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

In seguito alla crescita esponenziale del web in questo secolo e con l'abituarsi di tutti coloro che ne usufruiscono ad un livello grafico sempre migliore e ad una esperienza mano a mano più interattiva e studiata per l'utente medio, le tecnologie utilizzate per costruire i siti web si sono adattate 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 in modo molto rapido, a partire da JQuery[2] nel 2006, che per primo si è occupato di risolvere il problema della compatibilità del codice scritto con i diversi browsers, permettendo ai developers di scrivere una volta, e poter eseguire su tutti i più diffusi.

Nel 2010 è stato creato AngularJS, il primo framework per applicazioni che seguono il pattern architetturale Model-View-Controller(MVC) ad offrire in un unico pacchetto un insieme di features che hanno facilitato molto la vita agli sviluppatori, come il two-way data binding, la dependency injection, il routing integrato, ed altri strumenti utili per rendere più standard lo sviluppo nel frontend [1]. AngularJS pur essendo stato largamente utilizzato, è stato cambiato radicalmente nel 2013 e rinominato in Angular2 (e nelle versioni più recenti semplicemente Angular) senza mantenere retrocompatibilità e senza offrire un modo preciso per migrare alla nuova versione ai precedenti utilizzatori di AngularJS. Anche per questo motivo, React, un altro framework più leggero e modulare mantenuto da Facebook e la cui prima versione risale al 2013, ha poco a poco preso il posto di Angular come framework più utilizzato dai developers nel frontend, visibile anche nell'ultimo questionario condotto dal sito StackOverflow nel quale è stato chiesto quali tecnologie vengano utilizzate a circa 90000 sviluppatori in tutto il mondo[3]. Infine Vue, lanciato nel 2014, è il terzo dei principali framework moderni per la creazione di interfacce utente ed è stato fortemente adottato, proponendo una versione intermedia tra il più opinionato Angular e il più flessibile React[4].

Oltre ad almeno uno dei framework citati, ciascuno con la propria semantica, organizzazione logica delle cartelle e spesso una CLI dedicata, un frontend developer odierno dovrebbe conoscere bene HTML, CSS e chiaramente Javascript, jQuery per retrocompatibilità ed almeno Bootstrap per creare componenti standard più in fretta.

In particolare se si utilizza Angular, come anche se si vuole scrivere codice più facilmente mantenibile e type-safe, il developer dovrà conoscere anche TypeScript. TypeScript è un linguaggio di programmazione che estende la sintassi di Javascript, rendendolo tipizzato(e non interpretato), e viene quindi compilato e trasformato in codice equivalente nel linguaggio javascript, per poter essere utilizzato dal browser. Questo processo di compilazione e traduzione da un linguaggio verso l'altro, prende il nome di transpilazione.

1.2 Problema

I continui aggiornamenti nei molti framework utilizzati e la diversità degli strumenti tra loro, che spesso realizzano in modo diverso la stessa cosa, rendono sempre più difficile per un junior developer iniziare a sviluppare, vista l'ampia curva di apprendimento e quindi il tempo e lo studio necessario, per poter essere pronto a lavorare come professionista.

Spesso i web developer infatti finiscono per scegliere se diventare uno sviluppatore frontend e concentrarsi su quello stack di tecnologie da conoscere ed approfondire, o se diventare un developer backend, dato che rimanere al passo anche solo uno di questi due campi richiede tempo, oltre all'impegno.

È quindi chiaro che da parte di tutti i web developer e specialmente per una figura esperta e mista identificata con il titolo "Full-Stack Developer", ci sia una continua ricerca del modo per rendere le proprie competenze quanto più trasversali possibile, anche in termini di tecnologie utilizzate.

1.3 Linguaggi General-Purpose

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

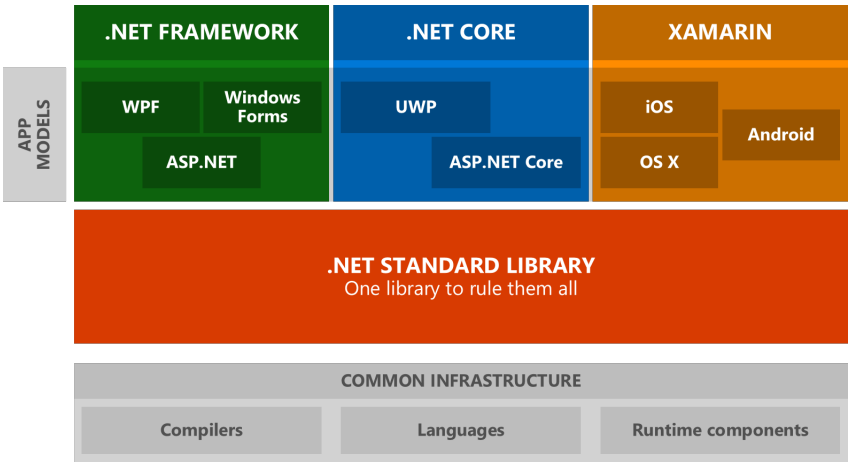

Figura 1.1: Implementazioni del .NET Standard

Se si decide di sviluppare codice per un'applicazione web che gestisca eventi del Browser con prestazioni native(click, drag, hover,...) e senza costringere l'utente a scaricare plugin, si è costretti a scriverla utilizzando Javascript(JS). Eventualmente sfruttando anche un framework basato su JS se si vuole essere più veloci nello sviluppo e scrivere codice mantenibile, esigenze fondamentali a livello enterprise.

