Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

---

Tesi di Laurea Triennale in Ingegneria Informatica

L'approccio Microfrontend nelle applicazioni web: Studio delle metodologie a disposizione e presentazione dei risultati del tirocinio con oggetto la migrazione a microfrontend delle applicazioni web nelle sale di controllo X2030 di Leonardo SPA

Candidato
Matteo Bavecchi

Relatori
Prof. Romano Fantacci
Prof. Tommaso Pecorella

Correlatore
Dott. Andrea Rizzo

Anno Accademico 2020/2021

Indice

Ringraziamenti

Grazie a tutti

Introduzione

Negli ultimi due anni a causa della crisi pandemica abbiamo assistito ad un aumento esponenziale del commercio online: secondo ISTAT nel nostro paese si è registrato un aumento del 50,2% nelle vendite e-commerce. [2] Il lavoro inoltre ha subito un cambiamento repentino, facendo migrare milioni di persone dall'ufficio alle loro mura domestiche. Tutti questi fenomeni si sono realizzati grazie a internet, che ha accolto tutte quelle interazioni che fino a qualche mese prima erano fatte fisicamente. Questo ha portato ad avere sempre più traffico e una maggiore richiesta di robustezza, sicurezza e ridondanza da parte delle aziende. Basti pensare al danno economico che hanno avuto le oltre **200 milioni** di attività commerciali che stavano pubblicizzando i loro beni su Facebook durante le 14 ore di disservizi lo scorso 4 ottobre [4]. Organizzare il lavoro per lo sviluppo di applicazioni web di grandi dimensioni, capaci di accogliere numeri sempre maggiori di utenti, non è per niente banale, ed è molto interessante capire come suddividere responsabilità tra i vari team che contribuiscono alla sua realizzazione.

L'approccio più diffuso per affrontare il problema è quello di suddividere le persone per competenze (ovvero in modo *orizzontale*), creando team che mettono in comune figure con abilità dello stesso ambito. Il classico approccio monolitico infatti prevede due suddivisioni, la parte *frontend*, che si occupa dell'esperienza utente e delle interfacce e la parte *backend*, che invece gestisce

server e basi di dati. Ad esempio in un sito e-commerce possiamo trovare un team che si occupa della parte frontend, uno che cura i servizi di pagamento e uno che segue la parte backend.

Quando il progetto aumenta di complessità, si sente la necessità di suddividere il lavoro in sotto-progetti, e l'approccio orizzontale potrebbe non essere la scelta migliore, in quanto rallenta l'introduzione di nuove funzionalità.

Possiamo allora pensare di assegnare ad ogni team una parte del progetto, i quali dovranno portarlo al termine interamente. Ogni team avrà bisogno quindi di competenze eterogenee al loro interno.

Abbiamo così il superamento dell'approccio monolitico, e possiamo parlare di microfrontend.

Questo già avviene analogamente nello sviluppo backend, dove l'applicazione viene vista come un insieme di *microservizi*.

Capitolo 1

Microfrontend

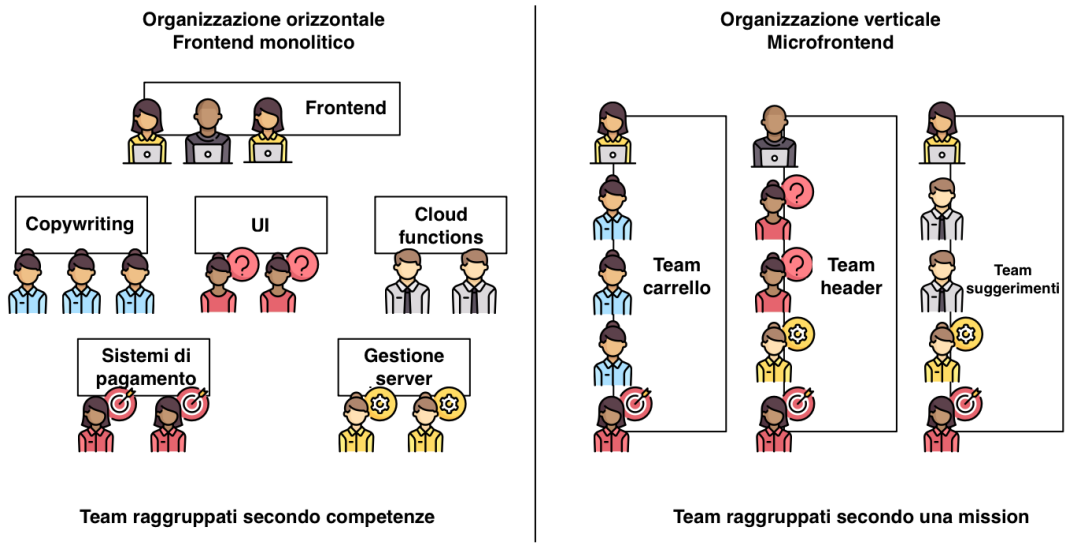

Figura 1.1: Differenze di composizione di team tra approccio monolitico e microfrontend

L'obiettivo della tecnologia microfrontend è quello di superare l'approccio monolitico, che vede lo sviluppo di applicazioni web suddiviso in due team: backend e frontend.

Si fa questo vedendo un'applicazione web come un insieme di elementi, chiamati fragment o microfrontend, molto disaccoppiati tra di loro e con la più bassa granuralità, ovvero con la funzionalità più minimale possibile.

Ogni microfrontend viene sviluppato da un team, che potrà lavorare con autonomia. Essendo i singoli microfrontend autonomi, questi possono funzionare anche se estratti dalla applicazione web che li contiene, e il malfunzionamento di un singolo microfrontend non compromette la stabilità degli altri. I team sono autonomi, ma non sono isolati: questi infatti condividono un insieme di dati, che chiameremo *contratto*, fondamentali per il corretto funzionamento dell'applicazione. A seconda delle tecnologie utilizzate per realizzare l'architettura microfrontend, il contratto tra i team sarà più o meno complesso.

1.0.1 Vantaggi

- **Ottimizzare lo sviluppo di funzionalità:** Nell'approccio orizzontale quando si vuole sviluppare una nuova funzionalità è necessario far convergere il lavoro di più team. Con microfrontend, tutte le persone coinvolte nell'implementazione di una nuova funzionalità sono nello stesso team, rendendo il lavoro più veloce ed efficiente.

- **Abbandono del frontend monolitico:** Con microfrontend le applicazioni, incluse il frontend, si dividono in sistemi verticali più piccoli. Ogni team controlla la sua piccola parte di frontend e di backend.

- **Adottare diverse tecnologie:** Strumenti di sviluppo e framework evolvono continuamente. Ogni team deve essere in grado di scegliere le proprie tecnologie autonomamente. Ci sono alcune grandi aziende come Github, che hanno impiegato molto tempo per eliminare alcune

dipendenze ormai obsolete dal loro codice (nel caso di Github si trattava di una versione di JQuery). Con l'approccio microfrontend questi cambiamenti sono più rapidi e possono essere fatti modularmente.

- **Indipendenza:** I progetti di ogni team sono autonomi, ovvero non hanno dipendenze condivise tra di loro. Questo come già detto precedentemente porta ad una grande autonomia.

  L'indipendenza però porta sicuramente a costi aggiuntivi. Si potrebbe pensare che sia più semplice quindi di sviluppare un unico progetto, e assegnare parti di questo a team diversi. Il problema sta nel fatto che la comunicazione tra i vari team è costosa e porta molti ritardi. In alcuni casi quindi è preferibile introdurre ridondanza nel codice dei vari team a favore di più autonomia e velocità di implementazione.

