Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

A.A. 2021/2022

Tesi di Laurea Magistrale

Architetture a Micro Frontend

Relatore:
Giovanni Malnati

Candidato
Samuele Mannalà

Abstract

Le modalità di sviluppo delle applicazioni web si sono notevolmente evolute nel corso degli ultimi decenni: al giorno d'oggi, le pagine web devono infatti essere altamente reattive, rapide nel caricamento e funzionanti su diversi dispositivi, offrendo quindi livelli di interazione diversi per i vari utenti.

Il front-end, gestendo l'interazione con gli utenti, è la principale interfaccia che permette la comunicazione dell'applicativo con il consumatore finale, poiché fornisce gli strumenti necessari per l'interazione con l'applicazione, svolgendo quindi un ruolo cruciale all'interno dell'esperienza utente.

Quando si tratta di progetti di piccole dimensioni gestiti da un team ristretto di sviluppatori, realizzare e gestire un'applicazione web performante e coerente è relativamente semplice. Tuttavia, per le aziende che dispongono di applicazioni di grandi dimensioni, riuscire a garantire una perfetta coordinazione tra i team e una consistenza interna al progetto risulta molto complicato, soprattutto se si desidera effettuare degli sviluppi specifici per aggiungere nuove funzionalità.

Per ovviare a queste criticità, è stata quindi sviluppata una nuova tecnica di sviluppo dell'architettura front-end: il micro front-end. L'obiettivo dell'architettura a micro front-end è, al contrario delle altre architetture, quello di modularizzare e suddividere l'applicazione in più sotto applicazioni, sviluppate e gestite autonomamente da team diversi e indipendenti tra di loro, prendendo ispirazione dall'architettura a microservizi utilizzata nello sviluppo back-end. Tale approccio permette di migliorare quindi la scalabilità e manutenibilità del progetto, semplificando, sul lungo termine, la gestione e lo sviluppo del front-end.

In questo lavoro di tesi, lo scopo è quello di approfondire la storia dell'evoluzione delle architetture web, in particolare quelle front-end, analizzando le motivazioni tecnologiche che hanno portato allo sviluppo dell'architettura a micro front-end.

L'obiettivo è quello di fornire una panoramica il più completa possibile di quelle che sono le possibili modalità di implementazione e progettazione di un progetto con un'architettura a micro front-end, esaminandone caso per caso punti di forza e di

debolezza; sarà quindi sviluppato un applicativo web utilizzando il framework Astro, così da poterne analizzare i suoi punti di forza.

Inoltre, con lo scopo di approfondire delle possibilità concrete di impiego di tale architettura, saranno studiati alcuni casi di successo di alcune delle grandi aziende del mondo digitale, come Netflix e Spotify, che hanno adottato l'architettura a micro front-end per i loro progetti, sviluppando dei framework interni all'azienda stessa.

Sommario

1. Introduzione

1.1 Contestualizzazione del tema di ricerca

Nel corso degli ultimi anni, il panorama delle architetture web ha subito notevoli cambiamenti grazie all'introduzione di nuove tendenze e nuove tecnologie che hanno rivoluzionato il modo in cui le applicazioni web vengono sviluppate e gestite.

All'interno di questo scenario in continua evoluzione, l'architettura a micro front-end, così come è stato per l'architettura a microservizi relativamente allo sviluppo back-end, sta emergendo come una delle principali tecnologie per lo sviluppo di applicazioni web scalabili e modulari.

L'obiettivo principale dell'architettura a micro front-end è quello di suddividere il software in piccole porzioni modulari per aumentarne la flessibilità e la scalabilità del progetto, semplificarne la manutenzione e la gestione, e consentire lo sviluppo, la manutenzione e la distribuzione delle singole funzionalità dell'applicazione in modo indipendente l'una dall'altra. Tutto ciò permette anche di utilizzare diverse tecnologie per lo sviluppo delle singole porzioni di applicazione modulari.