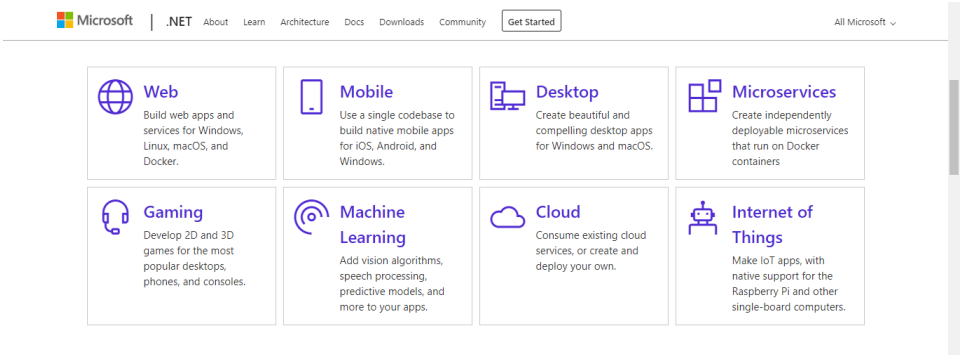

Figura 1.2: Possibilità di .NET

Già con il rilascio di Razor[5] nel 2011, Microsoft ha permesso la generazione di codice HTML e CSS in modo dinamico utilizzando C#, ma questa tecnologia è stata creata e pensata per il server-side rendering ed è quindi utilizzabile solo quando si scrive codice che sarà eseguito da un server, non da un browser. Ciò implica che la cattura di un evento client side come il click di un utente su un bottone, senza dover eseguire una chiamata al server sul quale viene eseguito l'hosting dell'applicazione e l'attesa della relativa una risposta, non risulti possibile senza l'utilizzo di Javascript.

1.4 Blazor

Ecco cosa è quindi Blazor: la versione più avanzata di Razor(Browser+Razor[6]), o meglio ancora un Framework per la creazione di User Interfaces(UI) di Single Page Application(SPA) che permette ai developer di gestire anche gli eventi client-side, direttamente in C# o nel linguaggio scelto tra quelli supportati. Per SPA si intende un applicazione web che interagisce con l'utente riscrivendo dinamicamente la pagina nel quale l'utente si trova a seconda delle sue azioni, piuttosto che ricaricare nuove pagine di volta in volta richiedendole al server ad ogni click[7]. Questo framework permette allo sviluppatore di scrivere codice come se il linguaggio scelto(non JS) fosse effettivamente ciò che viene eseguito dal client, mentre in realtà ciò che viene eseguito dal browser dell'utente cambia a seconda del modello scelto, come poi vedremo più nel dettaglio.

Blazor utilizza per i vari modelli, delle tecnologie diverse(ad esempio il WebAssembly) che rispettano però lo standard del web, in quanto non sfrutta plugin che ogni utente deve appositamente scaricare per poter utilizzare l'applicazione, e non dipende da un browser specifico.

Capitolo 2

Modelli e Funzionamento

2.1 Blazor Server

Il primo dei modelli ufficialmente rilasciati e per il quale si può già ricevere supporto in produzione, sin da settembre 2019[8], è Blazor Server.

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.

Per "componente" si intende un pezzo di UI autonomo, come una pagina, una finestra modale o un form da compilare. Un componente include quindi sia HTML che la logica necessaria per iniettarci dei dati o per rispondere ad eventi della UI. I componenti sono flessibili e leggeri. Possono essere annidati, riutilizzati, e condivisi tra progetti[9].

SignalR invece è una libreria di software open-source sviluppata da Microsoft, che facilita l'interazione tra Client e Server, specialmente quando si necessita di mandare messaggi dal server ad uno o più client con un'alta frequenza [10]. SignalR mette a disposizione delle API che permettono al codice del server di eseguire Remote Procedure Calls(RPC), in particolare è utilizzato per inviare messaggi real-time ad uno o più client connessi all'applicazione contemporaneamente(e.g. in una chat room). Questa libreria viene sfruttata dal modello Blazor Server per trasmettere al browser di ciascun utente connesso quali modifiche debbano essere applicate al Document Object Model(DOM) perchè la UI di ciascun Client rispetti la UI che il server mantiene in memoria. SignalR supporta le seguenti tre tecniche di gestione per comunicare in tempo reale, in ordine di fallback:

1. WebSockets
2. Server-Sent Events
3. Long Polling

SignalR sceglie automaticamente il miglior metodo di trasporto disponibile a seconda delle capacità di client e server che stabiliscono la connessione[11].

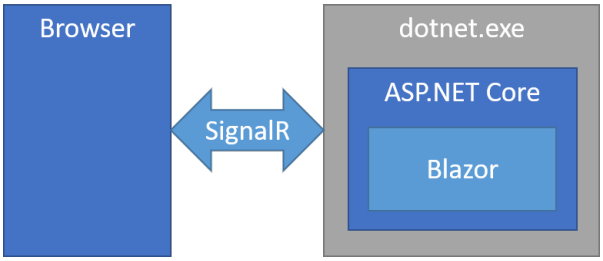

Figura 2.1: Blazor Server

Ciò significa che quando un utente scatena un evento, questo viene inviato attraverso la real time connection al server, dove il rispettivo componente di competenza lo gestisce. Quando l'evento è stato gestito, Blazor compara l'output appena generato con quello precedente all'evento, e se presenti manda le sole differenze al browser del client, per poi applicarle al DOM[12].

Questa caratteristica del modello Blazor Server è una fondamentale differenza rispetto a ASP.NET e Razor: le pagine rigenerate ad ogni evento lato server, in precedenza, venivano totalmente rispedite al client, operazione che non richiedeva RAM in modo persistente per ogni utente connesso al server, ma molto meno efficiente in termini di prestazioni. Blazor Server perciò necessita di una connessione stabile e a bassa latenza per funzionare al meglio, e gli scenari offline non sono supportati. Ciò significa anche che la posizione del server sul quale è ospitata l'applicazione non può essere troppo distante dal client che si sta connettendo per garantire un funzionamento senza lag.

Questo modello è particolarmente indicato quando si vuole delegare il costo computazionale al server e non ai client che ci si connettono, dato che ciò che il client deve computare è il solo codice statico scaricato inizialmente, e le differenze di volta in volta ricevute dal server, dove avviene invece la gestione di ogni evento e il calcolo delle differenze tra ciò che viene visualizzato prima di un evento e ciò che deve cambiare nella User-Interface del client dopo la sua gestione. Ciò rende molto veloce ed efficiente il download iniziale e l'avvio dell'applicazione per gli utenti, il che lo rende il modello più adatto se le macchine che devono utilizzare l'applicazione sono apparecchi a basso costo, ammesso che siano sempre essere connessi ad internet.