1.0.2 Svantaggi

- **Ridondanza:** In informatica si è addestrati a ridurre al minimo le ridondanze. Anche nel caso dello sviluppo frontend ci sono degli episodi nei quali la ridondanza può essere molto costosa: come ad esempio quando viene trovato un bug in una libreria e questo viene risolto da un team, il fix dovrà essere comunicato agli altri e questi dovranno provvedere a risolverlo autonomamente. Oppure quando si rende un processo più veloce, anche in questo caso va comunicato agli altri team e questi dovranno apprendere la scoperta. Si adotta quindi un approccio microfrontend quando i costi associati a ridondanze sono inferiori agli impatti negativi a uno sviluppo frontend monolitico, che porta a forti dipendenze tra team.

- **Inconsistenza:** Potrebbero essere presenti delle basi di dati, necessarie all'applicazione web, che vengono lette e scritte da più microfrontends. Per garantire la proprietà di indipendenza tra team è necessario replicare il database per tutti i progetti. Tutte queste copie devono però essere sincronizzate tra loro regolarmente per mantenere la consistenza e la coerenza dei dati. Questo introduce dei ritardi, che potrebbero penalizzare l'esperienza utente.

- **Eterogeneità:** Potrebbe essere controverso avere la libertà di utilizzare tecnologie diverse tra i vati team. Ovviamente se ogni team usa una diversa tecnologia lo scambio di pareri o di competenze tra team diventa più difficile.

Capitolo 2

Composizione

Per far convivere i progetti realizzati dai vari team esistono varie metodologie. Questo presuppone che i team, nella loro autonomia, debbano comunque scambiarsi un minimo di informazioni e di vincoli che servono alla corretta integrazione delle app web, chiamiamo questo insieme di dati *contratto tra team*.

2.1 Collegamento tra pagine con links

La soluzione più semplice è quella del collegamento tra pagine con link URL: in questo modo il contratto consiste negli indirizzi delle pagine, che i team dovranno scambiarsi tra di loro. Questa soluzione è quella che rende i team il più autonomi e disaccoppiati possibile, inoltre si ha grande robustezza, in quanto se un progetto si corrompe, questo non influenza minimamente gli altri, che possono anche essere detenuti in server diversi.

Lo svantaggio è che in una pagina web è possibile contenere informazioni provenienti da un solo team.

Questa soluzione viene usata quando è richiesta una forte robustezza, oppure quando si deve implementare i microfrontends in una app legacy già esistente.

2.2 Composizione tramite iframes

L' iframe è un elemento HTML che permette di incorporare un'altra pagina HTML all'interno di quella corrente. [**?**]

Questo elemento può essere usato per comporre fragments provenienti da teams differenti, visualizzandoli su un'unica pagina web. Un team si impegnerà a realizzare la pagina ospitante e dovrà far presente agli altri attori lo spazio riservato ai loro fragments per evitare problemi di visualizzazione, questo dato, oltre che agli indirizzi URL delle varie pagine, farà parte del contratto tra i team.

Un grande svantaggio dell'uso degli iframes è la loro incompatibilità con i motori di ricerca. Infatti le informazioni che vediamo nella pagina non sono in unico file HTML, di conseguenza il motore di ricerca non profila le informazioni contenute negli iframes.

2.2.1 Composizione con Ajax

Un modo per superare il problema dei motori di ricerca ai quali sono affetti gli iframes è quello di caricare i file HTML con Ajax.

La pagina ospitante, detta *wrapper* ha il compito di caricare nel proprio DOM (Document Object Model) i fragments dei vari team, con l'ausilio di un codice javascript, che utilizza la funzione fetch().

Il problema principale però è che l'Ajax request è asincrona, questo porta a dei ritardi nel caricamento completo della pagina.

2.2.2 Utilizzo del Routing Server-side

Il routing è un elemento fondamentale dell'architettura microfrontend. L'elemento che stiamo introducendo viene chiamato frontend proxy, cioè un web server che effettua routing, che intercetta le richieste con un certo percorso e le instrada al giusto fragment. In questo modo anche se i progetti dei vari team risiedono in spazi diversi, l'utente vedrà un URL omogeneo e non si accorgerà delle origini dei fragment. Il funzionamento consiste in questi passaggi:

- Il client apre l' URL "/foo/bar". La richiesta raggiunge il frontend proxy
- Il frontend proxy confronta il path "/foo/bar/"8 con la propria routing table, con corrispondenza in prossimità della regola i
- Il frontend proxy passa la richiesta al fragment associato
- Il fragment genera una risposta e la ritorna al frontdend proxy
- La risposta arriva infine al client

2.3 Composizione Server-side

Si suppone di lavorare ad un progetto microfrontend incentrato principalmente su contenuti e informazioni, senza particolari applicativi logici al suo interno. Un esempio di sito incentrato sulle informazioni(anche detto *content-centric*) è Wikipedia, le sue caratteristiche principali sono:

1. Tempi brevi di caricamento delle pagine
2. Ottimizzazione per motori di ricerca
3. Assenza di particolari contenuto logici basati sull'interazione con l'utente

Si introduce quindi la **composizione server-side**, che rispetta le richieste di un sito content-centric come Wikipedia.

La composizione dei fragment viene eseguita da un servizio che risiede nel server web, il client quindi riceve la pagina completa.

Come nel caso degli iframes, i team che producono i fragments devono fornire al team che detiene la pagina ospitante l'URL del loro codice di markup HTML.

Il team della pagina ospitante userà delle direttive per richiedere al server di aggiungere il codice di markup degli altri team in un preciso posto della schermata.

Un esempio di servizio di composizione è la funzione **SSI** (Server-Side Includes) di **Nginx**:

2.3.1 Server-Side Includes

Nginx contiene il modulo *ngx-http-ssi-module* che processa i comandi SSI. I comandi SSI sono istruzioni inserite nelle pagine HTML della pagina

ospitante. Un'istruzione SSI appare nella seguente sintassi:

```
<!--#include virtual="url/da/includere -->
```

Avviene quindi questo:

1. Il client effettua la richiesta della pagina
2. Il webserver accetta la richiesta e verifica la presenza di eventuali direttive SSI
3. Le direttive trovate vengono sostituite con il codice di markup reperibile all'url dell'attributo "virtual"
4. Il client riceve la pagina completa

Si noti che le prestazioni di composizione sono nettamente aumentate rispetto alla composizione con iframes, in quanto il client riceve la pagina già completa, effettuando un solo handshake HTML, indipendentemente dal numero di fragment ospitati nella pagina. Inoltre i motori di ricerca possono profilare anche il contenuto dei fragments

Modalità di caricamento dei fragments

Al contrario della soluzione con iframes, lo scorretto caricamento di un fragment può rallentare o addirittura bloccare il caricamento dell'intera pagina. Il webserver infatti prima di restituire la pagina aspetta di avere tutti i suoi componenti. La proprietà di Nginx chiamata *proxy read timeout* permette di fissare un tempo massimo per il quale il webserver può aspettare un fragment.

Nel caso in cui il webserver non riesce a caricare un fragment esiste il comando SSI **stub**. Si utilizza la direttiva *block* che sostiuirà il fragment

caricato senza successo con del codice HTML di riserva. Il nuovo *include* contiene il nuovo attributo stub, che ha il riferimento al block corrispondente, dovesse non andare a buon fine la richiesta.

```
   <!--# block name="fallback" -->
<a href="/page"> Link</a>
<!--# endblock -->
 <!--#include
virtual="/link/to/page"
 stub="fallback" -->
```

Quando Nginx deve scaricare più fragments, effettua i caricamenti in parallelo. Quando l'ultimo fragment viene scaricato,la pagina viene composta e inviata al client. Il tempo di risposta dell'intera pagina è chiamato *time to first byte* (TTFB) ed è definito come il tempo necessario per generare l'HTML della pagina, più il tempo di caricamento del fragment più lento.

E' possibile anche annidare fragments, ma non è consigliato, in quanto si interrompe la parallelizzazione dei caricamenti e si rallenta la composizione della pagina.

Caricamento differito

In genere si ottimizza il caricamento della pagina effettuando una composizione server-side per la sua parte principale, la cosiddetta *viewport*, le componenti secondarie invece è preferibile comporle lato client, con delle chiamate Ajax, utilizzando la metodologia prima descritta. Questa tecnica si chiama caricamento differito, o **lazy loading**.

Alternative a SSI

Il webserver con SSI, invia la pagina al client solo quando questa è stata completamente composta. Il servizio di composizione **ESI** presente nel webserver Vernish invece attua l'*invio parziale*, ovvero inizia a inviare parti di pagina anche prima che questa venga completamente assemblata.

Esistono altre soluzioni, come quella sviluppata dalla società di fashion e-commerce Zalando, chimata Zalando Tailor. L'azienda migrò infatti il loro microfrontend dall'approccio monolitico a quello microfrontend, utilizzando una composizione server-side. Per via dell'entità del sito web, il team Zalando ha avviato il progetto Tailor, che ha poi portato ad un rilascio ufficiale con licenza **TODOlibera**. Tailor consiste in una libreria Node.js, reperibile nel pacchetto NPM *node-tailor*.

Tailor riesce ad avere prestazioni migliori rispetto all'invio parziale, facendo in modo di iniziare a inviare parti di pagina al client, ancora prima che il server abbia finito di scaricare l'intera pagina.

2.4 Composizione Client-side

Ci sono dei siti web che non hanno principalmente uno scopo informativo, ma funzionale, ovvero rispondono all'input dell'utente e rilasciano un output influenzato da tale input. Questi siti web vengono detti *behavior-centric*, cioè incentrati su un comportamento o funzionalità. Il modo più indicato per chiamarli non è siti web ma web apps, applicazioni web. Una web app è ad esempio Google Meet, raggiungibile dall'URL https://meet.google.com. Questo non è un sito che veicola principalmente informazioni, ma bensì offre un servizio all'utente: permette a due o più persone di interloquire, tramite scambio di segnali audiovisivi. Per fare in modo di reagire all input degli utenti, le webapp cambiano il loro codice HTML nel browser, e inoltre permettono di cambiare pagina, e quindi anche URL, senza effettuare alcuna richiesta al webserver. Sono disponibili agli sviluppatori numerosi framework per realizzare webapps, come React.js, Angular e Vue.js. Seguendo un approccio monolitico dovremmo scegliere uno di questi framework ed essere costretti a farlo utilizzare a tutti i team che lavorano al progetto. Al contrario, con l'approccio microfrontend, si da libertà ad ogni team di scegliere autonomamente la soluzione migliore e di mantenerla aggiornata. Grazie agli **web components**, i team possono realizzare dei fragments con qualsiasi tecnologia a loro disposizione e farli operare con il resto della pagina.

2.4.1 Web Components

Gli web components consentono di creare nuovi elementi HTML personalizzati (Custom Elements), riutilizzabili e incapsulati da utilizzare in siti e web app [11].

Custom Elements è un insieme di API Javascript che consentono di defini-

re elementi HTML personalizzati, che includono istruzioni CSS e JS. Ogni custom element è dichiarato nell' oggetto *CustomElementRegistry*.

All'interno di un web component può essere incapsulato elementi di stile e componenti logiche, senza influenzare la parte restante del DOM (Document Object Model), questo grazie alla *shadow DOM*:

Shadow DOM

Sappiamo che il DOM (Document Object Model) è un albero di oggetti creato alla fine del caricamento di ogni pagina web, che permette di accedere dinamicamente e aggiornare il contenuto, la struttura e lo stile di un documento. [10] Possiamo annettere alla struttura del DOM, un numero arbitrario di *shadow trees*, o alberi ombra, questi con i loro elementi e il proprio design hanno una radice, detta *shadow root*. La shadow root è sempre annessa ad un *shadow host* che risiede nel DOM o in un altro shadow tree.

Grazie alla Shadow DOM è possibile gestire singoli elementi di un progetto web in modo indipendente rispetto al resto del sito. Dentro la shadow DOM è possibile gestire contenuti in maniera del tutto indipendente rispetto alle istruzioni di design o dalle strutture valide globalmente nel resto del progetto.

Questo aumenta la robustezza del microfrontend, garantendo isolamento.

Un altro elemento importante che sta alla base dei web components è il concetto di **template HTML**: un modo per creare codice HTML non ancora visualizzato sulla pagina, che può essere istanziato una o più volte durante il runtime grazie a codice javascript.

2.4.2 Comunicazione tra fragments

I fragment possono aver bisogno di ricevere dalla pagina ospitante delle informazioni di contesto, come ad esempio la lingua, la regione geografica, o il nome dell'utente recuperata da un database. Gli attributi dei custom element possono fungere da mezzo di comunicazione per inviare informazioni dalla pagina al fragment. I custom elements, con i quali sono implementati gli web components, forniscono agli sviluppatori una serie di metodi che interessano momenti del loro ciclo di vita:

- **connectedCallback**: invocato quando il custom element viene inserito nel DOM della pagina
- **disconnectedCallback**: invocato quando il custom element viene rimosso dal DOM
- **adoptedCallback**: invocato quando l'elemento viene spostato da un documento ad un altro
- **attributeChangedCallback**: invocato quando un qualsiasi attributo del documento cambia

Si può quindi utilizzare il metodo *attributeChangedCallback* per far reagire il fragment in risposta ad un attributo cambiato.

Per quanto riguarda la comunicazione da un fragment alla pagina, è possibile utilizzare l'API nativa **CustomEvents**, disponibile in tutti i moderni browsers. Permette di emettere eventi personalizzati allo stesso modo di come funzionano gli eventi standard *click* o *change*.

Per far si che il colloquio vada a buon fine, il team che gestisce il fragment deve comunicare a quello che gestisce la pagina il nome dell'evento.

Analizziamo adesso le diverse strategie per far comunicare due fragment ospitati sulla stessa pagina:

- **Comunicazione diretta**: consiste nel chiamare direttamente il metodo del fragment interessato, secondo il principio che qualsiasi fragment ha accesso all'intero DOM della pagina. Questo metodo è altamente sconsigliato, perchè introduce un **forte accoppiamento**

- **comunicazione tramite pagina ospitante**: si usa la pagina ospitante come tramite del messaggio, utilizzando le tecniche prima viste, si emette un evento dal fragment A, si riceve nella pagina, e si inoltra ad un attributo del fragment B.

- **Broadcast Channel API**: la Broadcast channel API permette la comunicazione tra elementi del browser della stessa *origin*, ovvero con hostname, e porta uguali. Si basa sul meccanismo publish/subscribe. Vengono instaurati dei canali nei quali vengono pubblicati messaggi (*publish*), che riceveranno solo i fragment che si sottoscrivono a tale canale(*subscribe*). Questa soluzione riduce al minimo l'accoppiamento tra i vari fragments. Broadcast channel API verrà approfondito successivamente.

E' importante che le comunicazioni tra i fragemnt siano minime e che interessino dati semplici, in quanto un eccessivo accoppiamento tra due oggetti può significare un inesatto confine tra due team.

2.4.3 Routing Client-Side

La maggior parte dei framework client come Angular o React hanno al loro interno un servizio che permette di effettuare routing, attuando una navigazione tra differenti pagine senza fare una completa richiesta al server. In questo modo le webapp con un routing lato client appaiono all'utente più reattive e offrono una migliore esperienza. Queste webapp vengono chiamate Single Page Applications, o SPA.

Si deve innanzi tutto distinguere due tipi di navigazione:

- **Hard navigation**: il browser durante la navigazione carica completamente il codice HTML dal server
- **Soft navigation**: la transizione avviene lato client, in questo modo non si richiede la pagina completa, ma solo alcuni dati, che vengono caricati dal server tramite una API Javascript

Di conseguenza i team possono scegliere autonomamente il tipo di navigazione. Si può trovare una webapp con una hard navigation in tutte le pagine, anche in quelle interne di un team, oppure quella che viene chiamata *linked SPA*, con hard navigation per passare da una pagina posseduta da un team ad un altro e soft navigation per le pagine dello stesso team.

In entrambe queste soluzioni l'unico dato che deve essere condiviso tra i team, il cosiddetto contratto, consiste unicamente nell' URL delle pagine. Questo garantisce una grande autonomia tra team e un alto disaccoppiamento.

Il passo successivo è quello di rendere l'esperienza utente più fluida, introducendo una soft navigation anche tra pagine di team diversi. Per fare ciò è necessario un infrastruttura condivisa, chiamata *Application Shell*.

Application Shell

L' Application Shell, o App Shell, è un app che sta alla base di tutti i microfrontend. Questa si occupa di instradare le richieste e di visualizzare nella pagina HTML il microfrontend giusto. Di solito risiede nella pagina principale(index.html) ed essendo condivisa con tutti gli altri elementi deve essere mantenuta il più semplice possibile. Di solito viene incluso nell'app shell le informazioni di contesto o di autenticazione, utili a tutti i microfrontend.

Un team sarà responsabile della sua manutenzione e dovrà essere a conoscenza del nome dei custom element con i quali gli altri team forniscono le loro web app. Inoltre l'app shell, per instradare correttamente tutti gli indirizzi, deve essere a conoscenza degli url di tutte le pagine di ogni team. Questo obbgliga i team a notificare l'app shell di qualsiasi aggiunta o rimozione di pagine nel loro progetto.

Con il **routing a due livelli** si porta l'app shell in una posizione più neutrale e si diminuisce l'accoppiamento tra questa e gli altri microfrontend.

Routing a due livelli

Risolviamo il problema facendo instradare le pagine all'app shell soltanto secondo il team di provenienza. Ogni team avrà un router interno che assegnerà all'url in arrivo la pagina corrispondente. Per identificare chiaramente il team, si utilizza un prefisso nell'URL.

In questo modo l'app shell viene cambiata solo se si aggiunge un nuovo team o si vuole cambiare il nome dei prefissi.

2.5 Universal Rendering

Dopo aver visto come comporre microfrontend in maniera server-side e client-side, vediamo come combinare queste due tecniche per capire quali vantaggi possono apportare ad un progetto web. Il concetto che sta alla base dell' universal rendering è il seguente: ' Avere un'unica sorgente di codice che è possibile renderizzare sia sul server che su un client ' [6]

Supponiamo di realizzare una pagina per un e-commerce di vestiti. Nella pagina sarà presente l'elemento più importante, quello che riesce a monetizzare la visita di un utente: il pulsante "compra". Il bottone "compra" è un microfrontend con stili e funzioni logiche gestite da un team specializzato, racchiuso in un web component. Data l'estrema priorità di avere visualizzato e funzionante il prima possibile il bottone, il team non si accontenta di una client-side composition, in quanto per dispositivi lenti,con internet scarso, o senza il supporto di javascript potrebbe non funzionare la composizione dell'elemento, il che porterebbe ad un mancato guadagno per il sito.

Una possibile soluzione è utilizzare la Server Side Integration per creare il bottone, prima di consegnare la pagina al client, in questo modo l'elemento verrà subito visualizzato. Successivamente il client attuerà la composizione lato client e il bottone sarà completamente stilizzato.

In questo modo l'utente può comprare il prodotto appena la pagina viene scaricata sul browser.

Vediamo i casi in cui è utile la Universal Rendering: [8]

- **Ottimizzazione SEO**: è necessario per i crawler dei motori di ricerca avere una versione statica del sito web facilmente navigabile senza l'utilizzo di javascript. In questo modo ogni URL ritorna un'anteprima fedele e completa della pagina

- **Performance in dispositivi lenti**: alcuni dispositivi poco potenti o non aggiornati potrebbero non supportare correttamente Javascript, è quindi utile avere una versione del sito funzionante senza l'uso di scripts.

- **Velocità di caricamento**: l'utente ha immediatamente una versione statica del sito completamente navigabile, che successivamente con la composizione client-side diventa anche completamente interattiva. Questo è fondamentale per le landing page, che hanno l'obiettivo di convertire il più possibile le visite

In conclusione universal rendering è la soluzione più complessa di composizione, ma anche l'unica che riesce a unire la velocità di caricamento delle pagine server-rendered e l'estrema velocità di risposta all'input delle pagine client-rendered.

Una SPA che ha elementi di server side rendering viene chiamata *universal unified SPA*.

Capitolo 3

Gestione delle risorse e design

Riuscire ad ottimizzare il caricamento delle risorse, come fogli di stile CSS, e mantenere un design coerente in tutti i sottoprogetti sono due grandi sfide della programmazione microfrontend, ne vediamo quindi delle strategie.

3.1 Caricamento delle risorse

Possono essere presenti nei progetti dei fragments dei file CSS e javascript, necessari al giusto design e funzionamento logico del componente. Esistono diverse politiche atte a caricare le risorse necessarie a tutti i microfronend:

3.1.1 Referencing con soluzioni client-rendered

Il modo più semplice consiste nel caricare le risorse necessarie direttamente dalla pagina ospitante. I team quindi dovranno fornire gli indirizzi dei file richiesti. Per far risparmiare traffico agli utenti, di solito viene impostata una regola che fa mantenere ai client le risorse in cache per un anno, in questo modo non devono essere scaricate tutte le volte.

Ma quando viene distribuito una nuova versione di file, i client hanno in cache delle versioni obsolete, e devono riscaricarle. Per far si che avvenga, si appone al nome dei file il *fingerprint*, un codice alfanumerico che deriva dal checksum del file. Questa tecnica, chiamata **cache busting**, forza il browser dell'utente a riscaricare i file. Per far si che i team possano distribuire autonomamente i file, senza comunicare ogni volta il nuovo nome del fingerprint alla pagina ospitante, si include in questa dei nomi generici, che verranno poi reindirizzati dal webserver all'ultimo file aggiornato.

3.1.2 Referencing con soluzioni server-rendered

Se il progetto utilizza già il rendering lato server, è possibile includere le risorse nel codice del fragment, in modo che quando questo viene sostituito dalla direttiva SSI dal webserver, va ad includere alla pagina HTML anche i file necessari. Se però il fragment viene incluso nella pagina più volte, anche le risorse vengono scaricate ripetutamente. Inoltre i file javascript vengono eseguiti tante volte quanto vengono inclusi nella pagina, il che va ad aumentare il carico sulla CPU del client e potrebbe creare errori di esecuzione.