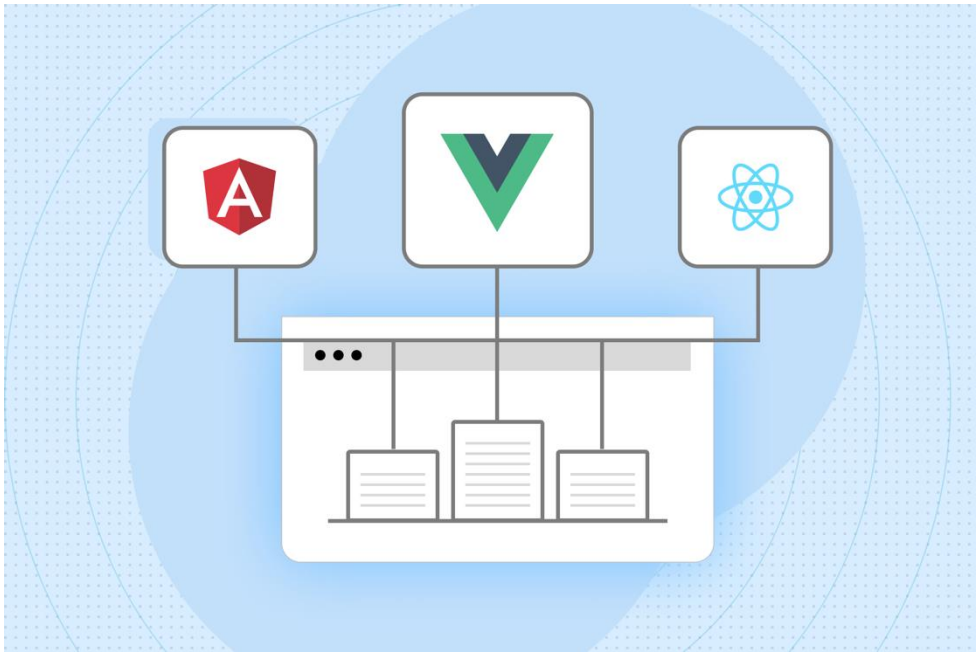


Figura 1.1 esempio di uno schema di una pagina web con architettura a micro front-end

Nel corso degli ultimi anni, infatti, l'aumento della complessità delle applicazioni web ha portato gli sviluppatori ad acquisire competenze sempre più specializzate e settorializzate, determinando quindi la possibilità di utilizzare diverse tecnologie a seconda delle necessità specifiche del progetto. Tuttavia, senza l'utilizzo di un'architettura opportuna come quella a micro front-end, le scelte tecnologiche di un team vincolano necessariamente la possibilità di scelta degli sviluppatori, portando quindi a una restrizione delle possibilità di sviluppo e alla coerenza del progetto.

L'architettura a micro front-end è stata sviluppata in risposta alla crescente adozione dell'architettura a microservizi, che ha dimostrato l'importanza e la necessità di avere una struttura del progetto front-end altrettanto scalabile e modulare, in maniera tale da permettere la suddivisione del software in porzioni modulari e indipendenti.

Inoltre, durante questo decennio, sono state introdotte diverse tecnologie di sviluppo web come React, Angular, Vue.js, che hanno permesso di sviluppare applicazioni web sempre più performanti e scalabili. Queste tecnologie, infatti, hanno introdotto un approccio modulare e component-based, che ha permesso di realizzare applicazioni web complesse e scalabili in modo molto efficiente.

Nonostante ciò, senza l'architettura a micro front-end, queste tecnologie hanno portato alla creazione di applicazioni monolitiche, dove i diversi moduli sono integrati tra di loro e sono necessariamente sviluppati con la stessa tecnologia di base, con ovvi vincoli circa la modalità di sviluppo e deploy all'interno dei vari ambienti di sviluppo.

L'architettura a micro front-end consente invece di mantenere l'indipendenza tra i diversi moduli, garantendo dunque flessibilità e scalabilità del progetto, sia dal punto di vista dello sviluppo che dal punto di vista della pubblicazione e gestione.

Alcune grandi aziende del mondo digitale, come Netflix, Spotify e Amazon, negli ultimi anni hanno adottato l'architettura a micro front-end per gestire la complessità delle loro applicazioni web, sviluppando team specializzati nel front-end e investendo in tecnologie e strumenti per sviluppare e gestire l'architettura a micro front-end. Tale scelta ha permesso loro di migliorare la scalabilità, la manutenibilità e l'esperienza dell'utente, fornendo contemporaneamente una maggiore autonomia e flessibilità ai vari team di sviluppo.