2.1.1 BlazorPong

Un esempio di applicazione scritta utilizzando questo modello, è BlazorPong, appositamente implementata per questo lavoro di tesi e la cui demo è disponibile nel relativo repository su GitHub[13].

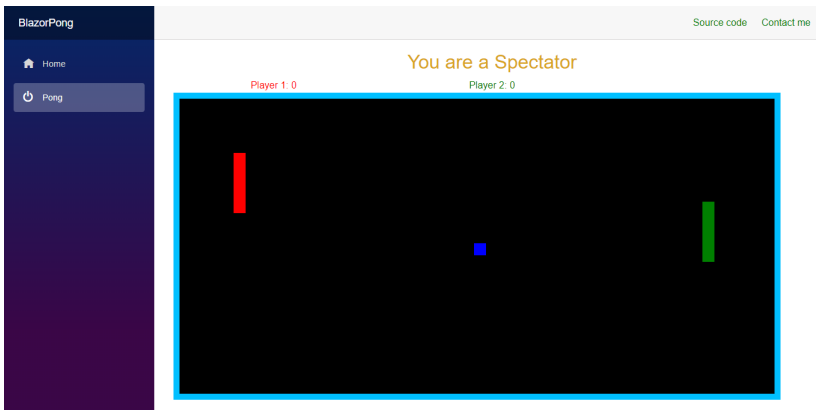

Figura 2.2: BlazorPong

Questa applicazione, visibile nella figura 2.2 permette a due giocatori che si collegano al sito contemporaneamente di giocare, e ai successivi utenti che si collegano di seguire la partita in corso come spettatori in tempo reale. In questa applicazione è stato utilizzato il modello Blazor Server side per vari motivi:

1. Il background worker del server 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.

2. Il calcolo delle differenze da applicare a ciascun DOM di ciascun utente connesso, per quanto avvenga molto spesso(è stato limitato per rendere 60 fps circa), non ha un peso tale da non poter essere eseguito lato server poiché l'applicazione è molto semplice e di conseguenza leggera. Una sola macchina host gratuita come quella che è stata utilizzata(su Azure), riesce a far giocare senza problemi due persone con diversi spettatori. È comunque bene specificare che tutti i test di performance che sono stati fatti sono stati eseguiti in Europa, e che il server utilizzato si trova in Francia.

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

Il funzionamento di BlazorPong è piuttosto semplice: quando almeno due utenti si collegano all'applicazione e ciascuno ha cliccato play, 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 dal background worker eseguito sul server sono i seguenti:

1. Collisione con un muro verticale con conseguente punto per il player che si trova dal lato opposto di quello in cui avviene la collisione;
2. Collisione con un muro orizzontale con conseguente inversione della velocità di spostamento della pallina sull'asse y;
3. Collisione con uno dei blocchi dei giocatori con conseguente inversione della velocità di spostamento della pallina sugli assi x ed y;
4. L'arrivo a 3 punti di uno dei due giocatori o la disconnessione di uno dei due, gestito con l'invio di un messaggio di fine partita contenente il player vincitore a tutti gli utenti connessi.

Ciò che invece avviene grazie all'utente, è lo spostamento del proprio player attraverso la cattura dell'evento di drag eseguito sul proprio blocco quando lo si trascina. Ad ogni evento scatenato dall'utente, la connessione SignalR invia l'evento al server, che lo processa e restituisce a tutti gli utenti connessi la nuova posizione. Viene escluso dall'aggiornamento di posizione del blocco inviato dal server, il solo player che ha scatenato l'evento, poiché l'invio delle differenze necessarie per renderizzare la UI aggiornata viene gestito già da Blazor Server.

2.2 Blazor WebAssembly

Blazor WebAssembly è un modello attualmente in anteprima, e sarà ufficialmente rilasciato verso maggio del 2020.

In questo modello il codice della SPA viene scaricato ed eseguito nel browser del client che si collega, come solitamente avviene quando si utilizza un framework moderno per UI come i già citati Angular, React e Vue.

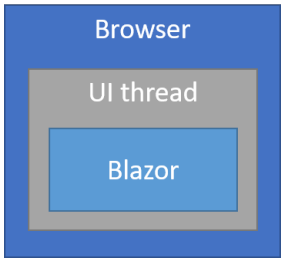

Figura 2.3: Blazor WebAssembly

Vengono quindi scaricati dal client l'applicazione Blazor, le sue dipendenze, ed il runtime .NET necessario per eseguire le DLL. L'applicazione viene quindi eseguita direttamente nel thread della UI del Browser utilizzato per collegarsi all'applicazione, come visibile nella figura 2.3.

L'ambito di esecuzione è la stessa sandbox di qualsiasi altra applicazione scritta con javascript, ossia il browser che si sta utilizzando. Ciò è molto importante perchè significa che potenzialmente un'applicazione web scritta utilizzando Blazor non può fare niente di più o di meno di un'applicazione web standard(come ad esempio accedere al file system). Ogni update alla UI e la relativa gestione, avvengono utilizzando lo stesso processo nel browser. Per questo modello, blazor.WebAssembly.js è il nome dello script Javascript che all'apertura della pagina index si occupa di scaricare il .NET runtime compilato in WebAssembly, l'applicazione e le sue dipendenze, come anche dell'inizializzazione dell'applicazione.

2.2.1 CLR e WebAssembly

In particolare il nome Blazor WebAssembly per questo modello è stato scelto perchè nel browser di ciascun client che utilizza un'applicazione di questo tipo, viene scaricato ed utilizzato il file "mono.wasm". Questo file contiene il CLR(Common Language Runtime) di Mono, una delle implementazioni del .NET Standard, compilato in WebAssembly. Il runtime è necessario per poter interpretare ed eseguire i file compilati dell'applicazione(le DLL) e i pacchetti sulla quale si basa, che hanno estensione .dll e come target uno dei framework che implementano il .NET Standard(nel caso di Blazor, .NET Core), come visibile nella figura 2.4.

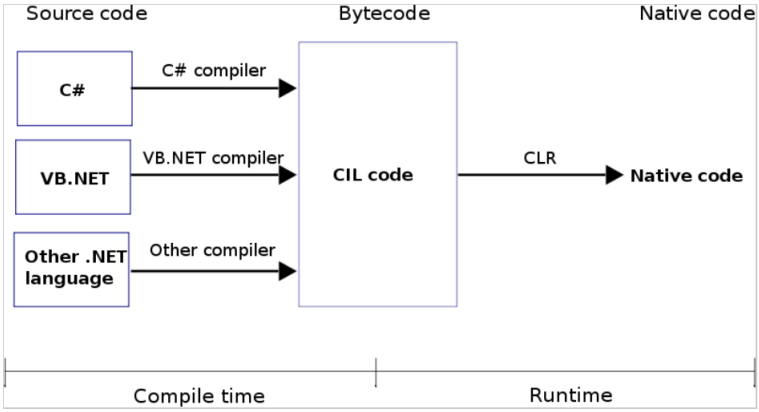

Figura 2.4: Utilizzo del CLR

Il WebAssembly(WASM) è un formato per istruzioni binarie creato per essere il target della compilazione di linguaggi ad alto livello come C, C++, Rust[14]. L'estensione .wasm è quella che è stata scelta per i file WASM. Il progetto per la creazione di WASM è nato nel 2015, mentre dal 2017 i browser più diffusi al mondo come Chrome, Firefox, Edge e Safari si sono impegnati per svilupparlo ed adottarlo[15].

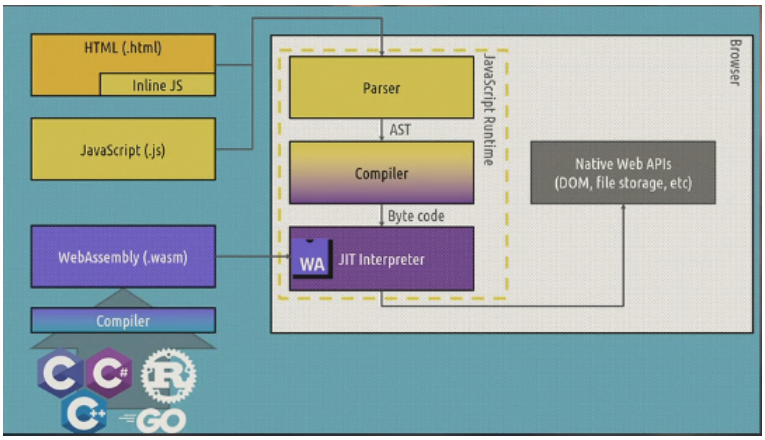

Figura 2.5: Confronto tra WebAssembly e Javascript

Come si può vedere nella figura 2.5, uno dei motivi per cui WASM è più veloce di Javascript è che non deve essere analizzato e compilato prima di poter essere interpretato, va solamente decompresso e per questo motivo spesso viene definito il byte code per il web.

La compilazione può avvenire in modalità AOT(Ahead of Time) e non solamente JIT(Just in Time) come per JS, il che si traduce in un sensibile miglioramento delle prestazioni del codice dinamico. Questa tecnologia è ciò che rende possibile il funzionamento di Blazor WebAssembly.

2.2.2 Blazor PWA

Il passaggio successivo per avvicinarsi a prestazioni e interazioni native per il client, allontanandosi ulteriormente dal modello Blazor Server, è chiamato Blazor PWA. È così chiamato perchè in questo modello Blazor, permette di sviluppare l'interfaccia utente di una Progressive Web-App(PWA).

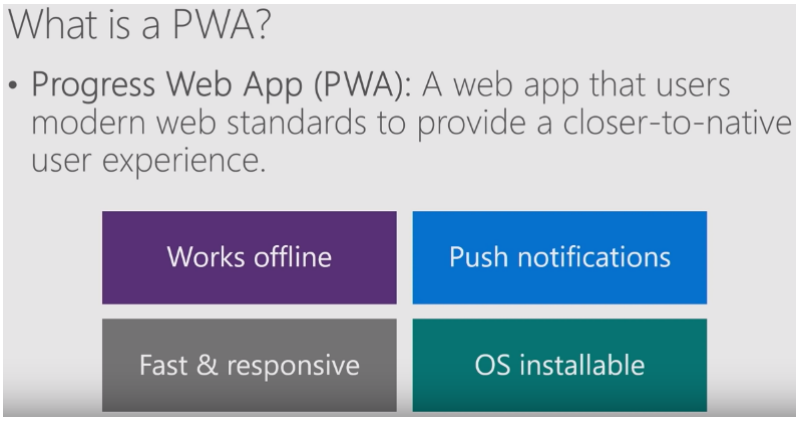

Figura 2.6: Progressive Web Apps

Nella figura 2.6 viene riassunto cosa siano le PWA ovvero applicazioni web che hanno la capacità di funzionare anche offline, possono essere scaricate in modo persistente sulla macchina dell'utente che le esegue. Offrono il vantaggio di una maggiore velocità di esecuzione e la possibilità di sfruttare alcune API native. Questo modello quindi, può tornare utile quando la necessità dell'applicazione che si vuole sviluppare è quella di utilizzare le notifiche push native del sistema operativo che sta utilizzando il client(al momento solo Windows).

Al momento, per realizzare una PWA utilizzando Blazor, bisogna partire dal modello Blazor WebAssembly aggiungendo un manifesto che descriva le capacità 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[16].

Un service worker Ã¨ uno script che viene eseguito in background ed Ã¨ delegato a gestire le attivitÃ di caching del sito.

Una cosa molto importante da tenere presente, è che le demo esistenti al momento per le PWA scritte in Blazor si basano sul browser Chrome e sul sistema operativo Windows. Per ora non risultano cross-browser e cross-platform.

2.3 Modelli Sperimentali

2.3.1 Blazor Hybrid

Blazor Hybrid permette di sviluppare applicazioni parzialmente native. Nel modello Hybrid, l'applicazione sviluppata non è quindi più considerabile un'applicazione web ma rimane ibrida perchè pur essendo un'applicazione con capacità native, utilizza tecnologie web per effettuare il rendering della UI.

Esempi di Hybrid Apps possono essere quelle applicazioni mobile native che hanno accesso alle API esposte 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, scritte in Blazor, che sfruttano Electron.

Electron, precedentemente noto come Atom Shell, è un framework open-source sviluppato e mantenuto da GitHub. In particolare Electron permette di sviluppare interfacce grafiche per desktop eseguibili su Windows, Linux e macOS utilizzando tecnologie web[17]. Electron combina il motore di rendering del browser Chromium ed il runtime Node.js, che vengono inclusi in ogni applicazione sviluppata utilizzando questo framework. Utilizzando quindi il pacchetto NuGet Electron.NET, si può fare in modo di eseguire un'applicazione .NET Core nell'ambiente(browser e runtime) garantito da Electron, utilizzando Blazor in fase di sviluppo[18].

In questo modo si può ottenere un'applicazione con capacità native, che sia cross-platform, la cui interfaccia sia stata scritta utilizzando Blazor. Nella figura 2.7 si può vedere una Web Application compilata nativamente con Electron(con target Windows), e quindi eseguita come applicazione desktop:

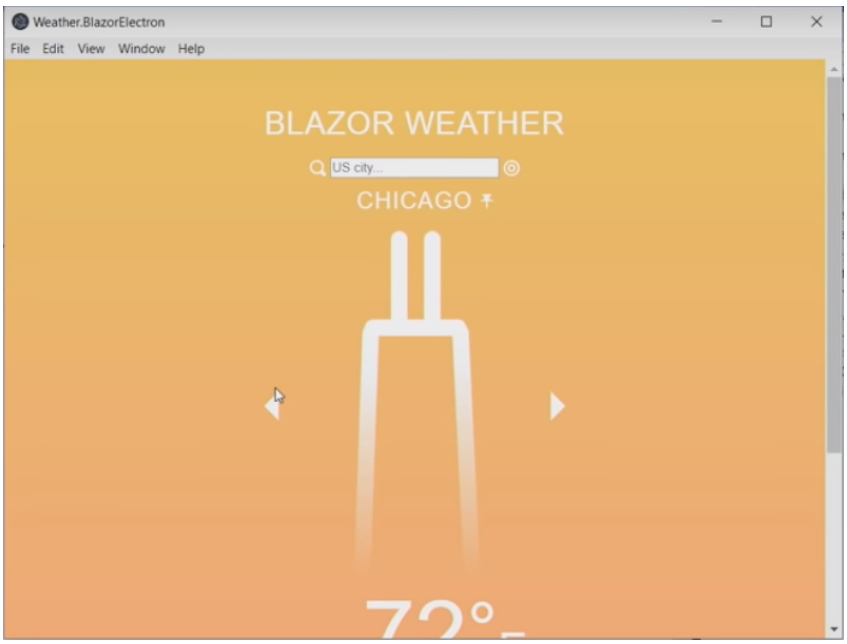

Figura 2.7: Blazor Hybrid Application

Il codice open source di questa applicazione, si può trovare al seguente link: https://github.com/danroth27/BlazorWeather/tree/master/BlazorWeather.Electron

2.3.2 Blazor Native

Grazie a Blazor Native è possibile sviluppare applicazioni completamente native, sfruttando il fatto che Blazor è stato architettato per poter renderizzare controlli della UI che non siano obbligatoriamente strumenti web, e può quindi integrarsi con controlli nativi. Il rendering layer è infatti intercambiabile, pur essendo quello di default dedicato all'HTML.

Un esempio di applicazione sviluppata utilizzando Blazor per il rendering di controlli nativi nella UI, si può vedere durante la presentazione di Steve Sanderson all'evento NDC ad Oslo del 2019, pur non essendo stato ancora rilasciato il codice di un esempio ufficiale[19]. In questa applicazione, si è scelto di sostituire il default rendering layer per utilizzarne uno personalizzato, 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 è presente nella documentazione ufficiale ed è solo stato citato da Daniel Roth durante la presentazione dei futuri modelli di Blazor sul client[20]. Blazor Native è quindi da considerare del tutto sperimentale.

2.4 Funzionamento

Quando si scrive codice nei component Blazor, 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.

Writing Blazor Code

```
<div>
    <FancySpan IsBold="@(currentCount % 2 == 0)">So exciting</FancySpan>
    <p>Current count: @currentCount</p>
</div>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Figura 2.8: File Razor

Si può scrivere utilizzando questa sintassi solamente nei file che hanno estensione ".razor", il cui esempio si può trovare in figura 2.8.

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 è descritto[21]. Un esempio di output di file razor compilato, generato a partire da quello in figura 2.8 è visibile in figura 2.9.

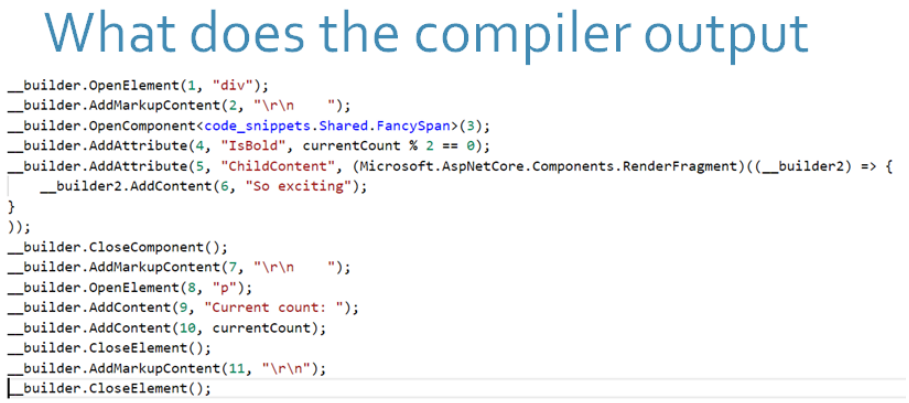

Figura 2.9: Output

Saranno poi questi file ad essere effettivamente utilizzati da Roslyn(che è 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 è solo necessario per fare in modo che si possano generare le DLL compilate relative ai file di partenza, ma è anche il momento in cui viene ottimizzato ciò 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 visibili nella figura 2.10.

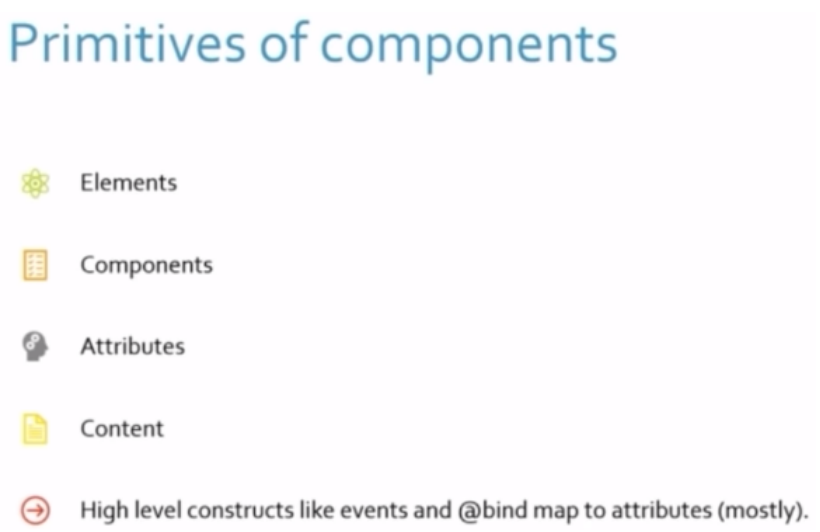

Figura 2.10: Primitive di Blazor

In ordine quindi possiamo descrivere le primitive:

1. Un elemento è ad esempio l'elemento html "div";
2. Un component è un altro componente(file .razor) che viene utilizzato nel file che si sta compilando;
3. Un attributo è appunto un attributo di un elemento html, o un parametro passato in ingresso ad un component;
4. Un contenuto è 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

Confronto tra Client e Server

3.1 Scalabilità

La Scalabilità, ossia la capacità dell'applicazione di resistere all'aumentare del numero di utenti concorrenti che ne usufruiscono, dipende fortemente dal modello scelto, come anche dalla tipologia di applicazione sviluppata. Quindi scegliere il modello più adatto non è banale e non ne esiste uno standard sempre corretto.

Il problemi introdotti dalla scalabilità comunque sono fondamentalmente diversi tra Blazor Server, e tutti gli altri modelli. Questo principalmente per 3 motivi:

1. Blazor Server delega completamente al server che ospita l'applicazione il carico computazionale necessario per gestire ogni singolo evento della UI di ogni sessione per ogni utente connesso, compreso il salvataggio in memoria RAM dello stato di ciascuna UI durante l'utilizzo dell'applicazione, visto che ad ogni browser coinvolto, dopo l'inizializzazione, arrivano solamente le minime differenze necessarie per effettuare il render della UI rispetto al frame precedente, ogni volta che deve cambiare qualcosa. Ciò 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 è che Blazor Server, necessita di almeno una connessione costante ed affidabile con ogni sessione di ogni utente collegato.

3. Il terzo è 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 è consigliato che lo supporti, pur non essendo necessario.

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 basati su Javascript come Angular o React, in particolare Blazor WebAssembly. Di seguito verranno quindi confrontati pro e contro dell'utilizzo del modello Blazor Server e del modello Blazor WebAssembly, partendo dai punti evidenziati nella figura 3.1.

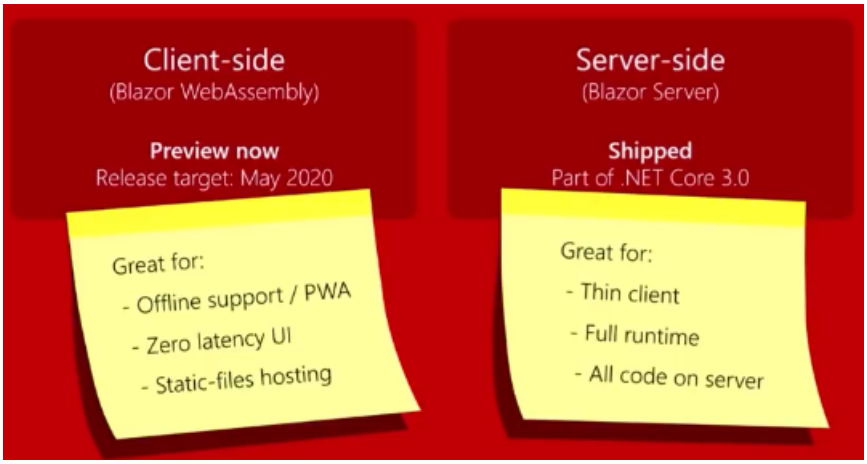

Figura 3.1: Confronto Modelli Client-Server

3.2 Blazor Server

3.2.1 Pro

Blazor Server vanta, come primo punto a favore, il fatto di essere stato creato per essere una soluzione completa, ma che può essere utilizzata in modo complementare in applicazioni già esistenti che ad esempio si basano su Razor. Si può quindi andare ad ampliare queste applicazioni senza doverle riscrivere da zero. In questo modo si possono continuare a sfruttare le pagine completamente renderizzate lato server dove necessario, ma si possono anche gestire le interazioni del client con la UI potendo referenziare codice C# senza doverlo riscrivere in Javascript o anche solo passare tramite JS, in parti dell'applicazione dove si ritiene sia più adatto farlo.

Un altro grande pro di questo modello è il fatto di poter delegare al server la responsabilità di dover sopportare un peso maggiore, al crescere dell'applicazione, mentre il client non deve scaricare più dati. Infatti i file scaricati lato client sono solo quelli necessari all'inizializzazione della UI e allo stabilimento di una connessione con il server, che quindi al crescere del peso dell'applicazione, non invia dati in più ai client, ma continua solamente a inviare le differenze della UI rispetto allo stato precedente. Questo secondo vantaggio, rende il modello ideale per i casi in cui si deve creare un'applicazione per nella

quale i client che la utilizzeranno potrebbero avere a disposizione computer o cellulari a basso costo.

Il terzo punto a favore, molto utile durante lo sviluppo di applicazioni è che il codice dinamico relativo alla logica dell'applicazione sviluppata, comprese eventuali logiche di business, non escono mai dal server, in quanto tutto viene eseguito lato server dopo la connessione iniziale.

Il quarto punto a favore, è che se si vuole sviluppare un'applicazione Blazor da utilizzare in produzione, fino a maggio 2020 questo modello è l'unico ad essere ufficialmente supportato e che si può quindi cominciare già ad utilizzare.

