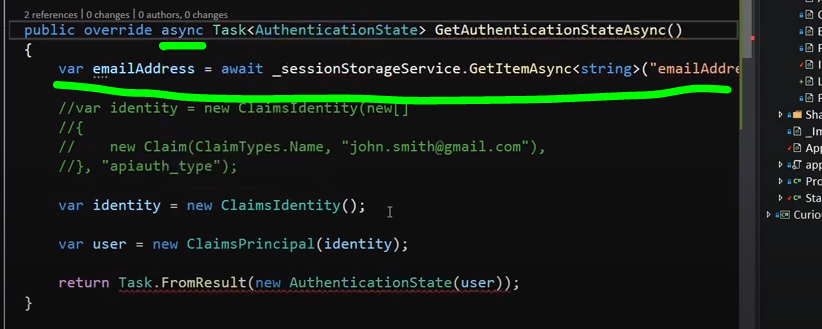
Inject sessionserver

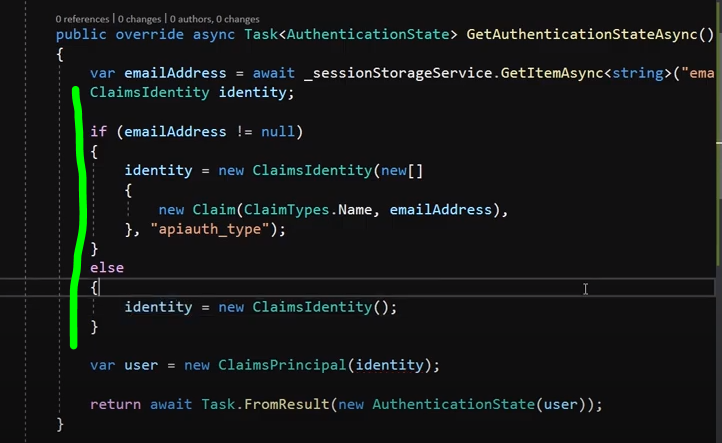


Ora creeremo una chiave email che verrà memorizzata al refresh.

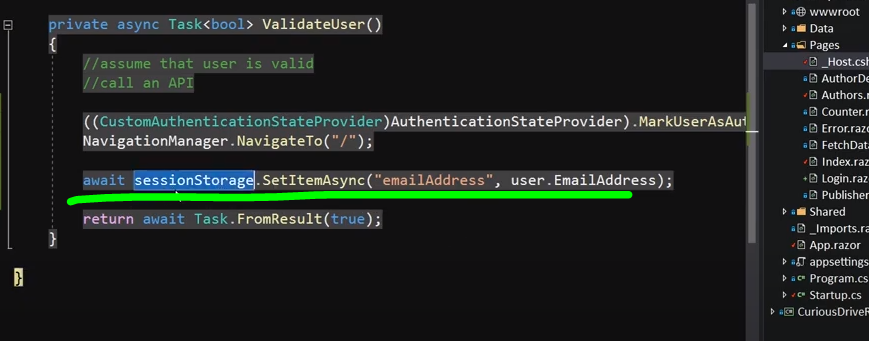


Recuperiamo la variabile email dello storage





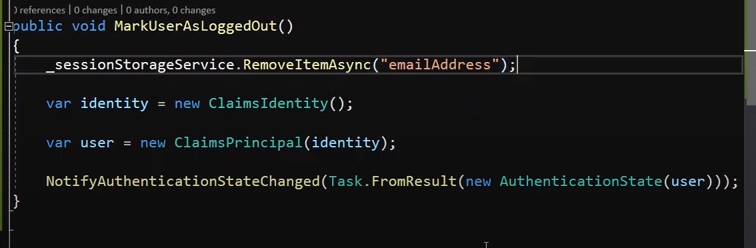
Settiamo la variabile dello storage



Ora vogliamo mostrare logout quando l’utente si è autenticato:



Quindi creiamo la routine per buttare l’utente fuori pulendo l’identity e rimuovendo dal localstorage la variabile:



Quindi andiamo a chiamare questa routine:

