Sommario

[CAPITOLO 1 3](#_Toc70369814)

[Da dove nasce la necessità di un micro front-end? 3](#_Toc70369815)

[Dallo standard al caos 3](#_Toc70369816)

[CHALLENGE 1 4](#_Toc70369817)

[Cos’è un micro front end? 5](#_Toc70369818)

[Struttura del sistema e organizzazione dei team 6](#_Toc70369819)

[Team Mission 6](#_Toc70369820)

[Team cross funzionali 7](#_Toc70369821)

[Il Frontend 8](#_Toc70369822)

[Page ownership 8](#_Toc70369823)

[Fragments 9](#_Toc70369824)

[Integrazione del front end 10](#_Toc70369825)

[Routing e navigazione tra pagine 10](#_Toc70369826)

[Composizione 11](#_Toc70369827)

[Comunicazione 12](#_Toc70369828)

[Shared topics 13](#_Toc70369829)

[Web Performance 13](#_Toc70369830)

[Design System 13](#_Toc70369831)

[Sharing Knowledge 13](#_Toc70369832)

[CHALLENGE 2 13](#_Toc70369833)

[PARTE 2 14](#_Toc70369834)

[Perché scegliere un micro frontend. 14](#_Toc70369835)

[Velocità 14](#_Toc70369836)

[Micro frontend vs frontend monolitico 15](#_Toc70369837)

[Velocità di aggiornamento tecnologico 15](#_Toc70369838)

[Sistemy legacy 16](#_Toc70369839)

[Sistema decisionale isolato 16](#_Toc70369840)

[Indipendenza 17](#_Toc70369841)

[Sovraccarico di risorse? 17](#_Toc70369842)

[Shared Nothing 18](#_Toc70369843)

[Gli svantaggi di un micro frontend 19](#_Toc70369844)

[Ridondanza 19](#_Toc70369845)

[Consistenza 19](#_Toc70369846)

[Eterogeneità 20](#_Toc70369847)

[Quando adottare un micro frontend 21](#_Toc70369848)

[Ottimo per progetti medio-grandi 21](#_Toc70369849)

[Applicare micro frontend al web funziona meglio 21](#_Toc70369850)

[Produttività contro spese generali - costi (setup e complessità organizzativa) 21](#_Toc70369851)

[Dove non sono adatti i micro frontend? 22](#_Toc70369852)

[Chi usa i micro front end? 22](#_Toc70369853)

[Concetti conclusivi essenziali 22](#_Toc70369854)

[PARTE 3 23](#_Toc70369855)

[IFRAME 24](#_Toc70369856)

[AJAX(NGINX Server side routing/composition) 25](#_Toc70369857)

[Quando utilizzare questa tecnica 25](#_Toc70369858)

[Server side composition 25](#_Toc70369859)

[Come lavora NGINX SSI 25](#_Toc70369860)

[Aspetti negativi dell’utilizzo dell’SSI 26](#_Toc70369861)

[AJAX(Docker Server side composition) 26](#_Toc70369862)

[WEB COMPONENTS(angular elements) 27](#_Toc70369863)

[Web component e custom elements 27](#_Toc70369864)

[Ciclo di vita di un custom elements 27](#_Toc70369865)

[Benefici dei web components 28](#_Toc70369866)

[Svantaggi dei web component 28](#_Toc70369867)

[SINGLE-SPA (Client-side-composition) 29](#_Toc70369868)

[RESOCONTO DI SINGLE-SPA 30](#_Toc70369869)

[Error-handling 30](#_Toc70369870)

[Memory management 30](#_Toc70369871)

[Single point of failure 30](#_Toc70369872)

[Shell management 31](#_Toc70369873)

[Comunicazione 31](#_Toc70369874)

[Quando utilizzarla 31](#_Toc70369875)

[RESOCONTO SCHEMA FINALE 32](#_Toc70369876)

[Casi di successo 32](#_Toc70369877)

[DAZN 32](#_Toc70369878)

[IKEA 33](#_Toc70369879)

[Upwork 33](#_Toc70369880)

[Spotify 33](#_Toc70369881)

[HelloFresh 33](#_Toc70369882)

[Zalando 33](#_Toc70369883)

# CAPITOLO 1

## Da dove nasce la necessità di un micro front-end?

Lo sviluppo software di un frontend negli anni si è evoluto. Le applicazioni web che stiamo realizzando oggi devono essere caricate rapidamente, essere eseguite su un'ampia gamma di dispositivi e dovrebbero reagire rapidamente alle interazioni degli utenti. Per molte aziende, il frontend web è la principale fonte di interazione per i propri utenti. Quindi è naturale dedicare molta attenzione nel suo sviluppo.

In caso di applicazioni di piccole dimensioni lo sviluppo è semplice(in teoria) e nella maggioranza dei casi intuitivo. Tuttavia quando una singola applicazione comincia ad essere di dimensioni medio/grandi, oppure vi sono più applicazioni da gestire nella tua area applicativa, un teams singolo potrebbe essere sopraffatto dalle attività . Qui che entra in gioco l'architettura del micro frontend. Tramite questa architettura è possibile suddividere una singola applicazione in più applicazioni indipendenti tra di loro, dove ogni team può lavorare(teoricamente) in maniera indipendente l’una dall’altra

### Dallo standard al caos

In applicazioni di piccole dimensioni, è facile suddividere il lavoro, soprattutto ogni sviluppatore conosce più o meno tutte le funzionalità di un progetto di piccole dimensioni. Ogni funzionalità può essere sviluppata facilmente e velocemente, in quanto la conoscenza richiesta per un progetto di piccole dimensioni è limitata e circoscritta. In questi casi quante volte ti è capitato di discutere di soluzioni tecniche e funzionali davanti alla macchinetta del caffè?

ATTENZIONE:

Questo paradigma subisce un importante rallentamento quando una applicazione comincia ad avere una struttura molto più complessa. Quante volte vi è capitato di dover apportare una modifica ad una parte di un sistema ed avere timore che questa modifica rompa una parte di un'altra applicazione?

Le riunioni davanti alla macchinetta del caffè per decidere di eventuali soluzioni diventano riunioni ufficiali all’interno del team stesso. Ogni decisione all’interno del team diventa più complessa, allungando di conseguenza i tempi di sviluppo.

Ecco allora che vengono fuori i ‘SILOS’ di conoscenza. Da un grande team, cominciano a crearsi piccoli team che conoscono piccole parti funzionali e tecniche dell’intero applicativo.

La complessità continua ad aumentare, ed il passaggio successivo diventa suddividere il team per tecnologia. Vengono cosi introdotti dei livelli orizzontali con team dedicati al front-end e team dedicati al backend.

In questo caso i micro front end offrono un approccio alternativo. Invece che avere una suddivisione orizzontale delle responsabilità, avremo una suddivisione verticale. Di conseguenza ogni team avrà la conoscenza tecnica e funzionale di una struttura che parte dal database fino all’interfaccia utente. Questo approccio è utilizzato soprattuto in architetture dove sono previste architetture a micro-servizi, con la sostanziale differenza , che adesso un singolo servizio include anche la sua interfaccia utente.

Quali sono i vantaggi di queste architettura?

* **Ottimizza lo sviluppo di funzionalità**: un team include tutte le competenze necessarie per sviluppare una funzionalità. Non è richiesto alcun coordinamento tra i team separati di frontend e backend.
* **Semplifica gli aggiornamenti del frontend:** ogni team possiede il proprio stack completo dal frontend al database. I team possono decidere di aggiornare o cambiare la loro tecnologia di frontend indipendentemente dagli altri team.
* **Maggiore attenzione al cliente**: ogni team fornisce le proprie funzionalità direttamente al cliente. Non esistono team API puri o team operativi.

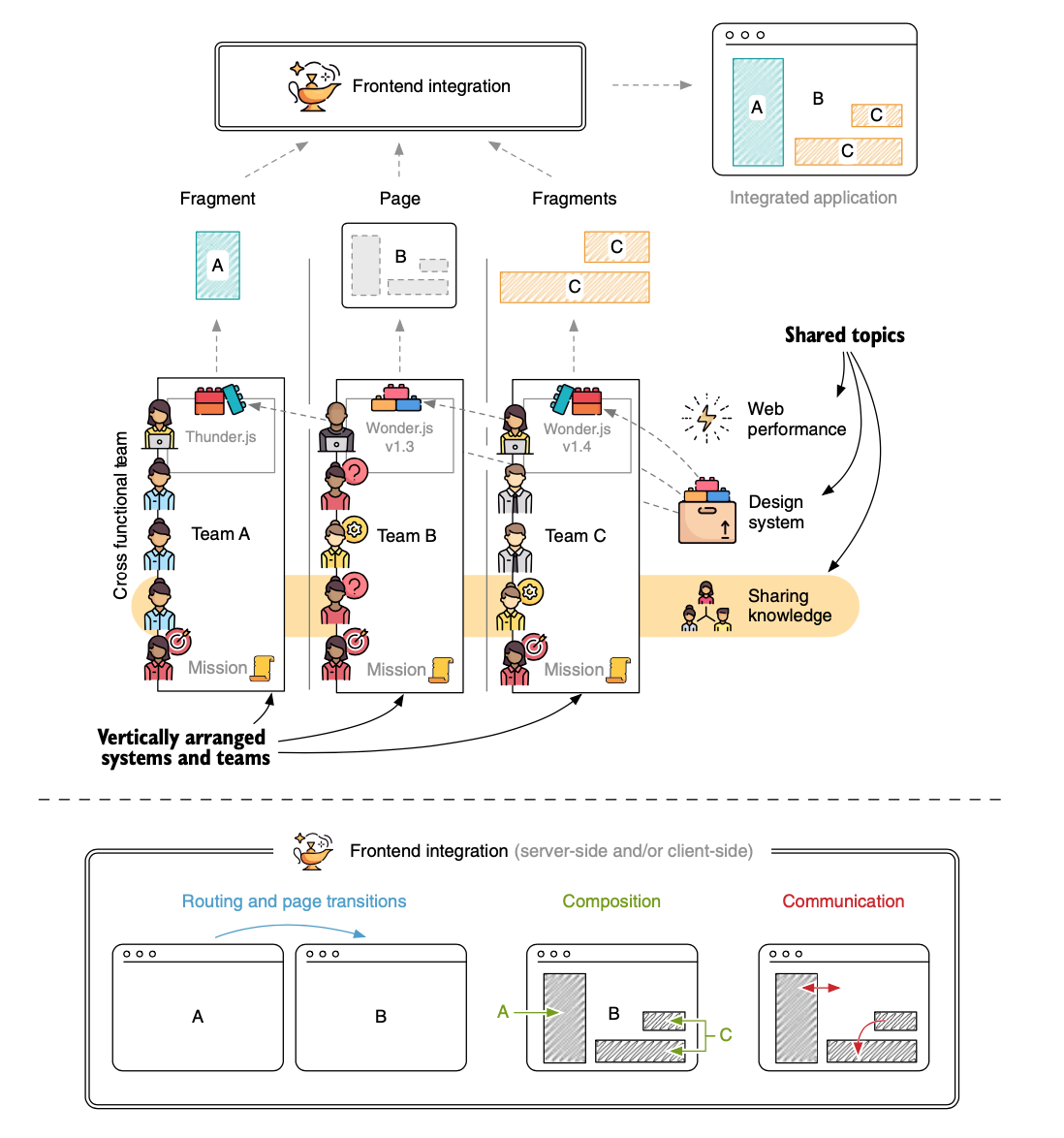
## CHALLENGE 1

Descrivere ed individuare le problematiche di un progetto a cui si è lavorato. Inoltre descrivere la soluzione che si è adottata per risolvere eventuali problematiche. TEMPO 10 minuti

# Cos’è un micro front end?

Di seguito uno schema riassuntivo di tutte le parti fondamentali che sono coinvolte quando si implementa un micro front end.

Da questa immagine è possibile notare come i micro frontend non sono una tecnologia concreta, piuttosto consistono in un approccio organizzativo e architetturale alternativo. Ecco perché in questo grafico vi sono molti elementi diversi, come la struttura del team, le tecniche di integrazione e altri argomenti correlati.



## Struttura del sistema e organizzazione dei team

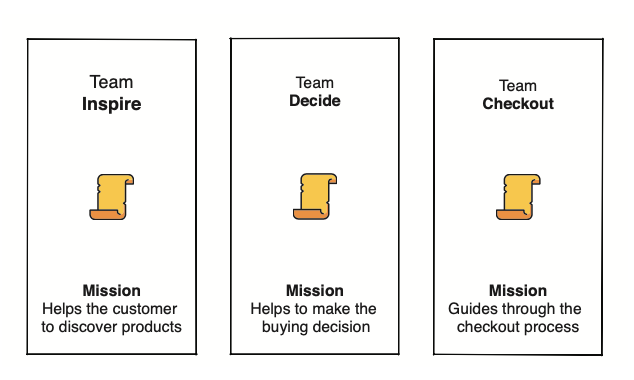
Le tre caselle con i team A, B e C mostrano i sistemi software disposti verticalmente. Ogni teams ha la responsabilità di un unico sistema, che costituisce il nucleo strutturale di questo tipo di architettura. Ogni sistema è autonomo, il che significa che può funzionare anche quando i sistemi vicini sono inattivi.

Essere funzionalmente indipendenti significa anche che ogni sistema sistema avrà un database dedicato, in maniera tale da essere indipendente dagli altri sistemi.

Inoltre, non si basa su chiamate sincrone ad altri sistemi per rispondere a una richiesta(altrimenti vi sarebbe una relazione di dipendenza da altri sistemi, ed è quello che vuole essere evitato). Notare come questa struttura riflette il mondo di lavorare dei micro servizi, traslando sui front end un paradigma già esistente nei backend.

### Team Mission

### 



Ogni team avrà la sua area di competenza, questo naturalmente fornisce valore aggiunto per il cliente, essendo team funzionalmente suddivisi in aree ben specifiche. Prendiamo come esempio un progetto di un e-commerce con 3 team.

Ogni team lavora su una parte differente del nostro e-commerce, dichiarando prima dello start del progetto la sua mission , in maniera tale da chiarire sin da subito quali siano le responsabilità legate al proprio team.

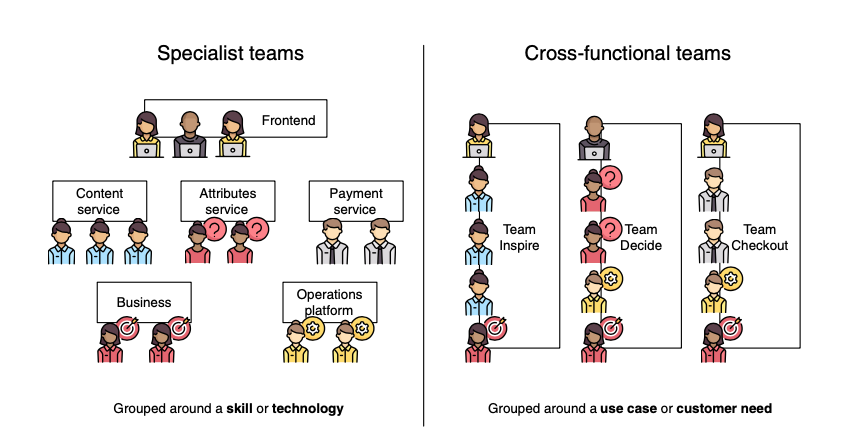
Ogni squadra ha un nome descrittivo e una chiara missione incentrata sull'utente. In questo esempio avremo quindi:

* **Team Inspire's missione** -> come suggerisce il nome, è ispirare il cliente che naviga e presentare prodotti che potrebbero essere di interesse.
* **Team decide** -> aiuta a prendere una decisione di acquisto informata fornendo eccellenti immagini dei prodotti, un elenco di specifiche pertinenti, strumenti di confronto e recensioni dei clienti.
* **Team Checkout** -> Guida un cliente attraverso il checkout.

Una missione chiara è vitale per il team. Fornisce focus ed è la base per dividere il sistema software.

### Team cross funzionali

La differenza più significativa tra micro frontend e altre architetture è la struttura del team.



In questa immagine è rappresentata la struttura del team di un'architettura in stile microservizio( sinistra) rispetto ai team di micro frontend (destra). Qui i team sono formati attorno a un'esigenza del cliente e non sulla base di tecnologie come frontend e backend.

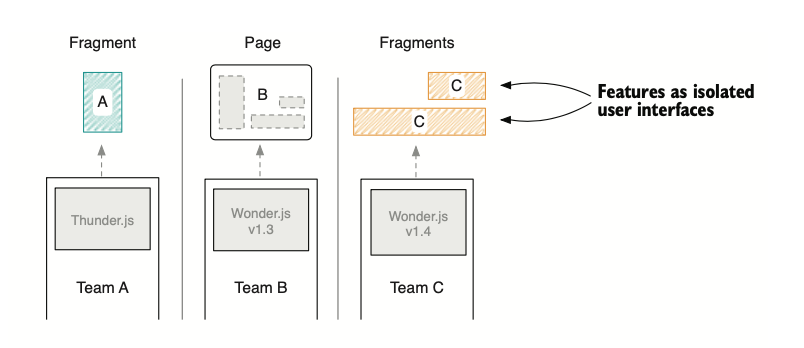
Tornando all’immagine a sinistra, le risorse sono raggruppate in base a competenze o tecnologie diverse. Gli sviluppatori di frontend stanno lavorando sul frontend, esperti nella gestione dei pagamenti lavorano su un servizio di pagamento. Anche gli esperti di business e operazioni formano i propri team. Questa struttura è tipica quando si utilizza un approccio basato sui micro servizi.

Agli sviluppatori di frontend piace lavorare con altri sviluppatori di frontend. Possono discutere i bug che stanno cercando di correggere o trovare idee su come migliorare una parte specifica del codice. Lo stesso vale per le altre squadre specializzate in una specifica abilità. I professionisti mirano alla perfezione e hanno un bisogno di trovare la migliore soluzione nel loro campo. Quando ogni squadra fa un ottimo lavoro, anche il prodotto nel suo insieme sarà ottimo, giusto?

Questa ipotesi non è necessariamente valida. La creazione di team interdisciplinari sta diventando sempre più popolare. Hai un team in cui gli ingegneri frontend e backend lavorano insieme. A causa delle loro diverse prospettive, escogitano soluzioni più creative ed efficaci per il compito da svolgere. Questi team potrebbero non creare la piattaforma o l’applicazione di front end migliore della categoria, ma sono specializzati nella missione del team. Ad esempio, stanno lavorando per diventare esperti nel presentare suggerimenti sui prodotti pertinenti, o creare un'esperienza di pagamento senza interruzioni. Invece di padroneggiare una tecnologia specifica, si concentrano tutti sul fornire la migliore esperienza utente per l'area in cui lavorano.

I team inter funzionali hanno il vantaggio aggiuntivo che tutti i membri sono direttamente coinvolti nello sviluppo delle funzionalità. Nel modello a micro servizi, i servizi o i team operativi non sono coinvolti direttamente. Ricevono i loro requisiti dal livello superiore e non sempre hanno il quadro completo del motivo per cui sono importanti.  
L'approccio di squadra inter-funzionale rende più facile per tutte le persone essere coinvolte, contribuire

## Il Frontend



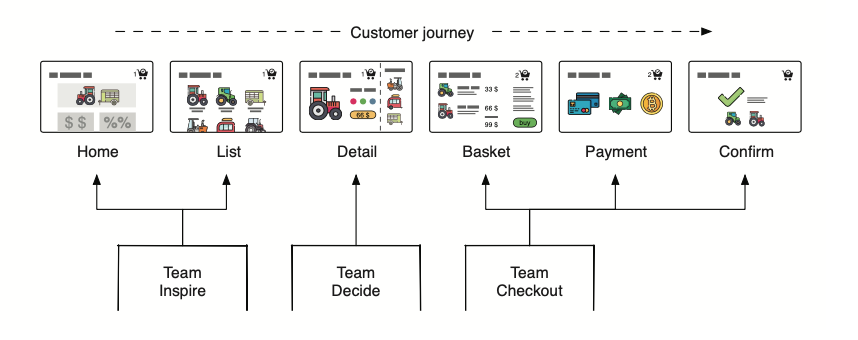
Nel caso quindi dei micro-front end, i vari team hanno la responsabilità end-to-end di una singola funzionalità(mi raccomando concentrarsi sul concetto di funzionalità e non di applicazione nel caso dello sviluppo di un micro front end).

Come illustrato nella porzione di immagine, un micro front end può essere associato sia ad una intera pagina, sia ad una porzione della nostra funzionalità che andrà inclusa nella nostra applicazione.

Un team genera HTML, CSS e JavaScript necessari per una determinata funzionalità. Per fare questo potrebbero utilizzare una libreria che in automatico esegua queste funzionalità. I team non condividono la libreria e il codice del framework. Ogni team è libero di scegliere lo strumento più adatto al proprio caso d'uso. Altro vantaggio come ripetuto più volte, è l’inpindenza sia funzionale che tecnica(ogni team potrà utilizzare la tecnologia che più preferisce(Angular, react , web components etc etc)).

Andiamo ora ad approfondire il concetto di pagina e il concetto di singola funzionalità (da ora in avanti useremo la parola fragment).

### Page ownership



Nel nostro esempio, abbiamo diversi team che si occupano di diverse parti del negozio. Se dividi un negozio online per tipi di pagina e provi ad assegnare ogni tipo a uno dei tre team, quello che potrebbe venire fuori è una struttura del genere.

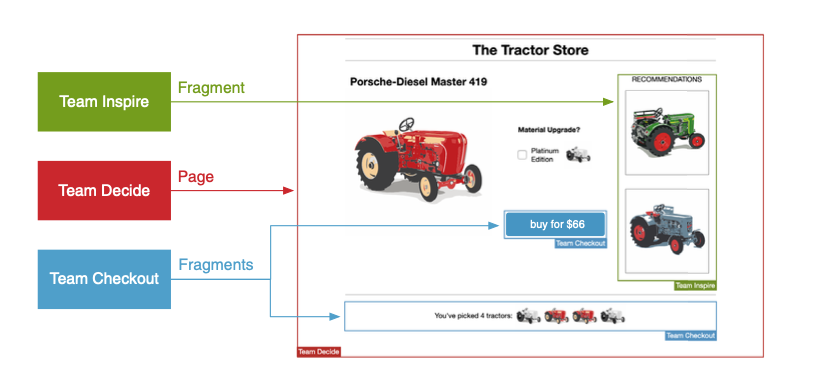
Ora come possiamo implementare questa struttura?

Ogni team potrebbe creare le proprie pagine, servirle dalla propria applicazione e renderle accessibili tramite un dominio pubblico. È possibile collegare queste pagine tramite collegamenti in modo che l'utente finale possa navigare tra di esse.

In questa maniera l’integrazione potrebbe risultare più complicata, tuttavia cominciamo a definire meglio quello che è il concetto di architettura a microfront end:

* I team possono lavorare in modo autonomo nel loro campo di competenza.
* I team possono scegliere lo stack tecnologico più adatto al lavoro da svolgere.
* Le applicazioni sono liberamente accoppiate e si integrano solo nel frontend (ad es. Tramite link)

### Fragments



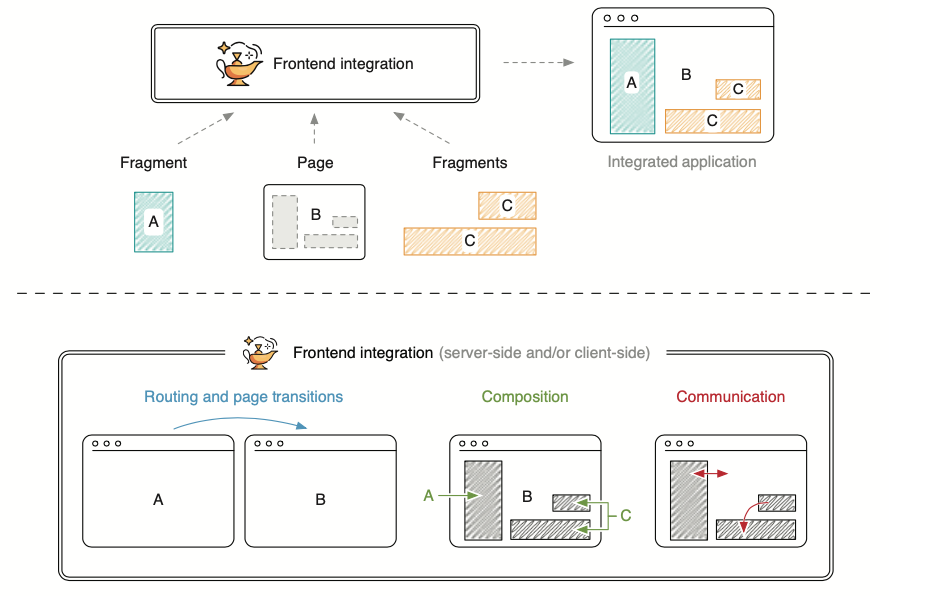
Il concetto di pagine non è sempre sufficiente. In genere hai elementi che compaiono su più pagine (esempi banali potrebbero essere l’header o il footer di una applicazione). Secondo uno schema architetturale distribuito su pagine, la logica vorrebbe che ogni team debba re implementare questa funzionalità. Tuttavia questa soluzione non è assolutamente migliore, creerebbe ridondanza di codice, scarsa manutenibilità, nonché esposta ad essere error prone

Ed è qui che entrano in gioco i Fragments

Posso quindi esserci delle funzionalità che in realtà competono ad altri team, ed è quindi loro ownership implementare e manutenere (da immagine è chiaro come questi requisiti vengano fuori).

Di conseguenza, un team può decidere di includere funzionalità di un altro team aggiungendole da qualche parte nella pagina. Alcuni frammenti potrebbero richiedere informazioni di contesto, come un riferimento al prodotto per il blocco Prodotti correlati. Il team che include il fragment nel proprio codice non deve conoscere lo stato e i dettagli di implementazione del fragment stesso.

## Integrazione del front end



In questa sezione verranno descritti i passaggi necessari che dovranno essere affrontati e discussi nella fase di integrazione di un micro front end. È utile inoltre ricordare che questi aspetti dovrebbero essere discussi in fase di progettazione della nostra architettura software, in maniera tale da poter identificare una roadmap architetturale (potrebbe esserci un team dedicato che si occupa di come gestire l’integrazione dei vari front end).

Verranno evidenziati principalmente tre aspetti relativi all’integrazione di un Front end:

* Routing e navigazione tra pagine diverse
* Composizione
* Comunicazione

### Routing e navigazione tra pagine

Qui stiamo parlando di integrazione a livello di pagina. Abbiamo bisogno di un sistema per passare da una pagina sviluppata dal Team A ad una pagina sviluppata del Team B. Le soluzioni possono essere semplici. Puoi ottenere questo risultato semplicemente utilizzando un file Link HTML. Può essere anche utilizzata una soluzione lato client, integrando soluzioni che eseguono il rendering della pagina successiva senza dover ricaricare(queste soluzioni al momento risultano più complicate “*Client-side routing*”).

Il client-side routing può essere implementato tramite due soluzioni:

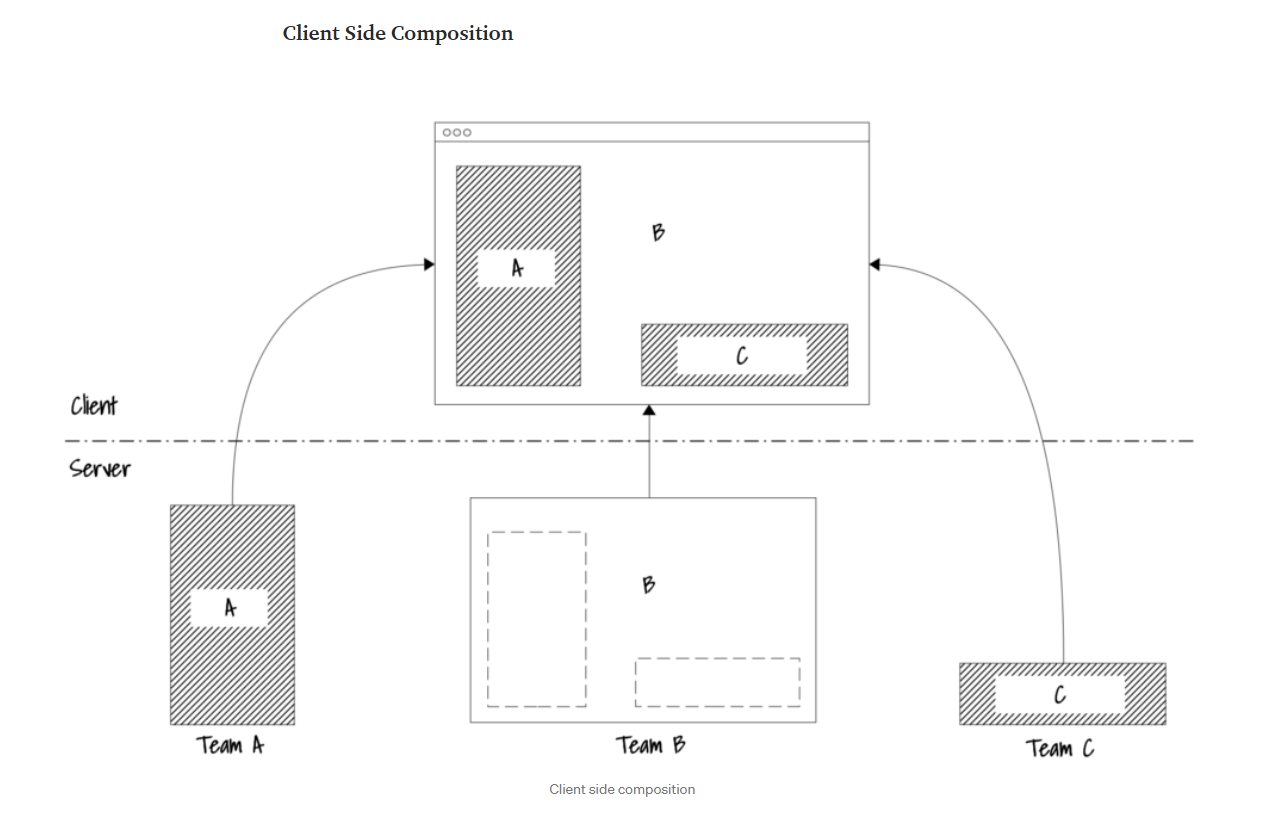
* Tramite l’implementazione di una Shared application shell
* Tramite l’utilizzo di un meta-framework come single-spa
  + <https://single-spa.js.org/>
* Routing tramite Nginx

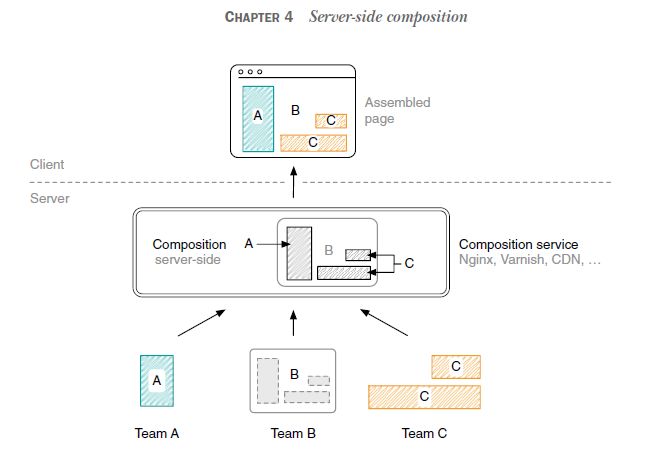
### Composizione

In questa fase vi è la composizione dei vari fragments tra di loro. Il team che solitamente integra un fragments, non recupera direttamente il suo contenuto, bensì solitamente utilizza un marker o un placeholder, che inserisce nel punto dove vuole che sia inserito quel fragments. Un esempio di questa logica sono l’utilizzo dei web component (javascript +html). Per ottenere questo risultato vi sono due tecniche principali:

* 1 Composizione lato server, ad esempio con SSI, ESI, Tailor o Podium
  + <https://podium-lib.io/>
  + <https://craftsmen.nl/tailor-made-micro-frontends/>
  + <https://www.esentri.com/composing-micro-frontends-server-side/>
* Composizione lato client, ad esempio con iframe, Ajax o web component (solitamente molto più semplici da utilizzare)
  + Nginx + SSI(activate ServerSideComposition)

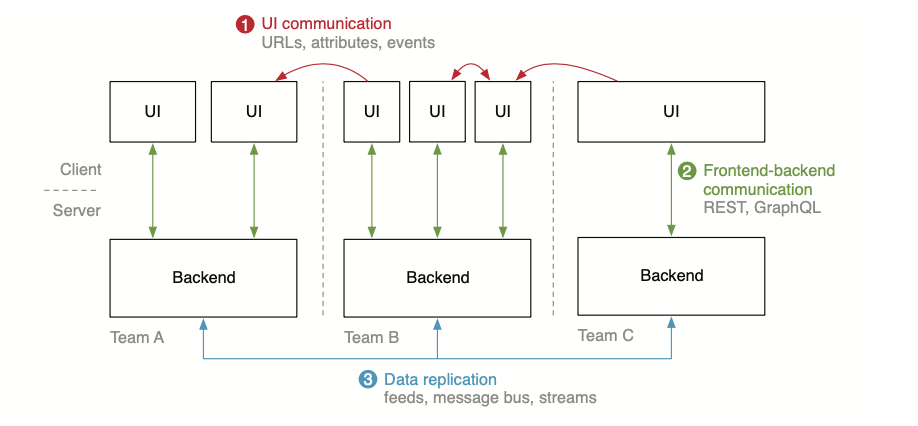
A seconda delle tue esigenze, potresti sceglierne uno, oppure una combinazione di entrambi.





### Comunicazione

Per le applicazioni interattive, è necessario anche un modello per la comunicazione. Come osservato in precedenza, potrebbe crearsi la necessità che alcuni fragment debbano comunicare tra di loro(vedi esempio comunicazione tra Iframe)



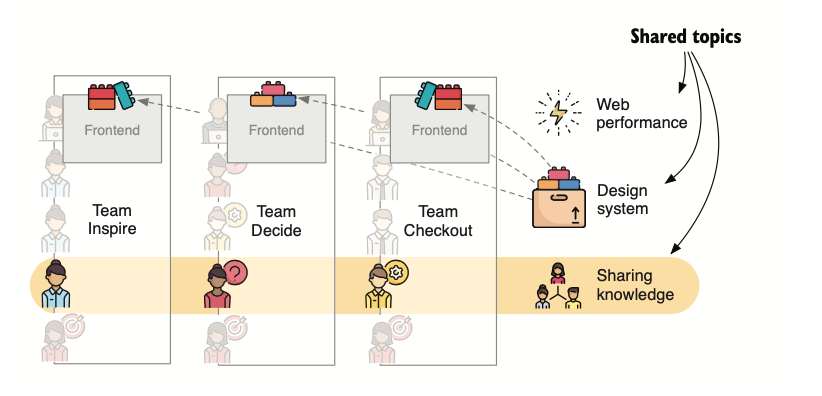
L’immagine riassume quelle che sono solitamente i differenti tipi di comunicazione in una applicazione a microfront end.

Possiamo riassumerli quindi in:

1. Comunicazione tra UI differenti.
2. Ogni frontend recupera i dati dal proprio backend e
3. In alcuni casi, è necessario replicare i dati tra i backend dei vari team.

## Shared topics

L'architettura dei micro frontend è incentrata sulla possibilità di lavorare in piccoli team autonomi che hanno tutto ciò di cui hanno bisogno per creare valore per il cliente. Tuttavia alcuni argomenti condivisi sono essenziali da affrontare quando si lavora in questo modo



### Web Performance

Poiché assembliamo una pagina da frammenti realizzati da più team, spesso ci ritroviamo con più codice che il nostro utente deve scaricare. È fondamentale avere un occhio sulle prestazioni della pagina fin dall'inizio. In questi casi è possibile anche creare delle librerie condivise, in maniera tale da ottimizzare quelli che sono le risorse condivise da più micro front end

### Design System

Il design è una parte fondamentale del nostro applicativo, e rientra quindi tra gli argomenti che vanno discussi e progettati precedentemente. Infatti sarà verosimile che molti elementi come pulsanti, campi di input, caratteri tipografici o icone, siano comuni. Anche in questo caso librerie comuni che possano agevolare lo sviluppo andrebbero previste, in maniera tale che il team rimanga concentrato solo sulla propria mission. È inoltre utile creare degli spazi che consentano lo scambio di informazioni comuni tra i vari team.

### Sharing Knowledge

Lo sviluppo a micro-front-end promuove l’autonomia architetturale, tuttavia andrebbe evitata la creazione anche di silos informativi. Un esempio è la creazione di una infrastruttura per la gestione degli errori. Non condividere la conoscenza sotto questo punto di vista, crea la situazione di duplicazione di logica, che potrebbe diminuire la produttività del team. Di conseguenza alcune tematiche andrebbero discusse, e trovate soluzioni comuni.

## CHALLENGE 2

Challenge basata sulla creazione di una struttura suddivisa per team, basata sulle richieste di un eventuale stakeholder.

* Applicare tutti i punti discussi precedentemente

# PARTE 2

## Perché scegliere un micro frontend.

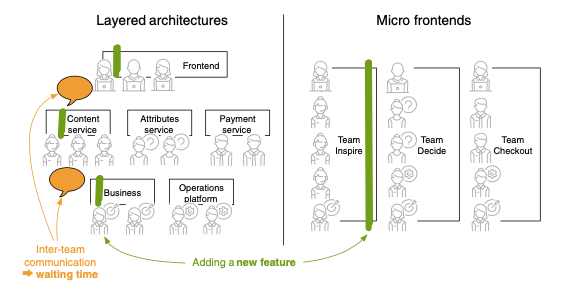
Dopo aver discusso di cosa è un micro-front end, cerchiamo di capire se è veramente adatto alle tue esigenze, e soprattutto se adatto alle esigenze del tuo cliente.

### Velocità

Il motivo principale per cui le aziende scelgono di seguire la rotta del micro frontend è aumentare la velocità di sviluppo. In un'architettura a più livelli, più team sono coinvolti nella creazione di una nuova funzionalità. Questo fattore è molto importante qual’ora una azienda riesca a suddividere il proprio carico di lavoro in tanti piccoli progetti. Di seguito un esempio

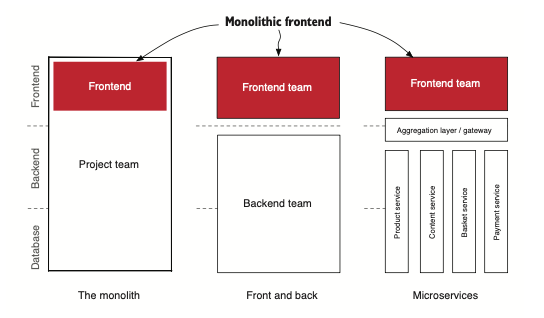
Supponiamo che il reparto marketing abbia l'idea di creare un nuovo tipo di banner promozionale. Parlano con il team dei contenuti per estendere la struttura dei dati esistente. Il team di contenuti parla con il team di frontend per discutere le modifiche alla loro API. Le riunioni vengono organizzate e le specifiche vengono scritte. Ogni team pianifica il proprio lavoro e lo pianifica in uno dei prossimi sprint. Se tutto funziona come previsto, la funzione è pronta quando l'ultimo team finisce di implementarla. In caso contrario, sono previste più riunioni per discutere le modifiche.

In questo caso è essenziale ridurre i tempi di attesa dei vari team, e l’utilizzo dei micro front end come architettura potrebbe essere la soluzione.



Questo diagramma mostra cosa serve per creare una nuova funzionalità. Sul lato sinistro, vedi un'architettura a strati. Tre squadre sono coinvolte nella sua costruzione. Queste squadre devono coordinarsi e potenzialmente aspettarsi a vicenda. Con l'approccio dei micro frontend (a destra), un team può creare questa funzionalità in maniera indipendente, in quanto tutte le persone necessarie sono già organizzate all’interno dello stesso team.

### Micro frontend vs frontend monolitico



La maggior parte delle architetture odierne non ha un concetto per ridimensionare lo sviluppo del frontend. In questa immagine sono mostrate attualmente le architetture utilizzate attualmente nella maggior parte delle casistiche. Questa architettura sgnifica che il front end proviene unicamente da una singola base di codice su cui solo un team può lavorare. Con i micro frontend, l'applicazione, incluso il frontend, viene suddivisa in sistemi verticali più piccoli. Ogni squadra ha il proprio frontend più piccolo, rispetto ad un monolite frontend, costruire e mantenere un frontend più piccolo ha dei vantaggi. Potremmo riassumere i vari vantaggi in:

* Deploy indipendente
* Il rischio di guasti è isolato ad una area più piccola
* Più comprensibile dal punto di vista funzionale (una nuova risorsa può imparare e capire più velocemente come funziona l’applicazione)
* Ha una base di codice più piccola che può essere d'aiuto quando vuoi rifattorizzarla o sostituirla

### Velocità di aggiornamento tecnologico

In qualità di sviluppatore di software, l'apprendimento costante e l'adozione di nuove tecnologie fanno parte del lavoro. Ma quando lavori nello sviluppo del frontend, questo è particolarmente vero. Negli anni lo sviluppo front end si è evoluto (e anche molto velocemente rispetto al mondo backend). Nel lontano 2005 le tecnologie più utilizzate erano Ruby on Rails, Prototype.js e Ajax, essenziali per uno sviluppo web complesso.

Attualmente lo sviluppo front end è diventato un vero e proprio campo professionale di ingegneria. Per fornire un buon lavoro, uno sviluppatore web al giorno d'oggi ha bisogno di conoscere argomenti come il responsive design, l’usabilità, le prestazioni web, i componenti riutilizzabili, la testabilità, l'accessibilità, la sicurezza e le modifiche negli standard web e il supporto del browser. L'evoluzione di strumenti, librerie e framework di frontend ci ha permesso di creare applicazioni web di qualità superiore e piuù capaci per soddisfare le crescenti aspettative dei nostri utenti. Strumenti come Webpack, Babel, Angular, React, Vue.js attualmente oggi svolgono un ruolo fondamentale. Tuttavia potremmo ancora non aver raggiunto la fine di questa evoluzione. Essere quindi in grado di adottare una nuova tecnologia (e anche abbastanza velocemente) è diventato un punto cruciale delle nostre applicazioni.

### Sistemy legacy

Anche trattare con i sistemi legacy sta diventando un argomento più diffuso nel frontend. Un sacco di tempo dello sviluppatore viene speso per il refactoring del codice legacy e per elaborare strategie di migrazione. I grandi attori stanno investendo una notevole quantità di lavoro nel mantenere le loro grandi applicazioni. Ecco tre esempi:

* GitHub ha effettuato una migrazione pluriennale per rimuovere la loro dipendenza da jQuery.
* Trivago, un motore di ricerca di hotel, ha fatto uno sforzo enorme con Project Ironman per rielaborare il loro complesso CSS in un sistema di progettazione modulare.
* Etsy ([https://www.etsy.com](https://www.etsy.com/it/)) si sta sbarazzando del loro bagaglio legacy JavaScript per ridurre le dimensioni del pacchetto e aumentare le prestazioni web. Il codice è cresciuto nel corso degli anni e uno sviluppatore non può avere una panoramica del sistema completo. Per identificare il codice morto, hanno creato uno strumento di copertura del codice nel browser che viene eseguito nel browser del cliente e riporta ai loro server.

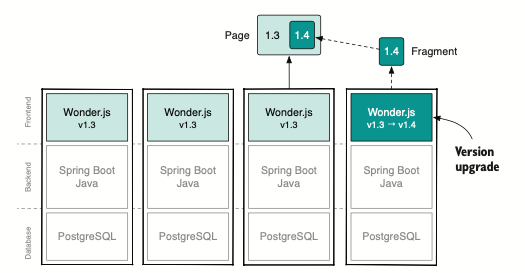
Quando stai realizzando un'applicazione di una dimensione specifica e vuoi rimanere competitivo, è essenziale essere in grado di passare a nuove tecnologie quando forniscono valore al tuo team. (Questo tuttavia non vuol dire riscrivere il tuo codice da zero ogni anno).

### Sistema decisionale isolato

L’approccio dei micro front end prevede un processo decisionale molto più veloce ed intuitivo, in quanto attuare decisioni (sia a livello architetturale che progettuale), richiede meno tempo, essendo il tutto svolto a livello di team. Con l'approccio dei micro frontend, i team hanno il pieno controllo del loro stack tecnologico ( micro architettura), e a fronte di varie riunioni, possono decidere in autonomia se cambiare magari linguaggio (da javascript a typescript). Essendo inoltre il team ristretto, è anche più semplice eseguire un improvement tecnologico sulle risorse. In questo caso, l’unica accortezza da avere, è quella di rimanere compatibili con la convenzione di comunicazione tra micro frontend scelta precedentemente.

### Indipendenza

L'autonomia è uno dei vantaggi critici dei microservizi e anche dei micro frontend. È utile quando i team decidono di apportare modifiche più significative come descritto nella sezione precedente. Ma anche quando lavori in un ambiente omogeneo in cui tutti usano lo stesso stack tecnologico, ha i suoi vantaggi.



Le pagine e i frammenti sono autonomi. Ciò significa che portano i propri markup, stili e script e non dovrebbero(dipende anche qui dalle scelte architetturali) avere dipendenze di runtime condivise. Questo isolamento consente a un team di distribuire una nuova Funzionalità in un frammento senza dover prima consultare altri team. Un aggiornamento può anche essere fornito con una versione aggiornata del framework JavaScript che stanno utilizzando.

### Sovraccarico di risorse?

I microservizi di backend introducono un sovraccarico. Sono necessarie più risorse di elaborazione, ad esempio, per eseguire diverse applicazioni Java nella propria macchina virtuale o contenitore. Ma il fatto che i servizi di backend siano essi stessi molto più piccoli di un monolite ha anche dei vantaggi. È possibile eseguire un servizio su hardware più piccolo ed economico. Puoi scalare servizi specifici eseguendone più istanze e non devi moltiplicare il monolite completo. Puoi sempre risolvere questo problema con denaro e acquistare istanze di server più o più grandi.

Questa logica purtroppo non può essere applicata ai frontend, in quanto le risorse dei dispositivi sono limitate. Tuttavia attualmente esistono delle metriche per quantificare e migliorare le tecniche per mitigare effetti di sovraccarico computazionale (purtroppo solo questo capitolo richiederebbe uno studio a se stante).

### Shared Nothing

Alla base di tutto questo ragionamento, c’è la consapevolezza che l’interazione tra i vari team è costosa, ed il motivo per cui non dovrebbero crearsi risorse condivise, è perché l’interazione tra i vari team possono essere molto costose. Quando vuoi cambiare un pezzo su cui gli altri fanno affidamento, che sia solo una libreria di utilità, devi informare tutti, attendere il loro feedback e magari discutere altre opzioni. Più persone hai, più complicato diventa. L'obiettivo è condividere il meno possibile per consentire uno sviluppo più rapido delle funzionalità. Ogni parte di codice o infrastruttura condivisa ha il potenziale per creare una quantità non banale di overhead di gestione.

Questo approccio della non condivisione di risorse è anche chiamato “*shared nothing architecture”.*

Questa metodologia di approccio architetturale ha una forte tendenza ad accettare la ridonzanza a favore di una maggiore autonomia e velocità di iterazioni e sviluppo.

## Gli svantaggi di un micro frontend

Come affermato in precedenza, l'approccio dei micro frontend consiste nel dotare i team autonomi di tutto ciò di cui hanno bisogno per creare funzionalità significative per il cliente. Questa autonomia è molto utile, ma non è gratuita.

### Ridondanza

Ogni bravo informatico ha studiato per ridurre al minimo la ridondanza nei sistemi che vengono progettati. Sia che si tratti della normalizzazione dei dati in un database relazionale o dell'estrazione di parti di codice simili in una funzione condivisa. L'obiettivo è aumentare l'efficienza e la coerenza. I nostri occhi e le nostre menti hanno imparato a trovare il codice ridondante e trovare una soluzione per eliminarlo.

Avere più team fianco a fianco che costruiscono ed eseguono il proprio stack introduce molta ridondanza. Ogni team deve configurare e mantenere il proprio server delle applicazioni, il processo di creazione e la pipeline di integrazione continua e potrebbe inviare codice JavaScript / CSS ridondante al browser. Di seguito due esempi:

* Un bug critico in una libreria popolare non può essere risolto in un punto centrale. Tutti i team che lo utilizzano devono installare e distribuire autonomamente la correzione.
* Quando un team si è impegnato per rendere il proprio processo di costruzione due volte più veloce, gli altri team non traggono automaticamente vantaggio da questo cambiamento. Questa squadra deve condividere queste informazioni con gli altri. Gli altri team devono implementare la stessa ottimizzazione da soli.

Il ragionamento alla base di una architettura a micro-front end, è che i costi relativi ad un tipo di architettura “*shared nothing architecture” sono inferiori rispetto agli impatti negativi e ai costi introdotti dalla dipendenza e interazione tra i team.*

### Consistenza

Questa architettura richiede che tutti i team abbiano il proprio database per essere completamente indipendenti. Ma a volte una squadra ha bisogno di dati che un'altra squadra possiede. In un negozio online, il prodotto ne è un buon esempio. Tutti i team devono sapere quali prodotti offre il negozio. Una soluzione tipica per questo è la replica dei dati utilizzando un bus di eventi o un sistema di alimentazione. Un team possiede i dati del prodotto. Gli altri team replicano questi dati regolarmente. Quando una squadra viene interrotta, le altre non sono interessate e hanno ancora accesso alla loro rappresentazione locale dei dati. Ma questi meccanismi di replica richiedono tempo e introducono latenza. Pertanto, le variazioni di prezzo o disponibilità potrebbero essere incoerenti per brevi periodi di tempo. Un prodotto promosso con uno sconto sulla home page potrebbe non avere questo sconto nel carrello. Quando tutto funziona come previsto, stiamo parlando di ritardi nella regione di millisecondi o secondi, ma quando qualcosa va storto, questa durata può essere più lunga. È un compromesso che privilegia la robustezza rispetto alla consistenza garantita.

### Eterogeneità

La libera scelta della tecnologia è uno dei vantaggi più significativi introdotti dai micro frontend, ma è anche uno dei punti più controversi. Voglio che tutti i team di sviluppo abbiano uno stack tecnologico completamente diverso? Ciò rende più difficile per gli sviluppatori passare da un team all’altro o persino scambiarsi delle best practice. Tuttavia posso coesistere vari livelli di eterogeneità, da ognuno usa stack tecnologici differenti, a tutti che usano lo stesso stack tecnologico. Queste sono scelte che possono essere pensate e discusse precedentemente a livello aziendale, anche solo per limitare il livello di libertà.

## Quando adottare un micro frontend

Come tutti gli approcci all’interno del mondo informatico, non esiste “LA SOLUZIONE”, il santo graal architetturale, bensì esiste la giusta soluzione al problema da affrontare, ed è giusto in fase architetturale, discutere e comprendere i vantaggi ma anche io limiti di una determinata architettura.

### Ottimo per progetti medio-grandi

L'architettura dei micro frontend è una tecnica che semplifica il ridimensionamento dei progetti. Quando lavori su un'applicazione con poche persone, il ridimensionamento probabilmente non è il tuo problema principale.

Secondo Jeff Bezos (*Two-Pizza Team Rule*) , una squadra è troppo grande quando due pizze grandi non possono alimentarla. In gruppi più grandi, il sovraccarico di comunicazione aumenta e il processo decisionale diventa complicato. In pratica, ciò significa che la dimensione della squadra perfetta è compresa tra 5 e 10 persone. Se la dimensione della squadra supera le 10 persone, suddividere secondo una architettura a micro-front end potrebbe aiutare

### Applicare micro frontend al web funziona meglio

Sebbene le idee alla base dei micro frontend non siano limitate a una piattaforma specifica, funzionano meglio sul web.

Le applicazioni native(come IOS e Android), sono monolitiche per progettazione. La composizione e la sostituzione delle funzionalità “al volo” non è possibile. Per aggiornare l’App nativa, devi creare un singolo pacchetto di applicazioni che viene quindi inviato al processo di revisione di Apple o Google. Un modo per aggirare questo è caricare parti dell'applicazione dal web. I browser incorporati o le Web view possono aiutare a mantenere la parte nativa dell'app al minimo. Ma quando devi implementare l'interfaccia utente nativa, è difficile avere più team end-to-end che lavorano senza pestarsi l'un l'altro. Quindi, se il Web è la tua piattaforma di destinazione, i micro frontend potrebbero essere una buona soluzione. Se devi rivolgerti anche ai nativi, devi fare alcuni sacrifici.

### Produttività contro spese generali - costi (setup e complessità organizzativa)

Dividere la tua applicazione in sistemi autonomi porta molti vantaggi, ma non è gratuita. Potrebbero essere richieste più risorse per implementare questo tipo di architettura, inoltre la fase iniziale del progetto potrebbe richiedere più tempo(definizione name spacing, configurazione implementazioni dei vari sistemi, decisioni su strategie di implementazione etc etc). Avere sistemi verticali più piccoli riduce la complessità tecnica dei singoli sistemi. Ma l'esecuzione di un sistema distribuito aggiunge la sua complessità in cima.

Prendiamo l’esempio di un browser, che possiede un ambiente di run-time condiviso. In caso di malfunzionamento, quale team è coinvolto nella risoluzione del problema?

Un cambiamento effettuato sull’applicazione da un team potrebbe avere effetti negativi dal punto di vista prestazionale, in questi casi potrebbe essere difficile individuare il diretto responsabile

## Dove non sono adatti i micro frontend?

Ma ovviamente, i micro frontend non sono perfetti per ogni progetto. Come affermato in precedenza, sono una soluzione per il ridimensionamento dello sviluppo. Se hai solo una manciata di sviluppatori e la comunicazione non è un problema, l'introduzione di micro frontend non porterà molto valore. È fondamentale conoscere bene il dominio in cui stai lavorando per fare buoni tagli verticali. Idealmente, dovrebbe essere ovvio quale team è responsabile dell'implementazione di una funzionalità. Missioni di squadra poco chiare o sovrapposte porteranno a incertezze e lunghe discussioni.

Se è necessario creare molte app diverse e interfacce utente native da eseguire su ogni dispositivo, potrebbe anche diventare complicato da gestire per un team.

## Chi usa i micro front end?

Amazon ne è un esempio, il loro e-commerce è stato sviluppato in questo modo da molti anni ormai. Amazon utilizza anche una tecnica di integrazione dell'interfaccia utente che assembla le diverse parti della pagina prima che raggiunga il cliente. Spotify ha creato varie tema end-to-end autonomi chiamati SQUAD, inoltre DAZN ha ricostruito il suo front end monolito sulla base di una architettura a micro-front end.

# Concetti conclusivi essenziali

* I micro frontend sono un approccio architettonico e non una tecnica specifica.
* I micro frontend rimuovono la barriera del team tra sviluppatori frontend e backend introducendo team inter funzionali.

# PARTE 3

Dopo aver approfondito e compreso cosa è un micro front end, cerchiamo di accennare qualche soluzione tecnica utilizzata per implementare un micro Front end. Partiremo dagli esempi più semplici, fino a quelli un po' più complessi Per eseguire tutti i vari esempi bisogna aver installato sulla propria macchina:

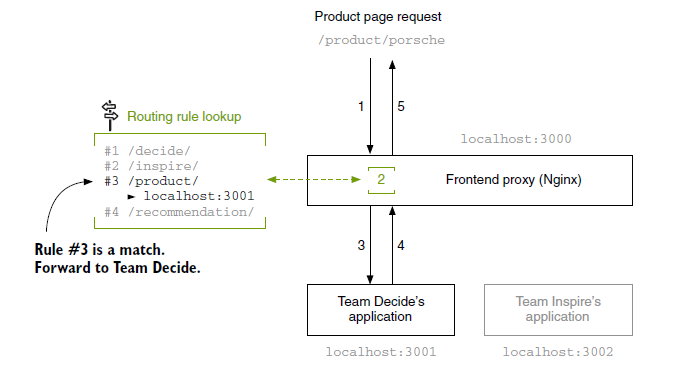
* https://nodejs.org/
* nginx
  + macOS (via [*Homebrew*](https://brew.sh/)): brew install nginx
  + *Debian/Ubuntu:* sudo apt-get install nginx
  + *Windows:* Binaries inlcusi nella cartella degli esempi

L’obbiettivo di questa seconda parte del corso, non è fornire una metodologia precisa per costruire un microFront end. Questo perché proprio a causa della natura multipla di questa architettura; le tecnologie che possono essere messe in campo sono molteplici, potrei voler scrivere un microFront end in Angular, e un altro in React, e le due configurazioni potrebbero non essere le stesse. Motivo per cui per ogni tecnologia affrontata, andrà richiesto un approfondimento personale. Verranno discussi inoltre pro e contro di ogni tecnologia messa in atto. Ogni tecnologia richiederà un approfondimento maggiore su come configurare l’applicazione lato server-side.

## IFRAME

Una delle tecniche utilizzate attualmente per sviluppare microFrontEnd, è attraverso l’utilizzo degli IFrame, che Spotify attualmente sviluppa la proprio applicazione utilizzando una architettura a microfront. La complessità degli IFrama non verrà discussa in quanto di facile utilizzo.

## AJAX(NGINX Server side routing/composition)



### Quando utilizzare questa tecnica

Questa tecnica ha notevoli vantaggi, come:

* Risolve il problema del cors quando integriamo vari microFrontEND
* Consente la condivisione di dati come lo stato di accesso tramite i cookie
* Migliora notevolmente le performance lato client.

Questa tecnica inoltre permette di creare siti web, che debbano essere indicizzati dai browser. Per un'applicazione interna, esso

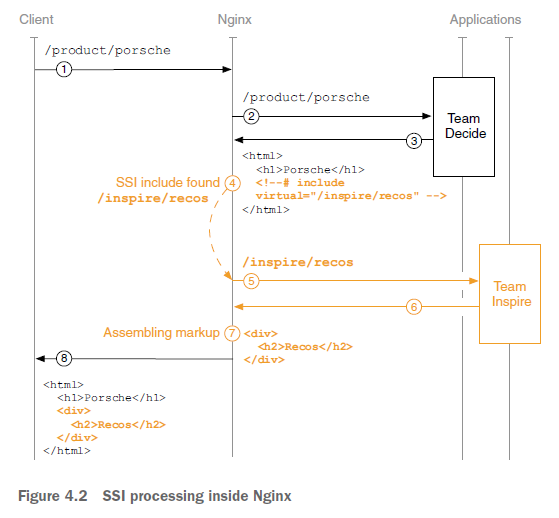
potrebbe anche andare bene saltare l'infrastruttura extra e andare con un sottodominio-perteam

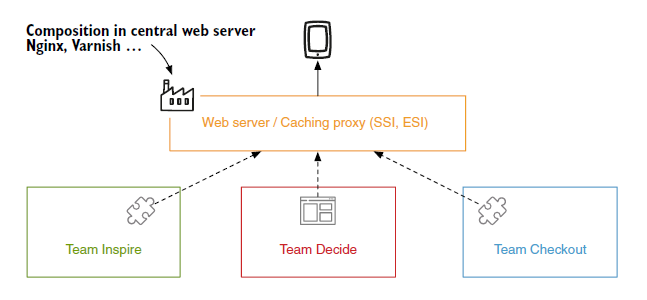
approccio.

### Server side composition

Nginx può essere anche configurato per la compositione lato server(SSI – server side inclusion). Questa tecnica migliora in maniera sorprendente le performance lato client.

### Come lavora NGINX SSI





### Aspetti negativi dell’utilizzo dell’SSI

Se la pagina da comporre è molto grande, il server potrebbe richiedere molto tempo e risorse per renderizzare, e quindi il primo caricamento dell’applicazione potrebbe richieder molto tempo.

Inoltre, l'integrazione lato server non viene fornita con l'approccio tecnico per isolare le varie componenti che vengono costruite. Bisogna quindi fare molto affidamenti sulla costruzione incapsulata dei tag html e css (potrebbe esserci conflitti e override). In questi casi quindi lo sviluppo potrebbe diventare più complicato.

### AJAX(Docker Server side composition)

Semplice cenno sulla possibilità di dockerizzare un progetto tramite nginx e Docker compose.

## WEB COMPONENTS(angular elements)

I web components, anche conosciuti come Custom element, sono una importante novità, in quanto permettono di creare componenti tramite l’utilizzo di semplice codice html e javascript. L’altra grande novità è che questi ultimi sono utilizzabili all’interno di qualsiasi libreria/framework. L’unico problema a cui bisogna prestare attenzione, è la compatibilità tra i vari browser.

### Web component e custom elements

La specifica dei componenti Web, è un modalità di sviluppo introdotta ormai da parecchio tempo all’interno del mondo dei front end. Il suo obiettivo è introdurre miglior incapsulamento e consentire l'interoperabilità tra diverse librerie o frameworks. Al momento tutti i principali browser possono utilizzare questa specifica di sviluppo. È anche possibile aggiornare l'implementazione in browser meno recenti che utilizzano un polyfill(vedi il solito internet explorer). Web Components è un termine generico. Descrive tre nuove API distinte:

* Custom Elements,
* Shadow DOM
* HTML template.

Noi ci concentreremo principalmente sui custom elements. Questi utlimi consentono di fornire funzionalità in un modo dichiarativo attraverso l’utilizzo del DOM. Puoi interagire con gli elementi personalizzati allo stesso modo modo in cui interagiresti con gli elementi HTML standard. Inoltre i web components possono essere progettati anche per incapsulare logica di business(vedi esempio).

Per poter creare ed utilizzare un custom elements, dobbiamo principalmente seguire 3 passaggi:

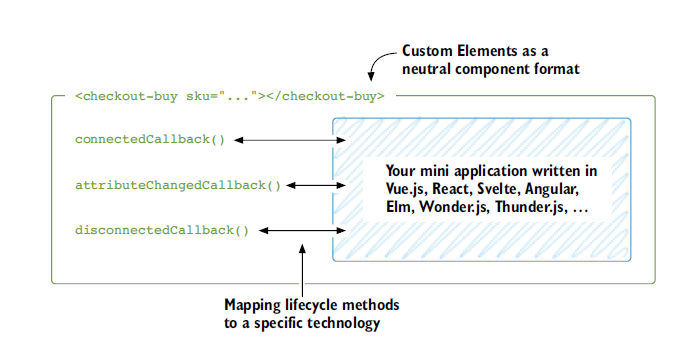
1. Dichiarazione del custom element
2. Utilizzo del custom element tramite tag dichiarato
3. Parametrizzazione tramite passaggio in input di vari attributi

### Ciclo di vita di un custom elements

Molto importante è capire il ciclo di vita di un custom element. Non avendo a disposizione tutta l’infrastruttura tipica di un framework (vedi Angular con ngOn init), dobbiamo stare molto attenti a preoccuparci di gestire adeguatamente il ciclo di vita di un custom elements. Il ciclo di vista si compone principalmente dei seguenti elementi:

* Constructor
* Connected-Callback
* disconnectedCallback,
* attributeChangedCallback

È semplice collegare questi cicli di vita metodi al codice di (de-) inizializzazione del framework o della libreria su cui stai lavorando, in quanto inviano delle notifiche all’applicazione che li sta integrando. Angular mette a disposizione gli angular Elements.



### Benefici dei web components

Il vantaggio più significativo dell'utilizzo dei web component come tecnica di integrazione, è ormai lo standard implementativo utilizzato ormai consolidato (sostanzialmente javascript/html/css). Questo perché lo standard web a evolve molto velocemente, prendendo spesso anche molte implementazioni tecnologiche, e molto spesso non tutti i tipi di librerie messe a disposizione, hanno vita lunga.

Inoltre i metodi del ciclo di vita introdotti dai Custom Elements rendono possibile il wrapping

del codice in diverse applicazioni in modo standard. Questa applicazioni possi quindi usare i componenti creati in maniera totalmente dichiarativa(tramita tag).

### Svantaggi dei web component

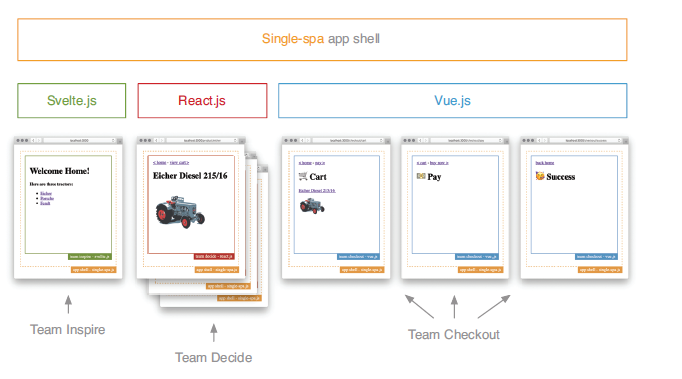
Uno degli svantaggi principali, è che i web component richiedono una composizione obbligata lato client, è questo può rendere l’applicazione lenta in termini di performance, infatti come abbiamo visto, è possibile anche fare la renderizzazione dei componenti lato server (vedi nginx + docker).

Un altro problema potrebbe essere legato alla compatibilità su browser più vecchi. Se lo sviluppo è orientato all’utilizzo delle ultime versioni dei browser, non ci dovrebbero essere problemi, altrimenti è meglio approfondire meglio la loro compatibilità.

VEDI CODICE DI ESEMPIO:

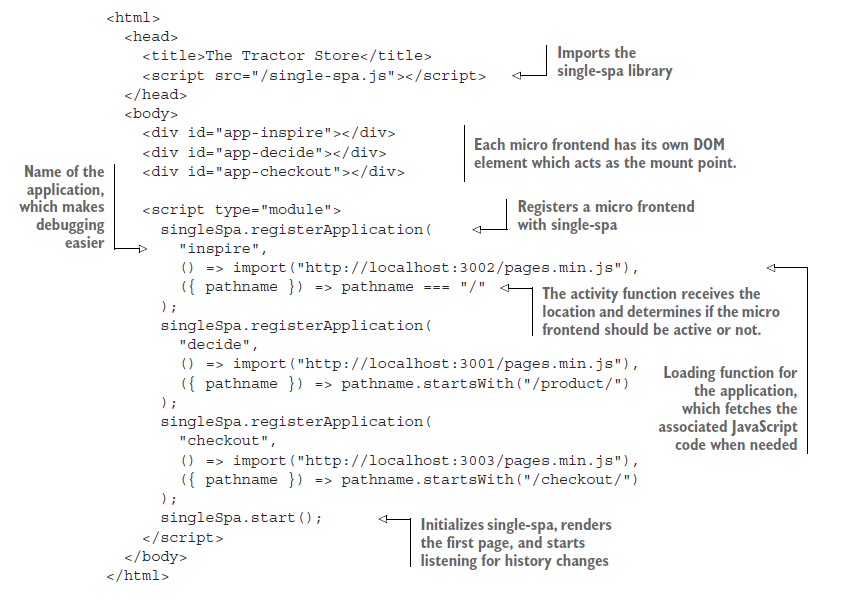
## SINGLE-SPA (Client-side-composition)

Single-spa è un framework javascript per microservizi front-end e può essere implementato con tutti e tre i framework / librerie popolari come Angular, React e Vue.js. Può caricare in maniera lazy le applicazioni in base alle necessità.



Il concetto di base è molto semplice. Abbiamo un unico HTML file che funge da punto di partenza. Include il codice JavaScript e le mappe della single-spa

La differenza principale, è che non vengono esposti dei dei webComponent , bensì i team espongono i loro micro frontend come oggetti JavaScript che aderiscono a un'interfaccia specifica.



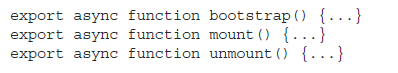
All'inizio, single-spa confronta l'URL corrente con tutti i micro frontend registrati. Single-spa richiama la funzione di **BOOTSTRAP** per determinare se l’applicazione è attiva.

L'applicazione diventa attiva per la prima volta, single-spa recupera il JavaScript

tramite la funzione di **MOUNT** e lo inizializza. Quando applicazione attiva

diventa inattivo, single-spa chiama la sua funzione di **UNMOUNT**, istruendolo a deinizializzarsi.

Possono essere attive più applicazioni contemporaneamente.



Queste funzioni (bootstrap, mount, unmount) sono simili al ciclo di vita dei custom elements messi a disposizione da angular(LINK CUSTOM ELEMENT). Singlespa richiama la funzione bootstrap quando un micro frontend diventa attivo per la prima volta. richiama (un) mount ogni volta che l'applicazione viene (de) attivata.

Tutte le funzioni del ciclo di vita sono asincrone. Questo fatto rende il caricamento il caricamento all’interno dell’applicazione in maniera lazy. Inoltre questa libreria ci assicura che la funzione di MOUNT venga chiamata sempre dopo la funzione di bootstrap.

## RESOCONTO DI SINGLE-SPA

Tramite l’utilizzo di questa libreria, l’esperienza lato utente viene notevolmente migliorata, tuttavia abbiamo dei lati negativi che andranno affrontati e analizzati in fase di studio architetturale.

### Error-handling

Se il codice di diversi team viene eseguito all'interno di un documento, a volte può essere complicato scoprire dove ha avuto origine un errore. Il codice contenuto in un frammento può potenzialmente causare comportamenti indesiderati sulla pagina completa(come per esempio lo sviluppo di una scrollbar personalizzata).

### Memory management

Trovare dei memory leak è più complicato che rintracciare un errore JavaScript. Una causa comune può essere l’inadeguata pulizia della memoria di un microfrontend. rimuovere parti del DOM senza annullare la registrazione dei listener di eventi

o scrivere qualcosa in una posizione globale e poi dimenticarsene. Poiché le applicazioni micro frontend vengono

inizializzate e deinizializzate regolarmente, anche problemi più piccoli nella pulizia possono accumularsi in un problema più grande.

Single-spa ha un plugin chiamato single-spa-leaked-globals che cerca di ripulire le variabili

globali dopo che un micro frontend è stato disattivato. Tuttavia l’unica soluzione al momento valida rimane quella di sensibilizzare i team di sviluppatori sul fatto che un unmountcorretto è importante quanto un mount corretto.

### Single point of failure

La shell dell'app è, per sua natura, l'unico primo punto di contatto. Avere un grave errore nella shell dell'app può far cadere l'intera applicazione. Ecco perché il codice della shell dell'app dovrebbe essere di alta qualità e ben testato. Mantenerlo concentrato e snello aiuta a raggiungere questo obiettivo.

### Shell management

La shell dell'applicazione è una parte fondamentale dell'infrastruttura che necessita di un chiaro proprietario. Tuttavia, una volta che si dispone di un sistema funzionante, non dovrebbe esserci una forte richiesta di aggiungere funzionalità o di far evolvere costantemente la shell dell'applicazione stessa. Se la shell della tua app è snella, di solito va benissimo che uno dei team di funzionalità si assuma la responsabilità di mantenerla.

### Comunicazione

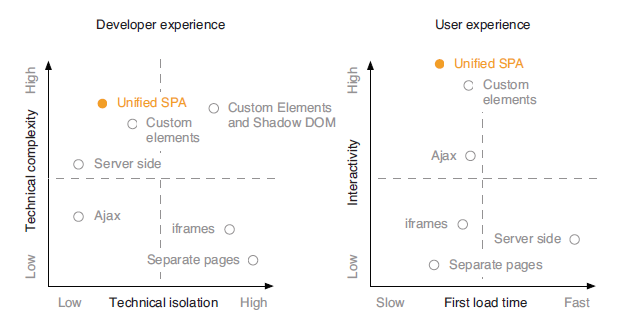
A volte il micro frontend A ha bisogno di sapere qualcosa che è successo nel micro frontend B. Le soluzioni possono essere:

* Condivisione del contestro tramite l’utilizzo della URL.
* Comunicazione trami API esposte dal backend
* Eventuali listener applicativi(ricordarsi di disattivarli)

### Quando utilizzarla

Come detto all’inizio, ha senso utilizzare questa metodologia quando la pagina web deve fornire un alto livello di interattività, a discapito del tempo di caricamento iniziale.

# RESOCONTO SCHEMA FINALE



# Casi di successo



## DAZN

DAZN è un servizio di streaming sportivo fondato in Europa ma che si è espanso in nove paesi in tutto il mondo.

I micro-frontends hanno permesso a DAZN di potenziare i team più piccoli che possono lavorare in modo indipendente. I micro-frontends servono a creare il tipo di decentralizzazione che le organizzazioni più grandi devono impiegare per mantenere la velocità sul mercato.

## **IKEA**

L’azienda europea di mobili (che è quasi ugualmente famosa per servire le polpette svedesi nella sua food court) utilizza micro-frontends per la loro esperienza di negozio online.

## Upwork

La piattaforma freelance per i professionisti aziendali indipendenti per connettersi e collaborare è passata a un’architettura micro-frontends nel 2017, come parte di uno sforzo globale per modernizzare la loro esperienza utente.

## Spotify

Spotify, il servizio di streaming multimediale online, utilizza micro-frontends per le loro applicazioni desktop e utilizza iframe per assemblare l’applicazione insieme.

SoundCloud

Anche SoundCloud, la piattaforma europea di distribuzione audio online e sito di condivisione di musica, ha visto i propri team trarre vantaggio dall’utilizzo di micro-frontends.

## HelloFresh

HelloFresh è un’azienda internazionale di kit per pasti con sede a Berlino, in Germania, e funge da più grande fornitore di kit per pasti negli Stati Uniti. E i loro team di sviluppo sono stati in grado di superare le barriere all’innovazione rompendo il monolite front-end.

## Zalando

Zalando è una società di e-commerce con sede in Germania. Tuttavia, fanno di più che vendere prodotti online. Oltre all’e-commerce, l’azienda è a capo anche di [Project Mosaic](https://www.mosaic9.org/), che è un insieme di servizi e librerie che supportano un’architettura in stile micro-frontends per grandi siti web, fornendo un contributo significativo a questo metodo di sviluppo frontend.