Un quinto punto a favore, è che essendo questo modello pronto per produzione, si trovano già oggi diverse guide online di utenti che hanno provato ad utilizzarlo e che si sono scontrati con i principali problemi esistenti, e in caso di problemi si può eseguire il debug del codice scritto utilizzando Visual Studio 2019 o Visual Studio Code o ricevere supporto ufficiale da Microsoft nel caso si trovi un problema negli strumenti che si hanno a disposizione.

Infine il fatto che l'applicazione esegua sul server, implica che si possa utilizzare l'intero runtime di .NET Core, che implementa completamente il .NET Standard 2.0.

3.2.2 Contro

Il principale punto a sfavore di Blazor Server è la già citata RAM consumata da ogni UI per ogni sessione di ogni utente che utilizza l'applicazione. Un benchmark eseguito da Microsoft a tal proposito, ha mostrato come il principale collo di bottiglia di un'applicazione Blazor sia proprio la RAM. Microsoft ha dichiarato che hanno stimato vengano occupati in media 250 KB in RAM per ogni nuova connessione di un utente ad un'applicazione base di tipo "hello world" e consigliano di dedicare almeno 273 KB per utente se non si ha chiaro da dove partire.Durante il test di carico svolto da Microsoft è stata utilizzata una sola macchina virtuale Azure di fascia medio-bassa(Standard D1V2)come server per ospitare l'applicazione Blazor, avendo quindi a disposizione 3.5 GB di RAM e le caratteristiche visibili in figura 3.2.

Dimensione	vCPU	Memoria: GiB	GiB di archiviazione temp (unità SSD)	Velocità effettiva massima di archiviazione temporanea: IOPS/Mbps di lettura/Mbps di scrittura	Valore massimo per dischi di dati	Velocità effettiva: operazioni di I/O al secondo	Schede di interfaccia di rete max/larghezza di banda della rete prevista (Mbps)
Standard_D1_v2	1	3,5	50	3000 / 46 / 23	4	4x500	2 / 750
Standard_D2_v2	2	7	100	6000 / 93 / 46	8	8x500	2 / 1500
Standard_D3_v2	4	14	200	12000 / 187 / 93	16	16x500	4 / 3000
Standard_D4_v2	8	28	400	24000 / 375 / 187	32	32x500	8 / 6000
Standard_D5_v2	16	56	800	48000 / 750 / 375	64	64x500	8 / 12000

Figura 3.2: Parametri VM utilizzata per il test di carico

Con questi parametri, l'applicazione è stata in grado di gestire 5000 utenti concorrenti senza perdita significativa di performance [12] [22]. Bisogna quindi tener conto che al crescere delle informazioni da tenere in memoria per ciascun utente, l'applicazione sviluppata richiederà più RAM per ciascuno, quindi per questo modello i costi dell'infrastruttura dipendono fortemente dai volumi di utenti concorrenti che la utilizzano.

Un altro punto a sfavore, è la certezza che l'applicazione non sarà performante per gli utenti che al momento del collegamento si trovano molto lontani(ad esempio in un continente diverso) dal server sulla quale questa è ospitata. Questo avviene perchè ogni update della UI deve prima avvenire sul server, quindi la latenza(ping) e la sua variazione(jitter) diventano estremamente importanti per l'esperienza dell'utente.

Infine bisogna tener conto che ogni utente che vuole poter utilizzare l'applicazione, deve potersi connettere ad essa, e restarci connesso per tutta la durata del suo utilizzo, dato che se la connessione SignalR sulla quale si basa Blazor Server per funzionare viene a mancare, in automatico Blazor proverà a ricollegarsi al server, ma fino a quando non ci riuscirà l'applicazione sarà inutilizzabile.

3.3 Blazor WebAssembly

3.3.1 Pro

Il principale vantaggio di Blazor WebAssembly, specialmente in applicazioni complesse, è quello di poter sfruttare le risorse dei client che vogliono utilizzare l'applicazione, senza sovraccaricare il server, specialmente quando il numero di utilizzatori è molto alto. Questo modello infatti, dopo lo scaricamento iniziale dell'applicazione, del runtime mono.wasm e delle DLL sulle quali l'applicazione si basa, si può procedere ed utilizzarla senza più essere collegati ad internet,

almeno per ogni parte nella quale non si cerca di comunicare in modo sincrono con il server.

Altro grande vantaggio, è quindi il fatto che dopo lo scaricamento iniziale e l'inizializzazione, l'applicazione sarà responsiva senza alcuna latenza, dato che al contrario del modello Blazor Server, la gestione degli eventi da parte dei component di Blazor avviene in locale, senza dover comunicare ogni volta con il server.

Un altro punto a favore al quale prestare attenzione è la maggiore facilità con cui si può trasformare questo modello in una PWA, anche se al momento della scrittura di questo lavoro di tesi non esiste ancora una guida ufficiale pubblicata da Microsoft. In alcuni casi questo modello può essere la scelta giusta e risultare anche più veloce di un'applicazione equivalente con componenti scritti in Javascript, dato che il codice dei componenti Blazor viene compilato nel .NET Intermediate Language come visibile nella figura 2.4 prima di essere eseguito lato client tramite il runtime Mono tradotto in WebAssembly contenuto nel file mono.wasm.

Il fatto stesso di essere equivalente ad un'applicazione moderna scritta usando Javascript agli occhi di un utente ma di poter arrivare ad avere prestazioni migliori e di scrivere in un linguaggio diverso dal solo javascript, rende questo modello estremamente importante e indicato per applicazioni dove ad esempio si vuole poter condividere il codice di validazione ed utilizzarlo tra frontend e backend. Un'altra categoria di applicazioni per le quali Blazor WebAssembly può risultare più indicato, è quella dei videogiochi con alta frequenza di aggiornamento nella UI dell'utente, con particolare enfasi su quelli per i quali la grafica è complessa e risulterebbe troppo lento e costoso far avvenire i cambiamenti lato server, causando lag e degradando l'esperienza degli utenti.

