1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Dipartimento di Scienze Pure e Applicate
Corso di Laurea in Informatica Applicata

Tesi di Laurea

COS'È BLAZOR?

Relatore:
Chiar.mo Prof. Emanuele Lattanzi

Candidato:
Francesco Belacca

Anno Accademico 2018-2019

A tutti quelli che mi hanno detto almeno una volta di non mollare.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto

A seguito della crescita esponenziale del web in questo secolo e dell'abituarsi di tutti coloro che ne usufruiscono ad un livello grafico sempre migliore e ad una esperienza mano a mano più interattiva e vicina all'utente medio, i siti web e le tecnologie utilizzate si sono adattati per permettere uno sviluppo sempre più rapido di codice più facilmente testabile e mantenibile.

Di conseguenza nel frontend si sono susseguiti una serie di framework e di strumenti, a partire da JQuery[2] nel 2006, che per primo si è occupato di risolvere il problema della compatibilità tra browsers, permettendo ai developers di scrivere una volta, e poter eseguire su tutti i browsers.

AngularJS nel 2010 è stato il primo MVC framework ad offrire in un unico pacchetto un insieme di features che hanno facilitato tantissimo la vita ai developers, come il two-way data binding, la dependency injection, il routin, oltre ad altri strumenti utili per rendere più standard lo sviluppo nel frontend [1]. Questo framework largamente utilizzato, é stato riscritto nel 2013 diventando Angular 2 (e nelle versioni piú recenti rinominato semplicemente in Angular) senza mantenere retrocompatibilità e senza offrire un modo preciso per migrare alla nuova versione agli utilizzatori di AngularJS. Anche per questo React, un nuovo framework più leggero e modulare sviluppato dagli sviluppatori di Facebook, ha preso il posto di Angular come framework più utilizzato nel frontend.

Vue infine é il terzo dei principali framework che ha provato a prendere piede proponendo una versione intermedia tra il fortemente opinionato Angular e il piú flessibile React.

Oltre a questi, ciascuno con la propria semantica, organizzazione logica dei folder, spesso una CLI dedicata, ad un developer frontend viene solitamente richiesto di conoscere HTML, CSS e chiaramente Javascript essendo ció su cui si basano poi i vari framework.

Oltre a Javascript, se si vuole scrivere degli unit test facilmente mantenibili, bisogna conoscere TypeScript(specialmente se si utilizza Angular, che rende il suo utilizzo obbligatorio) e degli altri framework che facilitino i test(Enzyme, Karma + Jasmine, ...).

1.2 Problema

I continui cambiamenti nei molti framework utilizzati, la diversità degli strumenti stessi tra loro, che spesso realizzano in modo diverso tutti la stessa cosa, rendono sempre piú difficile per un junior developer iniziare a sviluppare, vista l'ampia curva di apprendimento e lo studio necessario, per poter essere spendibile a livello professionale.

Oltretutto un web developer ad oggi finisce per essere costretto a scegliere se diventare uno sviluppatore frontend o backend, dato che rimanere al passo e aggiornarsi già in solo uno di questi due campi richiede tempo e non è scontato ad esempio che venga concesso di poterlo fare in orario lavorativo, pur essendo fondamentale.

É quindi chiaro che per tutti i web developer, e specialmente per una figura mista spesso identificata con il titolo "Full-Stack Developer", ci sia una continua ricerca del modo per rendere le proprie competenze quanto piú trasversali possibile, anche in termini di tecnologia utilizzata.

1.3 Linguaggi General-Purpose

Microsoft, nel backend e in ambito applicativo, ha reso nel tempo il framework .NET e le sue implementazioni(.NET Core, .NET Framework e Mono) utilizzabili nei vari propri linguaggi, C#, F# e VB. La figura 1.1 qui di seguito riassume le implementazioni del .NET Standard e le varie tipologie di applicazioni che si possono sviluppare con ciascuna.

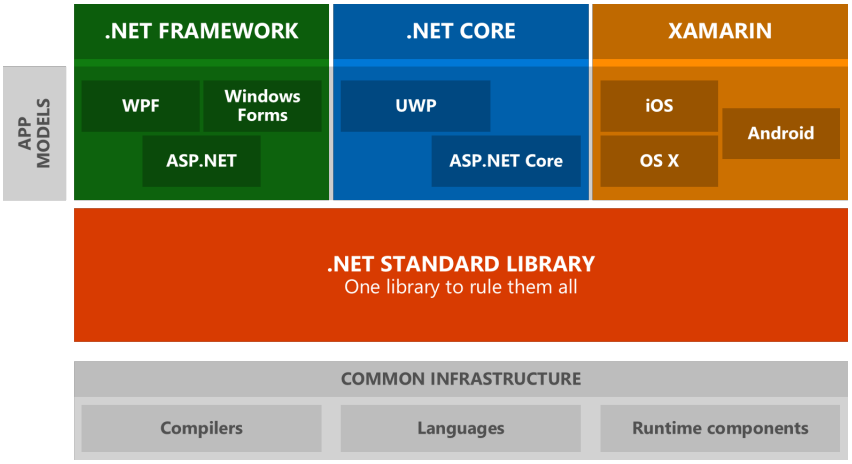

Figura 1.1: Implementazioni del .NET Standard

Scrivendo ad esempio in C# è possibile sviluppare vari tipi di applicazioni come si puó vedere nella figura 1.2, ma se si decide di sviluppare codice per un applicazione client web ad oggi si è ancora costretti a scrivere utilizzando Javascript e un suo framework se si vuole essere veloci nello sviluppo e scrivere codice mantenibile specialmente in team piú grandi, come nel mondo enterprise.

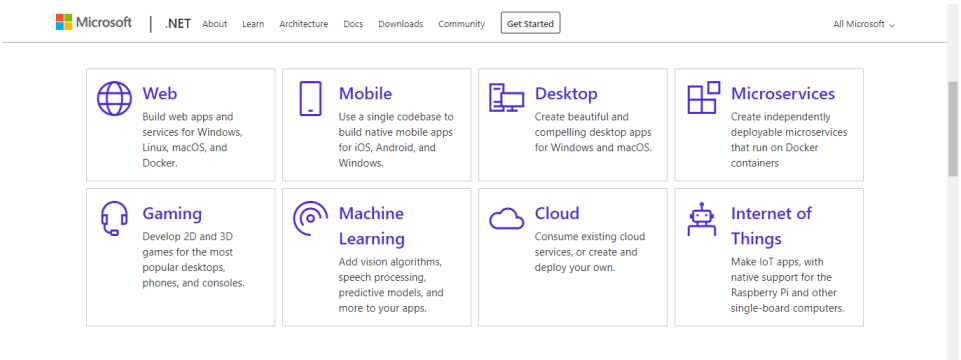

Figura 1.2: Possibilitá di .NET

Giá con Razor[3], Microsoft ha permesso la generazione di codice HTML e CSS in modo dinamico utilizzando C#, ma è utilizzabile solo lato server e quindi la cattura di un evento client side come il click di un utente su un bottone, senza contattare il server, non è stato possibile fino ad ora senza passare per Javascript.

1.4 Blazor

Ecco cosa é quindi Blazor: la versione piú avanzata di Razor(Browser+Razor[**?**]) che permette ai developer di gestire anche gli eventi client-side, direttamente in C#(o nel linguaggio scelto tra quelli supportati) come se questo fosse effettivamente ció che viene eseguito lato client, mentre in reatá l'esecuzione lato client cambia a seconda del modello scelto, come poi vedremo piú nel dettaglio.

Blazor utilizza per i vari modelli, delle tecnologie diverse(ad esempio il WebAssembly) ma che rispettano sempre lo standard del web. Non é invece un plugin da installare, e non dipende da un browser specifico. Non é nemmeno una "semplice" tecnologia di transpilazione verso javascript, contrariamente a tecnologie come TypeScript.

Capitolo 2

Modelli e Funzionamento

2.1 Blazor Server

Il primo dei modelli ufficialmente rilasciati e per il quale si puó ricevere supporto in produzione da settembre 2019[4], é proprio questo.

Un'applicazione Blazor Server ospita i componenti Blazor lato Server e gestisce le interazioni dell'utente con la UI attraverso una connessione in tempo reale sfruttando SignalR, come visibile nella figura 2.1.

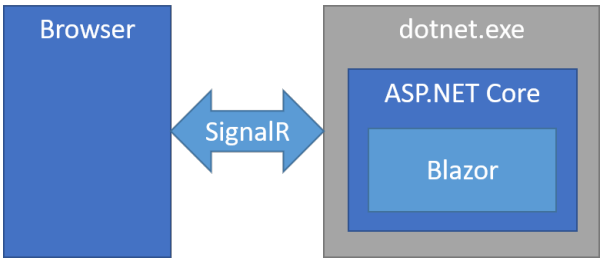

Figura 2.1: Blazor Server

Ció significa che quando un utente scatena un evento, questo viene inviato attraverso la real time connection al server, dove il rispettivo componente di competenza gestisce l'evento. Quando l'evento é stato gestito, blazor compara l'output appena generato con quello precedente all'evento, e manda quindi le sole differenze al browser del client, per poi applicarle al DOM.[5]

Blazor Server perció necessita di una connessione stabile e a bassa latenza per funzionare al meglio, e gli scenari offline non sono supportati. Ció significa anche che la posizione del server sul quale é ospitata l'applicazione non puó essere troppo distante dal client che si sta connettendo per garantire un funzionamento senza lag.

É particolarmente indicato quando si vuole delegare il costo computazionale al server e non ai client connessi, dato che ció che il client esegue é il solo codice statico e le differenze di volta in volta inviate ma calcolate lato server.

Ció rende molto veloce ed efficiente il download e l'avvio dell'applicazione lato client, il che lo rende il modello perfetto per funzionare su apparecchi a basso costo.

2.1.1 BlazorPong

Un'esempio di applicazione scritta con questo modello, é BlazorPong, da me implementata e la cui demo é disponibile sul relativo repository in GitHub[9].

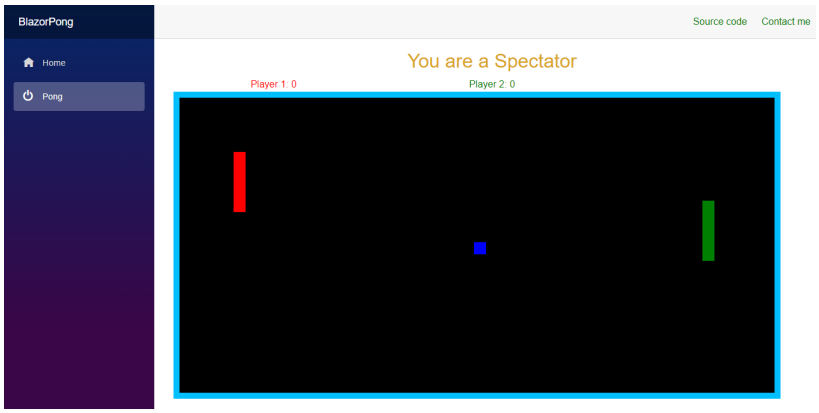

Figura 2.2: BlazorPong

Visibile nella figura questa applicazione permette a due giocatori che si collegano al sito contemporaneamente di giocare, e ai successivi utenti che si collegano di visionare la partita in corso come spettatori in tempo reale. In questa applicazione ho scelto di utilizzare il modello server side per vari motivi:

1. Il service worker che si occupa di aggiornare la posizione della pallina su tutti i client connessi, deve essere eseguito in un unico thread utilizzato per tutti i client connessi al server, ad ognuno dei quali invece devono arrivare gli aggiornamenti.

2. Il calcolo delle differenze da applicare a ciascun DOM di ciascun utente connesso, per quanto avvenga continuamente(60fps circa), conviene avvenga direttamente lato server poiché l'applicazione é molto semplice e infatti anche un host di livello gratuito come quello che ho utilizzato riesce a far giocare senza problemi due persone con diversi spettatori(i test che ho fatto sono tutti dall'europa, poiché il server utilizzato si trova in Francia).

3. Essendo la gestione degli eventi server side, il gioco puó essere visualizzato in modalitá spettatore senza problemi anche su cellulari non molto performanti.

Il funzionamento é piuttosto semplice: una volta che l'applicazione é stata avviata e che ciascun player si é connesso al sito ed ha cliccato play, quindi quando entrambi gli utenti sono pronti a giocare, inizia la partita. Lato server viene gestito lo spostamento costante della pallina ed eventuali collisioni con muri verticali, orizzontali o con il blocco di uno dei player. Rispettivamente gli eventi gestiti direttamente dal background worker sono quindi:

1. Punto per il player dal lato opposto in cui avviene la collisione con un muro verticale;
2. Inversione della velocitá di spostamento della pallina sull'asse y a seguito di una collisione con un muro orizzontale;
3. Inversione della velocitá di spostamento della pallina sugli assi x ed y a seguito di una collisione con uno dei player;
4. Invio di messaggio di fine partita contenente il player vincitore a tutti gli utenti connessi quando un utente arriva a 3 punti o uno dei due player si disconnette.

Ció che invece avviene grazie all'utente, é lo spostamento del proprio player attraverso l'evento di drag del proprio blocco. Ad ogni evento scatenato dall'utente, la connessione SignalR invia l'evento al server, che lo processa e restituisce a tutti gli utenti connessi(compreso quello che ha scatenato l'evento) la differenza di CSS necessaria a far combaciare la nuova posizione reale del blocco del player che si é mosso, con l'immagine visualizzata.

2.2 Blazor WebAssembly

Blazor WebAssembly é un modello attualmente in preview, che verrá ufficialmente rilasciato nella prima parte del 2020.

In questo modello il codice della SPA viene eseguito completamente lato client come solitamente avviene quando si utilizza un framework moderno per UI basato su JS, come i giá citati Angular,React,Vue.

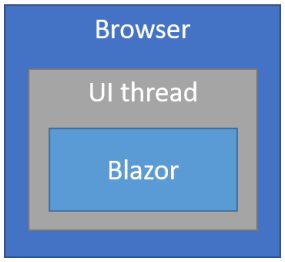

Figura 2.3: Blazor WebAssembly

Vengono quindi scaricati dal client l'applicazione Blazor, le sue dipendenze, ed il runtime del .NET scelto come target per l'applicazione. L'applicazione viene quindi eseguita direttamente nel thread della User Interface del Browser utilizzato, come visibile nella figura 2.2.

L'ambito di esecuzione é la stessa sandbox di qualsiasi altra applicazione scritta con javascript, ossia il browser che si sta utilizzando. Ció é importante perché implica(ed é cosí) che una web app scritta utilizzando Blazor non puó fare niente di piú o di meno di una web app standard. Ogni update alla UI e la relativa gestione, avvengono utilizzando lo stesso processo nel browser. Per questo modello, blazor.WebAssembly.js é il nome dello script Javascript che si occupa di scaricare il .NET runtime, l'applicazione e le dipendenze, come anche dell'inizializzazione dell'applicazione.

2.2.1 WebAssembly

In particolare il nome Blazor WebAssembly é stato scelto perché utilizzlato client il codice viene eseguito grazie al file mono.wasm, ossia la compilazione del Runtime del framework Mono in WebAssembly, e Mono é una delle implementazioni esistenti del framework di base .NET Standard. Il WebAssembly, che é un open standard che definisce un formato portatile di codice binario per programmi eseguibili e il rispettivo linguaggio assembly, come anche delle interfacce per facilitare le interazioni del codice con il proprio host. É anche detto il byte code del web, dato che dal 2017 i browser piú diffusi al mondo si sono impegnati per svilupparlo ed adottarlo.[10]

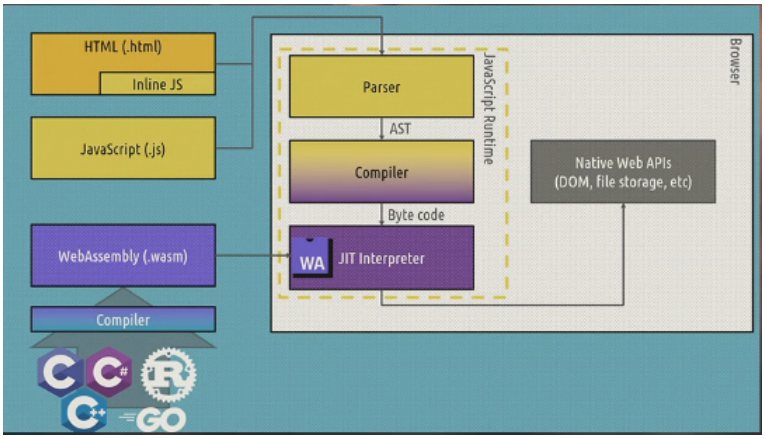

Figura 2.4: Confronto tra WebAssembly e Javascript

Come si puó vedere nella figura 2.3, uno dei motivi per cui WASM é piú veloce di Javascript é che non deve essere processato e compilato prima di poter essere interpretato, ma solamente scompattato ed eseguito.