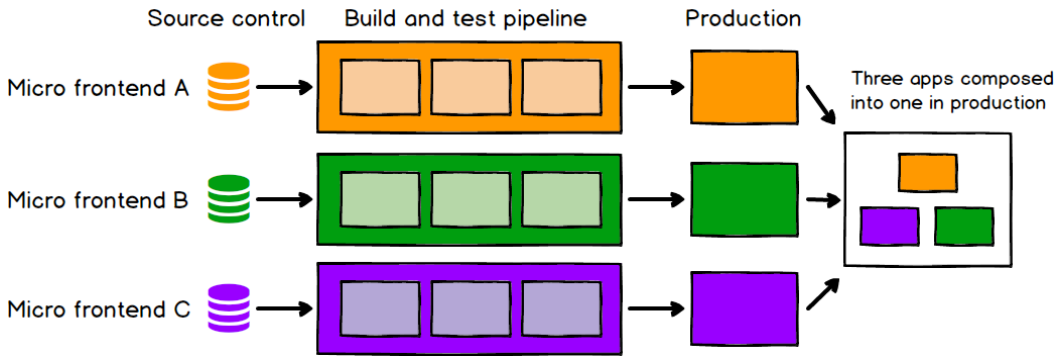


Figura 1.2 Workflow semplificati del processo di creazione e distribuzione di un'applicazione con architettura a micro front-end

1.2 Obiettivo della tesi

L'obiettivo di questo lavoro di tesi di Laurea Magistrale è quello di fornire una panoramica completa delle possibili modalità di implementazione di un'applicazione web basata sull'architettura a micro front-end, analizzando le potenzialità e le criticità di tale architettura che sta rivoluzionando il modo di immaginare e progettare il software per il World Wide Web.

Un punto fondamentale di tale lavoro di ricerca sarà l'analisi di come il Web si è evoluto dal 6 Agosto 1991, data in cui Tim Berners-Lee pubblicò il primo sito web presso il CERN, fino ad oggi. Durante questo percorso, saranno dunque analizzate i vari progressi tecnologici e le tappe fondamentali che hanno determinato la creazione delle tecnologie di sviluppo web che utilizziamo oggi.

Particolare attenzione sarà dedicata allo sviluppo front-end, che rappresenta un elemento chiave nella creazione di applicazioni web moderne e che è il focus principale del lavoro di tesi: verranno quindi esaminati i cambiamenti intervenuti nello sviluppo delle interfacce utente, sia dal punto di vista degli strumenti e dei vari framework introdotti che dal punto di vista delle architetture web. L'analisi di queste tecnologie permetterà dunque di comprendere le opportunità e le sfide implementative dell'architettura a micro front-end.

Successivamente, dopo aver analizzato l'architettura a microservizi a cui l'architettura a micro front-end si ispira, si procederà con un'attenta analisi delle caratteristiche implementative dell'architettura a micro front-end e di come viene effettuata l'integrazione

dei diversi micro front-end, ponendo particolare attenzione a quelle che sono le potenzialità e le criticità di tale architettura, con lo scopo dunque di individuare quelle che potrebbero essere delle best practice legate al suo utilizzo.

Seguirà quindi una sezione implementativa all'interno della quale verrà svolta l'analisi di alcuni framework, insieme allo sviluppo di un'applicazione web basata sull'architettura a micro front-end mediante l'utilizzo di del framework Astro, un framework che permette di sviluppare applicazioni a micro front-end basandosi sull'Island Architecture, un particolare pattern per la costruzione delle pagine web.

In conclusione, verranno studiati alcuni casi di successo di grandi aziende come ad esempio Spotify e Netflix, che hanno adottato l'architettura a micro front-end per le proprie applicazioni, anche mediante framework progettati internamente, con lo scopo di capire come tale architettura venga utilizzata all'interno di contesti reali e per comprendere quali siano le sfide e le soluzioni adottate sulla base dei casi specifici.

2. Archeologia dello sviluppo web

Nel corso degli ultimi decenni, le tecnologie per lo sviluppo web sono state caratterizzate da una rapida evoluzione che ha trasformato radicalmente le modalità di realizzazione dei contenuti web e l'esperienza utente online. Dal World Wide Web degli anni '90 fino alle tecnologie moderne di sviluppo, sono state infatti introdotte numerose innovazioni con il fine di migliorare la velocità e l'usabilità del web, sia dal punto di vista del consumatore ultimo che dal punto di vista dello sviluppo vero e proprio, fino ad arrivare, relativamente allo sviluppo front-end, alla recente ideazione dell'architettura a micro front-end, una metodologia di sviluppo che prevede la suddivisione dell'interfaccia utente in componenti modulari completamente indipendenti tra di loro.

L'architettura a micro front-end, insieme all'architettura a microservizi per il back-end, rappresentano infatti un'importantissima evoluzione per le tecnologie web, in quanto hanno determinato un cambiamento significativo nella progettazione e nello sviluppo delle applicazioni web.

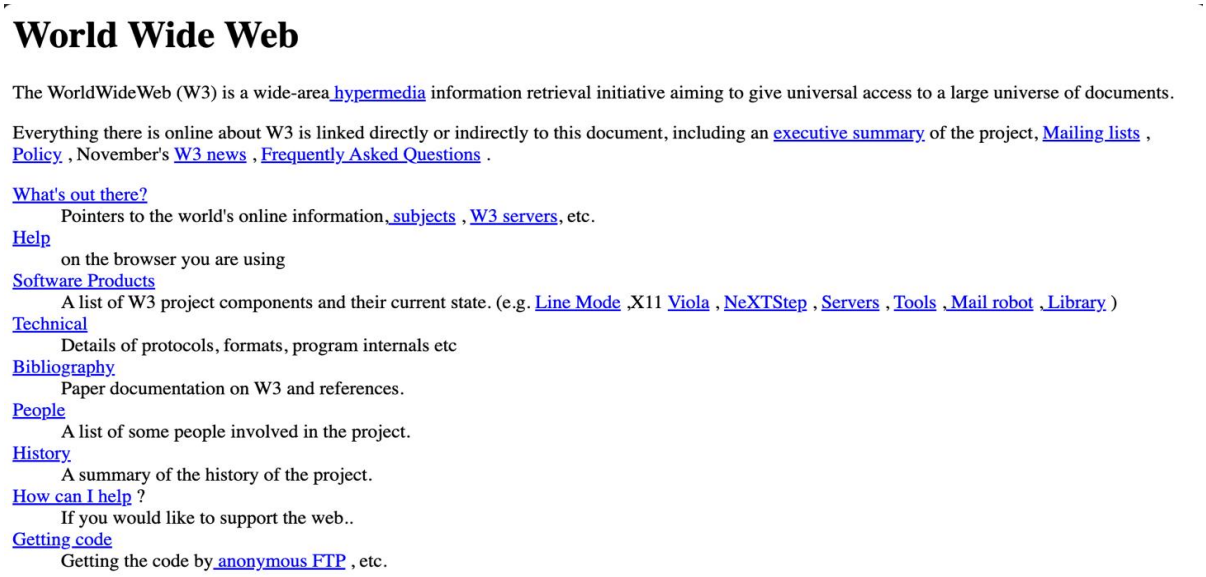
Prima di addentrarsi nel funzionamento dell'architettura a micro front-end, è però importante comprendere come le tecnologie web si siano evolute nel corso degli anni, dalla nascita del web alla realizzazione delle tecnologie avanzate che si utilizzano oggi nella realizzazione delle interfacce web.

2.1 Anni '90: Le origini del web

Il primo sito web della storia è datato 1991 ed è nato all'interno di un progetto del CERN (Organizzazione Europea per la ricerca nucleare) grazie ad un progetto di Tim Berners-Lee che aveva l'obiettivo di creare uno spazio di ricerca e di condivisione di documenti tra gli scienziati. L'obiettivo di Tim Berners Lee, infatti, era quello di creare un sistema di documenti collegati tra di loro tramite collegamenti ipertestuali, in cui ogni documento era univocamente identificato dalla URL (Uniform Resource Locator), e ogni documento poteva contenere riferimenti ad altri documenti con l'obiettivo di creare una possibile navigazione non lineare tra i documenti stessi.