3.3.2 Contro

Il primo svantaggio di Blazor WebAssemblyè quindi il peso dell'applicazione, che influisce negativamente sullo scaricamento iniziale. Ciò implica anche che quanto più codice viene scaricato, tanto più può essere studiato e utilizzato per capire logiche di business o cercare modi per attaccare il server. Questa cosa è vera anche per le applicazioni odierne scritte utilizzando Javascript, ma rimane uno svantaggio perchè il modello Blazor Server non presenta questa caratteristica.

Un ulteriore svantaggio da tenere in considerazione, è che come per ogni nuova tecnologia, la documentazione, il supporto della community, e le librerie dedicate, in questo primo momento sono e saranno fortemente limitate rispetto a framework per UI basati su Javascript equivalenti in teoria ma ben più consolidati e diffusi in pratica. Il modello Blazor WebAssembly inoltre, sfruttando il WebAssembly, necessita che il browser di ogni client che vuole poter utilizzare l'applicazione lo supporti.

L'ultima grande pecca di questo modello èsicuramente il fatto che sia ancora immaturo, e non ancora pronto per essere utilizzato in produzione. Prima di tutto perchè non è ancora stato ufficialmente rilasciato, dato che il rilascio è previsto per maggio 2020. In secondo luogo perchè gli strumenti a disposizione sono ancora acerbi e non è possibile fare cose fondamentali come eseguire il debug del codice direttamente da Visual Studio 2019 o da Visual Studio Code in modo consistente. Infine perchè pur essendo Blazor WebAssembly ottimizzato per essere molto efficiente durante il rendering della UI, c'è ancora molto margine di miglioramento quando l'applicazione Blazor scritta è CPU intensive. Questo semplicemente perchè ora come ora il codice viene scaricato sotto forma di DLL ed interpretato dal runtime di Mono tradotto in WASM, e non direttamente eseguito. D. Roth ha dichiarato che il team che si occupa di sviluppare Blazor sta lavorando per rendere possibile la compilazione del codice da .NET direttamente in WASM, così da ottenere performance decisamente migliori[12].

Capitolo 4

Conclusioni

4.1 Conclusioni

Questo studio, ha cercato di rispondere alla domanda: "Cos'è Blazor?". A tal fine, è stato descritto prima di tutto quali siano le tecnologie utilizzate da questo framework che in parte cambiano a seconda del modello scelto. È stata implementata una applicazione web sfruttando Blazor Server, e dove disponibili sono stati mostrati gli esempi disponibili per gli altri modelli. Ciò è stato fatto per spiegare il diverso funzionamento, a seconda del modello scelto, con i pro e contro di ciascuno. È emerso che non esiste una scelta universalmente giusta, poiché a seconda delle esigenze dell'applicazione da sviluppare, un modello può risultare il migliore o il peggiore

4.2 Futuro del progetto

Figura 4.1: Modelli blazor e supporto

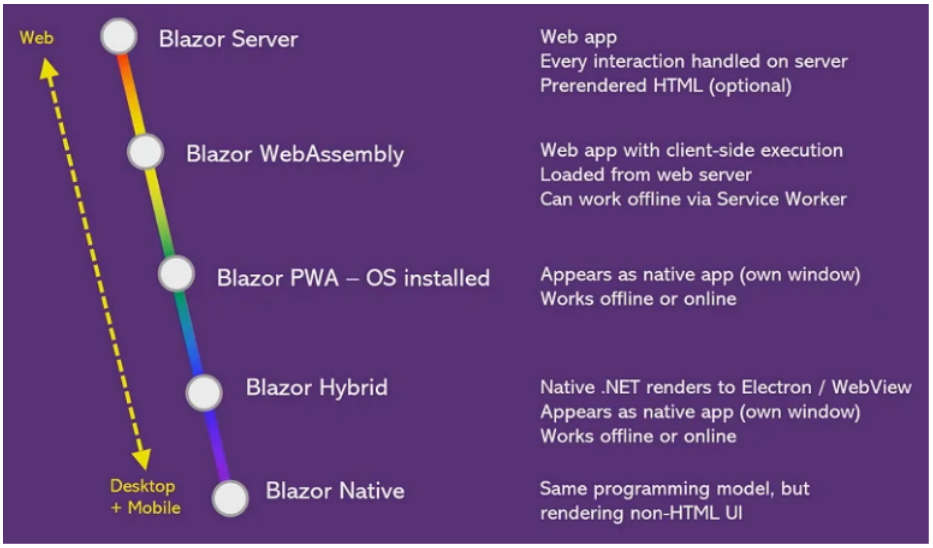

Figura 4.2: Modelli attuali e futuri

Onbeforeunload

Server

WebAssembly, promettente ma ancora interpretato, si dovrebbe tradurre .NET direttamente in WebAssembly e in modalitÃ AOT per avere prestazioni ben migliori e giustificare il non utilizzo di JS.

In Roadmap PWA solo su Chrome Electron, bello ma molto pesante Native –> la complessitÃ non giustifica la scelta, al momento e un esercizio accademico.

Bibliografia

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

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

[3] Disponibile: https://insights.stackoverflow.com/survey/2019#technology-_-web-frameworks

[4] Disponibile: https://it.wikipedia.org/wiki/Vue.js

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

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

[7] Disponibile: https://en.wikipedia.org/wiki/Single-page_application

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

[9] L. Latham, D. Roth Disponibile: https://docs.microsoft.com/en-us/aspnet/core/blazor/components?view=aspnetcore-3.1

[10] Disponibile: https://dotnet.microsoft.com/apps/aspnet/signalr

[11] https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1

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

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

[14] Disponibile: https://webassembly.org/

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

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

[17] Disponibile: https://en.wikipedia.org/wiki/Electron_(software_framework)

[18] Disponibile: https://github.com/ElectronNET/Electron.NET/blob/master/README.m

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

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

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

[22] Disponibile: https://docs.microsoft.com/it-it/aspnet/core/host-and-deploy/blazor/server?view=aspnetcore-3.1#deployment-server

Ringraziamenti

Vorrei ringraziare il professor Lattanzi per avermi aiutato a scrivere questa tesi, ed i miei genitori per avermi dato la possibilità di studiare ciò 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 capacità anche quando non l'ho fatto io.