3.1.3 I vantaggi di HTTP/2

Il nuovo protocollo HTTP/2, basato sul progetto SPDY sviluppato da Google nel 2009, estende il precedente HTTP/1 aumentando la velocità di caricamento delle pagine, grazie alla riduzione dell' header HTTP.

Il raggruppamento di più risorse in un'unica risposta dal server è stata un'ottimizzazione fondamentale per HTTP/1.x, in cui il parallelismo limitato e l'overhead elevato del protocollo in genere superavano tutte le altre preoccupazioni. Tuttavia, con HTTP/2, il multiplexing non è più un problema e la compressione dell'intestazione riduce drasticamente il sovraccarico dei me-

tadati di ogni richiesta HTTP. [9] Di conseguenza con l'avvento di HTTP/2, il concatenamento di richieste e la divisione delle risorse è un aspetto che non va più a invalidare le prestazioni, il che significa che ora possiamo ottimizzare le nostre applicazioni fornendo risorse più granulari. Ecco alcuni vantaggi pratici derivanti dalla più alta granuralità delle risorse con HTTP/2:

- I browser andranno a scaricare **parti di codice più piccole**, evitando di farlo per quelle che non sono state modificate e che risiedono in cache. Grazie a questo i team possono operare con una distribuzione continua, anche più volte al giorno.
- Si avranno **task di codice javascript più brevi**, che favoriranno la velocità del feedback con l'input dell'utente.
- Si potrà facilmente far caricare all'utente le risorse provenienti dai vari team necessarie esclusivamente a visualizzare la pagina corrente

3.2 Design system

Un progetto composto da microfrontend ha come priorità l'autonomia, ma tutti i pezzi del progetto devono apparire all'utente come parte di una cosa sola, e devono avere delle caratteristiche visive in comune. Insomma, i team hanno bisogno di un modo per condividere uno schema di design e stilizzazione.

Un design system contiene parti di design, come font, colori, icone, componenti di interfaccia riusabili, come bottoni e form, e anche un ben definito insieme di regole.

Un design system ben fatto porta ai seguenti benefici:

- **Consistenza:** i microfrontend appaiono familiari tra di loro agli occhi dell'utente
- **Linguaggio condiviso:** i team devono parlare la stessa lingua, questo porta a limitare misunderstanding.
- **Velocità di sviluppo:** avere componenti di design già pronti velocizza molto lo sviluppo di nuove features
- **Scalabilità**: avendo un buon design system condiviso, altri team nuovi possono adottarlo, e velocizzare lo sviluppo, niente è ridondante.

Ovviamente creare questo porta ad un impiego importante di risorse, ma va visto come un investimento a medio-lungo termine.

Non obbligatoriamente per un nuovo progetto si deve creare un design system proprietario, se il prodotto che stiamo costruendo è per uso interno di un'azienda, o non vogliamo investire troppo su una personalizzazione del design è molto meglio adottare progetti già realizzati, come Bootstrap o Material UI.

3.2.1 sviluppo di un design system

Ci sono due modi per sviluppare un design system, il modello centrale e quello federated:

Nel modello **centrale** un team di design, composto da UX designer, è impiegato esclusivamente per il design system. Gli altri team sono degli utilizzatori, che adottano i prodotti di quest'ultimo.

I team di produzione richiedono nuovi componenti grafici e il team di design lo crea. Possono crearsi dei rallentamenti, perche un unico team di design dovrà servire diversi team di produzione.

Nel modello **federated** invece gli UX designer vengono inseriti nei vari team di produzione, e realizzeranno le componenti grafiche richieste dai loro colleghi. Ovviamente tutti i nuovi elementi creati verranno messi a disposizione a tutti gli altri team.

Spesso questi due modi di sviluppare il design system vengono usati insieme quando si inizia un nuovo progetto dove all'interno si crea un apposito design system: all'inizio il lavoro da fare è tanto, e quindi si parte con l'approccio centrale, per poi via via migrare a quello federated.

Capitolo 4

Caso di studio: Leonardo SPA

4.1 Introduzione

Adesso, viste le tecniche per la realizzazione di un'architettura microfrontend, ne vediamo un'applicazione, realizzata in collaborazione con Leonardo, una realtà industriale globale nell'alta tecnologia, in settori come aerospazio, difesa e sicurezza.

4.1.1 La sala di controllo: la piattaforma Leonardo X2030

La sala di controllo è il luogo per la gestione degli eventi di pubblica sicurezza, delle indagini delle operazioni di polizia: rappresenta il punto logico di convergenza per i sensori di campo, ci sono presenti applicazioni che elaborano dati caratterizzati da diverso formato e sistemi dedicati alla pianificazione e gestione delle risorse.

Scenario

La complessità è ciò che gli operatori della sicurezza devono affrontare nella quotidianità. Il contesto operativo è caratterizzato dalla disponibilità

di molti flussi di dati eterogenei che devono essere elaborati, sintetizzati, contestualizzati in modo tempestivo per poi essere tradotti in informazioni fruibili, che consentano agli operatori di prendere le giuste decisioni in tempi brevi. Non riuscire a fare ciò, può segnare la differenza tra il successo e fallimento per una missione che, in contesti critici (*mission-critical*) come la sicurezza o la difesa pubblica, possono anche comportare rischi per la vita umana.

Le nuove applicazioni di comando e controllo hanno bisogno di padroneggiare l'enorme carico di informazioni al fine di:

- estrarre informazioni rilevanti da un'enorme quantità di dati eterogenei, anche facendo uso di esperienze passate elaborate da intelligenza artificiale
- convertire tali informazioni in indicazioni utili per la gestione delle operazioni
- essere in grado di diffonderle sul campo in modo sicuro ed efficace.

Il ruolo chiave della tecnologia:

Nelle applicazioni *mission-critical*, l'affidabilità della soluzione e il pieno rispetto delle procedure operative esistenti hanno la precedenza sulla disponibilità di eventuali funzioni sofisticate. Per questo generalmente viene adottato un approccio conservatore riguardo l'uso di tecnologie innovative ed emergenti.

L'evoluzione delle telecomunicazioni e dell'informatica hanno portato alla disponibilità di tecnologie capaci di superare alcuni problemi bloccanti, rendendo la progettazione di un sistema di nuova generazione finalmente possibile.

Tali tecnologie sono essenzialmente:

- **4G e 5G:** consentono di costruire un sistema di comunicazione scalabile, adattabile dinamicamente alle prestazioni richieste e di supportare l'integrazione con reti di generazione precedente.
- **Big Data:** IoT, videosorveglianza con telecamere multi megapixel, grandi database e accessi agli archivi storici, sono causa di un incremento enorme di dati che devono essere consultati ed elaborati, e che richiedono tecnologie adeguate per una corretta gestione.
- **Intelligenza artificiale:** L'aumento della quantità di dati nella sala di controllo richiede l'automazione per scaricare l'operatore da compiti di sorveglianza ingombranti,come l'analisi video,e per fornire supporto decisionale tramite simulazioni e proiezioni basate su esperienze passate simili.
- **Sicurezza informatica:** Le nuove sale di controllo sono sempre più connesse e quindi esposte a minacce che devono essere contrastate a partire dal design di quest'ultime fino al monitoraggio in tempo reale, utilizzando una gestione proattiva.
- **Cloud Computing:** Sono necessari nuovi modelli di elaborazione per far fronte un carico di lavoro distribuito e dinamicamente variabile. Il cloud computing consente l' ottimizzazione fisica delle risorse allocando la potenza di calcolo quando e dove richiesto. E' preferibile adottare una soluzione privata per questioni di sicurezza.

4.1.2 La soluzione di Leonardo: la piattaforma X2030

Sfruttando le numerose esperienze maturate negli anni, Leonardo ha disegnato **X2030**, una nuova generazione di prodotto per il comando e il controllo finalizzato a equipaggiare i clienti con soluzioni versatili e scalabili, seguendo il nuovo paradigma basato sul modello operativo **Data and Experience-Centric**.

X2030 implementa una architettura federale ,che può estendersi su più siti ed essere accessibile in modo sicuro da diverse organizzazioni, in grado di:

- Integrare tutti i sistemi e le applicazioni esistenti nella sala di comando aggiungendo nuove funzioni e servizi se necessario.
- Raccogliere informazioni da sorgenti di dati strutturate (banche dati, archivi) e non strutturate (video, sensori, media).
- Fornire agli operatori tutte le informazioni utili per un efficace monitoraggio, sorveglianza e gestione di un evento, fornendo **suggerimenti operativi** generati da attività di background di analisi (estrazione di dati, dati correlati, analisi video).
- Ottimizzare la logistica e la gestione delle risorse interagendo con tutti i database, strutturati o non strutturati.
- Supportare l'analisi delle indagini per i fenomeni monitorati analizzando grandi quantità di dati, fornendo simulazioni.

Architettura di X2030

X2030 è basato su un'architettura a micro servizi, supporta una distribuzione cloud e ha un accesso web. Dal punto di vista dell'architettura logica, è organizzato nei seguenti livelli:

- **Integration Layer:** include tutti i sottosistemi e sensori che acquisiscono informazioni direttamente dal campo. A tale livello, le informazioni ottenute potrebbero essere già oggetto di una prima elaborazione secondo la business-logic del suo dominio.

- **Core Layer:** è il livello logico centrale, nonchè il kernel di X-2030. Qui, dati ed eventi in arrivo dall'Integration Layer vengono raccolti attraverso l' infrastruttura a microservizi e messi a disposizione dei vari motori di elaborazione.

- **Presentation Layer:** questo livello si basa su un HMI(Interfaccia uomo-macchina) innovativa, costruita per semplificare le informazioni visualizzate dall'operatore.Il livello, basato su **GIS** (Geographic Information System) permette di essere utilizzato da ingresso per vari sottolivelli fornendo diversi servizi offerti dal sistema

La versatilità di X2030 e la possibilità di integrazione con applicazioni esistenti o specifiche danno il possibilità di progettare e distribuire soluzioni di controllo e comando in diversi domini applicativi come:

- **Pubblica sicurezza e forze di polizia**

- **Operazioni di difesa**

- **Governo della città**

Casi d'uso

Supponendo di voler gestire eventi relativi all'esecuzione dell'ordine pubblico, vediamo le caratteristiche della piattaforma:

- Organizzazione eventi: ispezione dell'aera su mappa 3D supportato da realtà virtuale
- Identificazione delle risorse disponibili e posizione sulla mappa comprese tutte le informazioni disponibili (abilità, attrezzatura,...).
- Tracciamento continuo delle risorse rilevanti sulla mappa.
- Suggerimento automatico del percorso migliore da parte dell'assistente virtuale.
- Istruzioni/divulgazione di informazioni sul campo (incluso immagini, percorso migliore, ecc.) tramite **radio broadband**.
- Visualizzazione e streaming di riprese da droni (se presenti) su smartphone o tablet per operatori sul campo.
- Estrazione di dati sintetici e inoltro a sistemi federati (incluse sale di controllo aggiuntive o altri responsabili) al fine di condividere le informazioni e migliorare il coordinamento
- Monitoraggio dei social media per eventuali informazioni rilevanti
- Analisi video utilizzata per rilevare potenziali eventi interessanti

La piattaforma aiuta molto gli addetti in casi di gestione di emergenze, infatti un eventuale evento viene localizzato sulla mappa al momento dell'occorrenza e assegnato ad un operatore. Tutte le informazioni rilevanti

disponibili possono essere facilmente visualizzate. Il sistema può interrogare i database disponibili cercando informazioni relative ai soggetti coinvolti (persone, telefono chiamate effettuate, armi) e visualizza i risultati in maniera chiara. A questo punto l'operatore della sala di controllo può ordinare l'intervento alla risorsa più adeguata a gestire la situazione. "Trascina e scegli" semplifica operazione di assegnazione che può essere completata via radio conversazione con gli agenti coinvolti, semplicemente effettuando azioni "drag and drop". Se necessario il sistema può automaticamente impostare una chiamata di gruppo dinamica che coinvolga le risorse scelte sul campo e sala di controllo. Tramite la radio broadband è possibile inoltre inviare alle risorse sul campo informazioni, come immagini o percorsi migliori. Qualora si rendessero disponibili risorse più adeguate durante gestione dell'evento, il sistema sarà in grado di proporre una modifica di squadra.

Inoltre tutte le attività relative all'evento vengono memorizzate e archiviate. Alla chiusura dell'evento, tutte le informazioni sull'evento sono disponibili in database strutturato per interrogazioni forensi o amministrative e indagini.

4.2 Progettazione del componente

Presentata la piattaforma X2030, andremo a focalizzarci sul progetto specifico realizzato. Precedentemente abbiamo spiegato che la dashboard aiuta l'operatore nei momenti di emergenza a gestire le risorse sul campo, assegnarle alla missione e comunicare con loro tramite gli apparati radio per condividere obiettivi e informazioni sull'accaduto. Un obiettivo della piattaforma è anche quello di seguire l'operatore il più possibile in queste delicate procedure, che sono scandite da precisi passaggi descritti in appositi protocolli operativi. E' prassi che l'operatore segua tutti i passaggi nel giusto ordine e che metta a verbale l'esecuzione di ognuno di questi.

Si vuole quindi realizzare un componente che appare a schermo ogni volta che l'operatore deve gestire un evento, che visualizzi le varie operazioni suggerite, e che tenga conto di quelle effettuate o meno. Il componente dovrà anche suggerire all'operatore la serie di sotto-operazioni (dette **routines**) necessarie al compimento di ognuna di queste. Un database remoto conterrà la lista dei vari eventi che potrebbero accadere, comprese le operazioni suggerite (chiamate **suggested-operations**) da effettuare, e relative routines. Ogni evento è identificato univocamente da due ID, chiamati *firstLevelId* e *secondLevelId*, dove il secondo specializza il primo(e.g. una rissa con armi bianche ha come ID: firstLevelId="Liti" secondLevelId="Aggressione con coltello").

4.2.1 Scelta dell'architettura microfrontend

La piattaforma web comprende vari strumenti, uniti da un unica dashboard che ha come elemento centrale una mappa satellitare. Gli strumenti sono stati sviluppati da diversi team dell'azienda, e vengono continuamente aggiornati tramite distribuzioni. Si è deciso di affrontare il progetto con un

approccio microfrontend, e la scelta dell'architettura che lo realizzi è caduta su una composizione di web-components effettuata lato client, realizzando così una Single Page Application unificata, ecco perché:

- E' necessario un feedback immediato ad ogni azione compiuta dall'operatore
- L'applicazione web viene usata da organizzazioni interne alle Forze dell'Ordine e non è pubblica, quindi non serve alcuna indicizzazione sui motori di ricerca
- Essendo il progetto nuovo, non sono presenti parti legacy, ovvero applicazioni non modificabili
- E' richiesta una soft navigation in tutta la web app
- Non è fondamentale il tempo di caricamento iniziale della pagina

Il microfrontend verrà quindi incapsulato in un Web Component, che avrà gli attibuti *firstLevel*, *secondLevel* e *eventId*. E' inoltre possibile passare al componente l'attributo opzionale *endpoint*, per indicare un differente URL per l'interrogazione al database.