La compilazione avviene in modalita AOT(Ahead of Time) e non piú JIT(Just in Time), il che si traduce in un sensibile miglioramenteo delle prestazioni del codice dinamico. Il C# quindi non é l'unico linguaggio che potrá essere utilizzato come partenza per essere compilato in WASM, tuttavia questa tecnologia é ció che rende possibile il funzionamento di Blazor.

In questo modello quindi, come visibile dai Chrome Developer Tools, vengono scaricate le DLL dell'applicazione direttamente nel browser dell'utente.

2.2.2 Blazor PWA

Lo step successivo per avvicinarsi al client allontanandosi dal modello server é poi Blazor PWA.

É cosí chiamato perché in questo modello Blazor, permette di sviluppare l'interfaccia utente di una Progressive Web App. Nella figura 2.3 viene riassunto cosa sia una Progressive Web App.

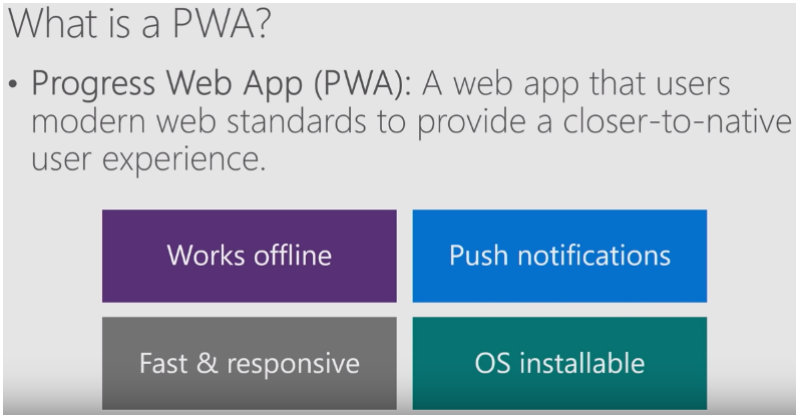

Figura 2.5: Progressive Web Apps

In particolare queste sono applicazioni web che hanno la capacitá di funzionare anche offline, e che spesso possono essere scaricate in modo persistente sulla macchina dell'utente che le esegue. Offrono chiaramente una maggiore velocitá di esecuzione e la possibilitá di sfruttare alcune API native. Ad esempio possono essere utilizzate quando la necessitá é quella di utilizzare le notifiche push native del SO che sta utilizzando il client(e.g. Windows).

Per realizzarne una in Blazor al momento, bisogna partire dal modello Blazor WebAssembly aggiungendo un manifesto che descriva le capacitá dell'applicazione, i permessi richiesti e l'icona da utilizzare una volta installata, oltre chiaramente a dover implementare l'applicazione in modo che possa lavorare anche offline, basandosi su un service worker.[6]

2.3 Modelli in roadmap

2.3.1 Blazor Hybrid

Questo modello di Blazor ed anche il successivo, servono a sviluppare applicazioni native. Nel modello Hybrid, l'applicazione sviluppata non é quindi piú considerabile una web app ma rimane ibrida perché pur essendo un app nativa, utilizza tecnologie web per effettuare il rendering della user interface.

Esempi di Hybrid Apps possono essere applicazioni mobile native, che hanno accesso alle API esposte ad esempio da Android, ma che utilizzano delle WebViews per la gestione dell'interazione dell'utente con la UI.

Un'altro esempio molto interessante sono le applicazioni che sfruttano Electron, che permette di compilare un'applicazione web in un applicazione desktop cross platform. Infatti utilizzando Electron si puó creare un'applicazione nativa, con l'interfaccia utente scritta utilizzando blazor, facendo in modo che in fase di esecuzione il processo host sia .NET Core(avendo quindi pieno accesso alle capacitá native e ad esempio al file system) pur rimanendo cross platform e potendo quindi eseguire su Windows, Linux e Mac. Di seguito nell'immagine 2.4 si puó vedere una Web Application compilata nativamente con Electron(con target Windows), e quindi eseguita come applicazione desktop:

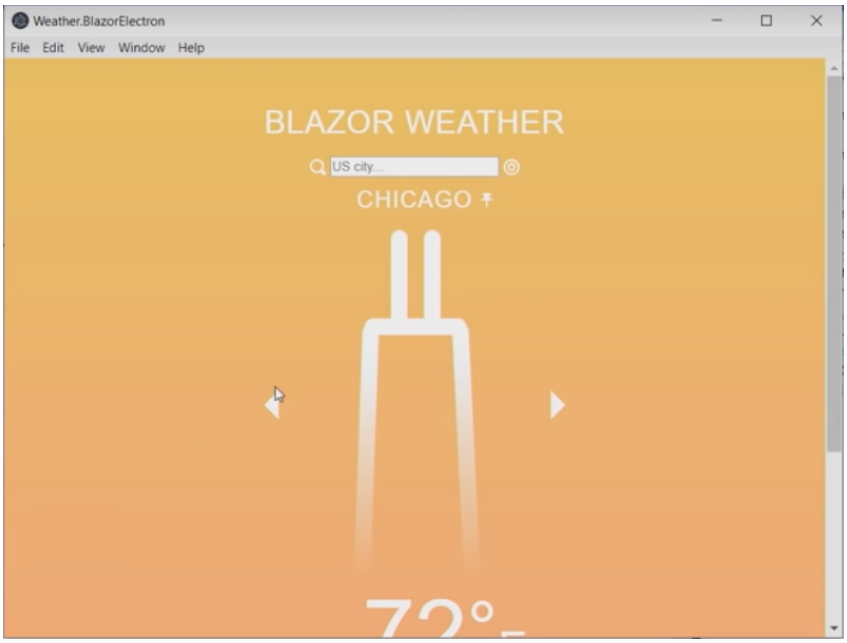

Figura 2.6: Blazor Hybrid Application

Il codice open source dell'applicazione qui sopra, si trovare al seguente link: https://github.com/danroth27/BlazorWeather/tree/master/BlazorWeather.Electron

2.3.2 Blazor Native

Infine esiste il modello Native che é possibile grazie al fatto che Blazor é stato architettato per poter renderizzare controlli della User Interface che non siano obbligatoriamente strumenti web, e puó quindi integrarsi con controlli nativi. Il rendering layer é infatti intercambiabile, pur essendo quello di default dedicato all'HTML.

Un esempio di applicazione sviluppata utilizzando Blazor per il rendering di controlli nativi nella user interface, si puó vedere durante la presentazione di Steve Sanderson all' evento NDC Oslo di quest'anno, pur non essendo stato ancora rilasciato il codice di un esempio ufficiale.[8] In questa applicazione, si é scelto di sostituire il default rendering layer per utilizzarne uno custom, utilizzando componenti di Flutter, il toolkit di Google per costruire interfacce utente native CrossPlatform. Questo modello viene qui citato per completezza, ma al momento non é nemmeno presente nella documentazione ufficiale ed é solo stato citato da Daniel Roth durante la presentazione dei futuri modelli di Blazor lato client.[7]

2.4 Funzionamento

Partendo dal codice, come giá anticipato quando si scrive si utilizza un mix di HTML per lo scheletro, CSS per lo stile del documento e C# preceduto dal relativo carattere di escape(@) per la parte dinamica del codice.

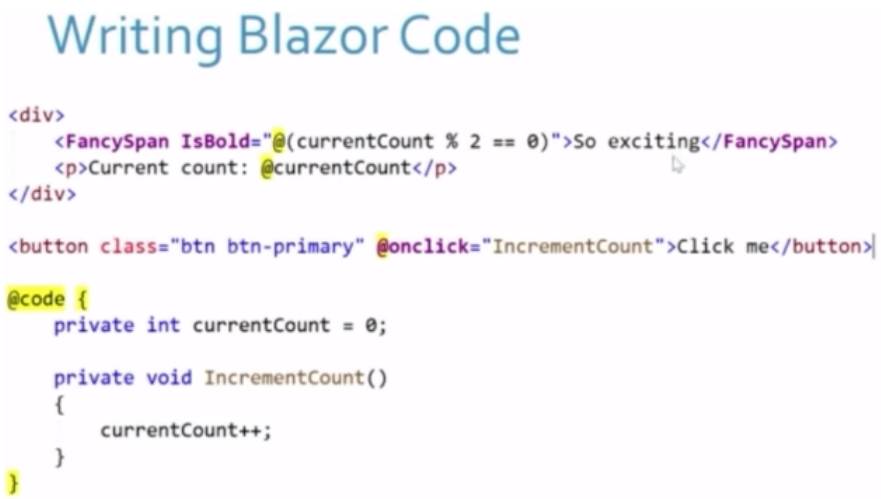

Figura 2.7: File Razor[11]

Si puó scrivere utilizzando questa sintassi solamente nei file che hanno estensione ".razor", il cui esempio si puó trovare in figura 2.7. Quando l'applicazione viene compilata, i file .razor vengono utilizzati come input per generare

dei file C#(quindi con estensione .cs) equivalenti al codice che si é descritto.[11] Un esempio di output di file razor compilato, generato a partire da quello in figura 2.7 é visibile in figura 2.8.

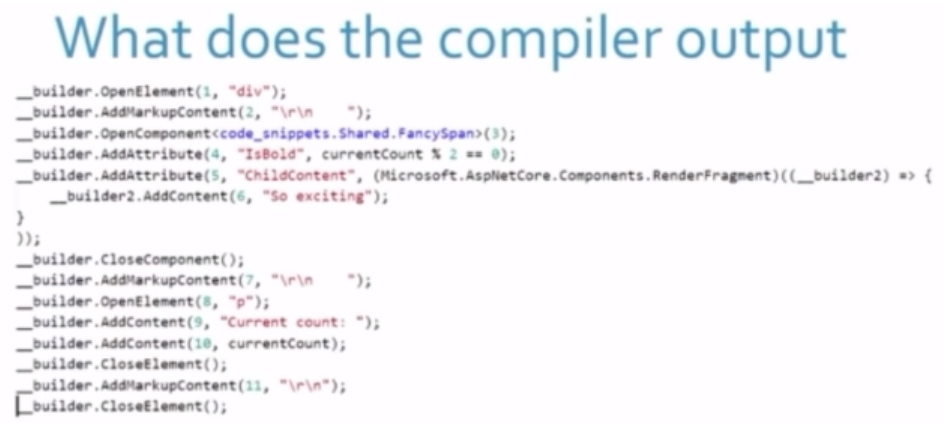

Figura 2.8: Output[11]

Saranno poi questi file ad essere effettivamente utilizzati da Roslyn(che é il nome del compilatore open-source del linguaggio C#) e a finire nella DLL generata come output dal progetto nel quale si trova il file .razor di partenza. Questo passaggio non é solo necessario per fare in modo che si possano generare le DLL compilate relative ai file di partenza, ma é anche il momento in cui viene ottimizzato ció che ha scritto il developer per ogni parte relativa alla UI in ogni file .razor del progetto, riducendolo alle sole primitive di Blazor, che sono vibili nella figura 2.9.

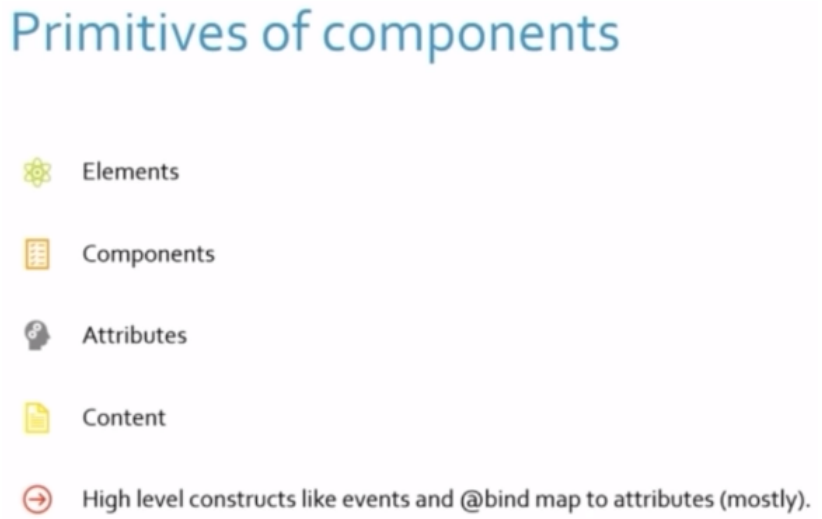

Figura 2.9: Primitive di Blazor[11]

In ordine quindi possiamo descrivere le primitive:

1. Un elemento é ad esempio l'elemento html "div";
2. Un component é un altro componente(file .razor) che viene utilizzato nel file che si sta compilando;
3. Un attributo é appunto un attributo di un elemento html, o un parametro passato in ingresso ad un component;
4. Un contenuto é del testo, costante o basato su un parametro, inserito all'interno di un elemento html;
5. L'ultima primitiva di un componente Blazor sono sostanzialmente gli eventi che vengono gestiti dal componente che si sta compilando e le direttive di bind che quindi collegano e mantengono sincronizzate due variabili.