4.2.2 Class diagram

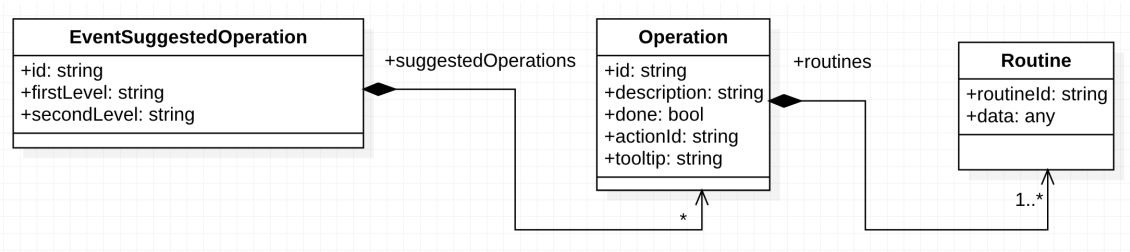

Figura 4.1: Class Diagram

Per ogni evento previsto dal sistema è associata l'entità **EventSuggestedOperation**, che contiene una collezione di **Operation**, queste consistono nelle operazioni che dovrà effettuare l'operatore per completare la risposta all'evento accaduto. l'attributo *tooltip* conterrà un eventuale suggerimento che verrà visualizzato al passare del mouse. *actionId* indica invece il tipo di operazione da effettuare per concludere tale operazione. Il booleano *done* invece indica se suddetta operazione è stata effettuata o meno. Ogni operazione ha una collezione di **routines**, che servono a automatizzare il compimento di quest'ultima, facilitando l'esperienza utente. Ogni routine ha una payload contenuto nell'attributo *data*.

4.2.3 Scelta del sistema di comunicazione

Il componente, chiamato **SuggestedOperationWidget**, ha bisogno di comunicare con la pagina ospitante con queste modalità:

- Se l'operatore conclude un operazione, il **SuggestedOperationWidget** deve saperlo, e lo sfondo dell'operazione appena effettuata deve risultare verde

- quando si sceglie di eseguire un operazione dal widget, questo deve inviare alla pagina ospitante le routines collegate

Per effettuare queste comunicazioni è stato adottato **Broadcast Channel API**, il quale è stato già introdotto precendetemente.

Si vanno quindi a definire due canali di comunicazione:

- **po.actions**: per comunicare dalla pagina ospitante al componente le azioni che sono state effettuate dall'operatore.
- **sow.routines**: permette al componente di inviare le routines necessarie al compimento dell'operazione alla pagina ospitante

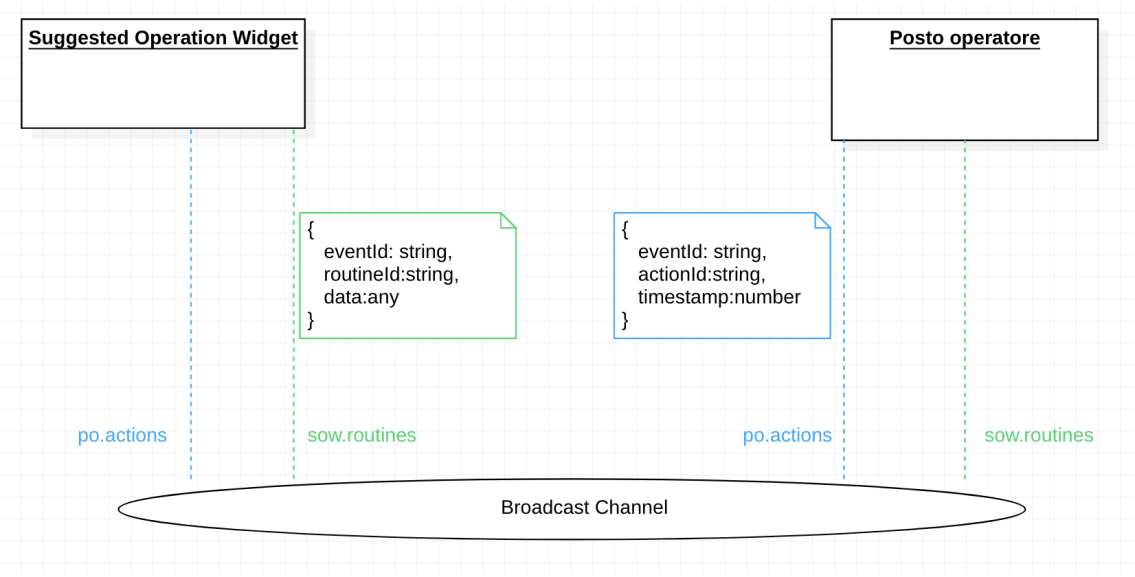

Figura 4.2: Schema del broadcast channel

4.2.4 Angular

Si è scelto di creare il componente con Angular, in quanto viene già usato per il resto del progetto di Leonardo.

Angular è un framework di progettazione per applicazioni e una piattaforma di sviluppo per la creazione di SPA efficienti e sofisticate. [7]

Si è creato quindi un Component Angular, il quale consiste in:

- Un template HTML che definisce l'aspetto
- Una classe Typescript che definisce la logica
- Un selettore per definire come il componente è usato nel template
- un foglio di stile CSS

Creare un fragment microfrontend con Angular è molto semplice, perchè il framework mette a disposizione il concetto di **Angular Elements**, che consiste nell'incapsulare un componente in un Web Component.

Il package *@angular/elements* esporta il metodo *createCustomElement()*, che mette a disposizione un ponte tra l'interfaccia del componente Angular e le funzionalità di riconoscimento dei cambiamenti presenti nelle API javascript.

L'obiettivo è quello di convertire anche gli altri componenti della applicazione web in Angular Elements, tutti provenienti da app diverse, sviluppate da team diversi. Un problema che è stato riscontrato durante il tirocinio è che di base non possono coesistere più app Angular nella solita pagina. Questo è dovuto ad un limite di compilazione: Angular utilizza Webpack per creare il bundle dell'app, creando un oggetto **window.webpackJsonp**. Il nome dell'oggetto non è modificabile, quindi eventuali altre app che si

eseguono nella pagina causeranno un conflitto di namespacing, causando il malfunzionamento dell'app. [5]

Per rinominare l'app abbiamo scelto la libreria **Npx Build Plus**, definendo un semplice comando per chiedere a Webpack di dare un nome univoco:

module.exports = output: library: 'suggOpWidget' ;

E' stato realizzato inoltre una repository pubblica su Github, che fungerà da esempio per i prossimi componenti. Questa contiene un Angular Element già correttamente configurata e pronta per essere compilata e per coesistere con più fragment in un'unica pagina. [3]

4.3 Sviluppo

Il codice del componente comprende principalmente i seguenti files:

- Le interfacce:

  eventOperation.model.ts

  suggestedOperation.model.ts

  routine.model.ts

- Il componente:

  suggested-operation-widget.component.ts

  suggested-operation-widget.component.html

  suggested-operation-widget.component.scss

- I servizi:

  events.service.ts

La logica del componente risiede principalmente nel servizio **events**. Per limitare le connessioni al server si effettua una cache delle operazioni suggerite scaricate dal database. Quando il componente si avvia, controlla se è presente una cache e se è aggiornata, in caso contrario si effettua una richiesta al server. La cache trattiene anche gli storici delle operazioni già effettuate, marcate come "done", per mantenerle anche nelle successive sessioni del browser.

4.3.1 Sequence diagram

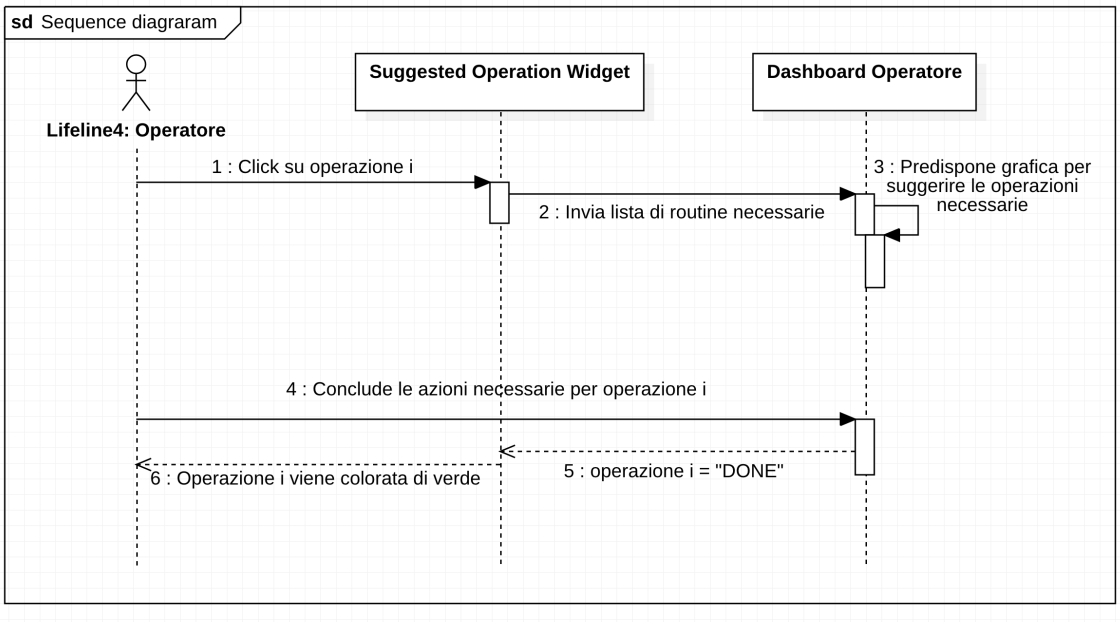

Figura 4.3: Sequence diagram che mostra l'interazione dell'operatore con il componente

4.3.2 User Design

Il componente è attualmente pensato per apparire nel lato destro dell'interfaccia. Ogni riga contiene un'operazione, il pulsante serve per avviare la procedura di suggerimento, nel caso l'operatore non ricordasse perfettamente

le varie azioni per terminarla. E' stato utilizzato un desing system già esistente, chiamato Nebula [1]. In questo modo i team non avranno bisogno di produrre i componenti di design, ma si limiteranno ad adottare quelli offerti dal pacchetto utilizzato.

Capitolo 5

Conclusioni

Come lo sono i microservizi per il lato backend, i microfrontend sono un efficace soluzione per dividere i compiti tra team omogenei e realizzare un progetto web modulare e reattivo ai cambiamenti. Ogni team produce delle applicazioni indipendenti, che possono essere estratte dal contesto e riproposte in altri progetti che richiedono funzionalità analoghe. Numerose aziende hanno migrato verso questa tecnologia, una di queste è la tedesca Zalando, leader nel settore e-commerce, che con il progetto Mosaic ha rilasciato pubblicamente una raccolta di servizi e librerie per aiutare proprietari di grandi siti web a migrare dall'approccio monolitico, contribuendo la diffusione di tale pratica rendendola più accessibile [12], non si deve trascurare il fatto che realizzare un progetto web con questa architettura è molto più complesso di farlo con l'approccio monolitico, o almeno inizialmente.

Una parola ricorrente in questo elaborato è **composizione**: ovvero il far convivere nella solita finestra del browser più fragments provenienti da team diversi. La composizione può essere effettuata prima di far arrivare la pagina al client, oppure dopo, delegando l'operazione al browser.

Molto spesso si è parlato di **autonomia**, che non vuol dire isolamen-

to: i team lavorano per lo più internamente, ma a seconda delle tecnologie utilizzate, devono presentare agli altri un *contratto*, delle informazioni che devono essere condivise per mettere in fase l'intero progetto. Queste informazioni però devono restare minime, perche quanto più il contratto è pieno di "clausole" da rispettare, tanto più il progetto si complica.

La piattaforma Leonardo X2030 è l'esempio perfetto per la messa in opera di un'architettura microfrontend: contiene molti strumenti, ha un'interfaccia modulare personalizzabile al massimo, è aggioranta continuamente per essere compatibile con le nuove tecnologie, come rilievi con droni o gestione di big data sfruttando l'intelligenza artificiale. Tutte queste funzionalità eterogenee devono comunicare tra di loro, e devono essere racchiuse in un interfaccia che le renda utilizzabili contemporaneamente al personale di diversi enti.

Bibliografia

[1] Akveo. Nebula. https://akveo.github.io/nebular/, 2021. [Online; accessed 8-November-2021].

[2] ANSA. Istat, calano vendite ma cresce e-commerce. https://www.ansa.it/sito/notizie/economia/2021/01/12/commercio-istat-calo-vendite-del-69-a-novembre.-bankitalia-covid-pesa-sulle-pmi_a2343dad-a259-48e8-85df-7ebf93dfac4f.html, 2021. [Online; accessed 14-November-2021].

[3] Matteo Bavecchi. angular-element-example. https://github.com/MatteoBavecchi/angular-elements-example, 2021. [Online; accessed 4-November-2021].

[4] Samantha Subin CNBC. Facebook's outage . https://www.cnbc.com/2021/10/09/facebook-instagram-outage-hurt-creators-small-businesses.html, 2021. [Online; accessed 14-November-2021].

[5] Nicholas Favero. Multiple Angular Apps on a single page. https://medium.com/swlh/multiple-angular-apps-on-a-single-page-9f49bc863177, 2019. [Online; accessed 4-November-2021].

[6] Michael Geers. *Microfrontends in action.* Manning, 2020.

[7] Google. Angular Docs. `https://angular.io/docs`, 2021. [Online; accessed 4-November-2021].

[8] Google. Server-side rendering (SSR) with Angular Universal. `https://angular.io/guide/universal`, 2021. [Online; accessed 15-October-2021].

[9] Ilya Grigorik. High Performance Browser Networking - Optimizing Application Delivery. `https://hpbn.co/optimizing-application-delivery/#minimize-concatenation-and-image-spriting`, 2013. [Online; accessed 25-October-2021].

[10] Mozilla. DOM. `https://www.w3schools.com/js/js_htmldom.asp`, 2021. [Online; accessed 10-October-2021].

[11] Mozilla. Web Components. `https://developer.mozilla.org/en-US/docs/Web/Web_Components`, 2021. [Online; accessed 10-October-2021].

[12] Zalando. Mosaic. `https://www.mosaic9.org/`, 2021. [Online; accessed 9-November-2021].