Capitolo 3

Scalabilitá, Pro e Contro

3.1 Scalabilitá

Il problema della Scalabilitá, ossia della capacitá dell'applicazione di resistere all'aumentare del numero di utenti concorrenti che ne usufruiscono, dipende dal modello scelto, come anche dalla tipologia di applicazine sviluppata. Non é quindi risolvibile banalmente e non esiste una scelta universalmente giusta.

Il problema della scalabilitá comunque é fondamentalmente diverso tra Blazor Server, e tutti gli altri modelli. Questo principalmente per 3 motivi:

1. Il primo é che Blazor Server delega completamente al Server che ospita l'applicazione il carico computazionale necessario per gestire ogni singolo evento della UI di ogni utente connesso, compreso il salvataggio in-memory dello stato di ciascuna UI nel tempo durante l'utilizzo, visto che ad ogni Client connesso (o meglio allo specifico DOM) arrivano solamente le minime differenze necessarie rispettper generare la ui corretta rispetto al frame precedente, ogni volta che deve cambiare qualcosa. Ció implica che la potenza del server debba tener conto di eventuali picchi di utenze, e debba avere a disposizione sufficiente RAM per poter mantenere in memoria lo stato dell'applicazione di ciascun utente concorrente connesso.

2. Il secondo é che Blazor Server, necessita di una connessione sempre attiva con ogni utente collegato.

3. Il terzo é che questo modello sfrutta SignalR, che per funzionare al meglio utilizza il protocollo di trasporto WebSocket, quindi la macchina server sul quale viene ospitata l'applicazione Blazor é consigliato che lo supporti.

Gli altri modelli invece sfruttano l'hardware di ogni client connesso, come solitamente avviene per le SPA odierne. Sono quindi in linea con il comportamento degli altri framework giá citati e come Angular React o Vue.js,

in particolare Blazor WebAssembly, che quindi prenderemo in considerazione confrontandolo con Blazor Server.

3.2 Blazor Server

3.2.1 Pro

3.2.2 Contro

3.3 Blazor WebAssembly

3.3.1 Pro

3.3.2 Contro

Capitolo 4

Conclusioni

4.1 Conclusioni

4.2 Futuro del progetto

Bibliografia

[1] M. Wanyoike, Disponibile: https://blog.logrocket.com/history-of-frontend-frameworks.

[2] Disponibile: https://en.wikipedia.org/wiki/JQuery.

[3] R. Anderson, R. Nowak, Disponibile: https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-3.0&tabs=visual-studio

[4] D. Roth, Disponibile: https://devblogs.microsoft.com/aspnet/asp-net-core-and-blazor-updates-in-net-core-3-0.

[5] D. Roth, Disponibile: https://devblogs.microsoft.com/aspnet/blazor-server-in-net-core-3-0-scenarios-and-performance.

[6] D. Roth, Disponibile: https://youtu.be/qF6ixMjCzHA?t=485

[7] D. Roth, Disponibile: https://youtu.be/qF6ixMjCzHA?t=872

[8] S. Sanderson Disponibile: https://www.aka.ms/blutter

[9] Disponibile: https://github.com/MACEL94/BlazorPong

[10] Disponibile: https://webassembly.org/roadmap/

[11] S. Sanderson, R. Nowak
Disponibile: https://www.youtube.com/watch?v=dCgqTDki-VM

[12] Disponibile: https://github.com/aspnet/Blazor/wiki/FAQ#q-where-does-the-name-blazor-come-from

Ringraziamenti

Vorrei ringraziare il professor Lattanzi per avermi aiutato a scrivere questa tesi, ed i miei genitori per avermi dato la possibilitá di studiare ció che volevo e lavorare quando l'ho scelto senza condizionarmi.

Infine un ringraziamento speciale va alla mia ragazza Ivana per aver creduto in me e nelle mie capacitá anche quando non l'ho fatto io.