La creazione del primo sito web, che conteneva le informazioni circa il progetto World Wide Web e che tutt'oggi è ancora visitabile tramite la seguente URL:

http://info.cern.ch/hypertext/WWW/TheProject.html, rappresenta una pietra miliare dello sviluppo web, poiché ha dato il via alla creazione dell'enorme infrastruttura ad oggi esistente.



World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

Figura 2.1 Screenshot del primo sito web della storia

Nel corso di questi anni le tecnologie web iniziano ad emergere come uno dei principali strumenti di comunicazione e di condivisione di informazioni, con il delinearsi delle tecnologie e dei linguaggi di programmazione che saranno e sono ancora oggi la base dello sviluppo web come HTML, CSS e Javascript, che rappresentano la base delle applicazioni web e che hanno aperto la strada verso una vasta gamma di innovazioni e applicazioni nel mondo digitale.

2.1.1 HTML (Hypertext Markup Language)

L'HTML (Hypertext Markup Language) è il linguaggio di markup standard utilizzato per la creazione delle pagine web, ed è stato sviluppato da Tim Berners Lee stesso negli anni 90 per realizzare un linguaggio che potesse essere comprensibile da parte di qualsiasi computer collegato alla rete.

L'HTML ha dunque lo scopo di definire la struttura e il contenuto di un documento web, tramite l'utilizzo di una serie di tag che descrivono la struttura vera e propria del documento e il modo in cui deve essere visualizzato all'interno della pagina.

Esso permette quindi di inserire all'interno del documento blocchi di testo, immagini, video, audio e altri elementi multimediali sulla pagina web tramite dei tag specifici, oltre che a definire la struttura e il layout della pagina.

Di seguito è presentata la struttura base di un documento HTML, contenente gli elementi peculiari di una pagina web.

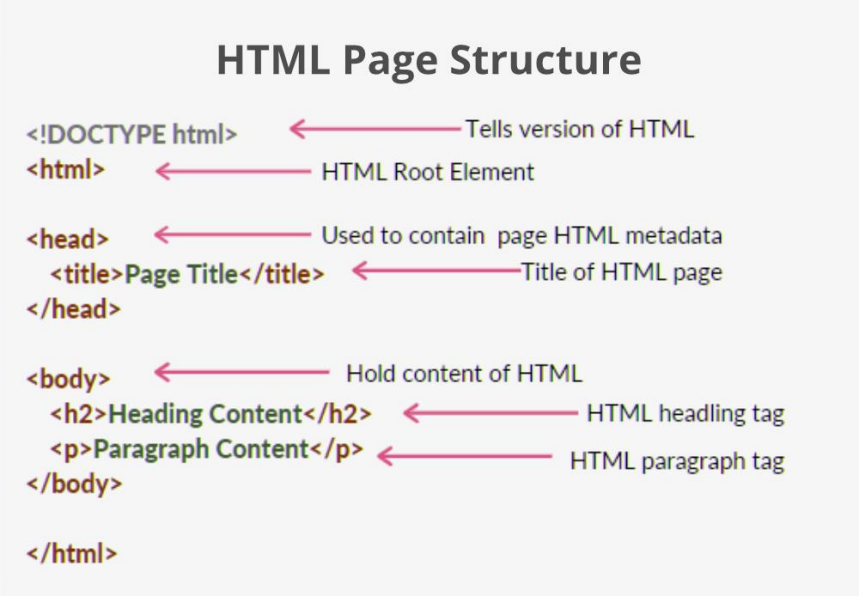


Figura 2.2 Struttura base di un documento HTML

I tag fondamentali all'interno di un documento HTML sono i seguenti:

- *<!DOCTYPE html>:* il tag Doctype è un'istruzione che indica al browser web il linguaggio di markup utilizzato per la realizzazione della pagina corrente;
- *<html>:* questo tag definisce la radice del documento html stesso;
- *<head>:* tag all'interno del quale vanno definite quelle che sono le informazioni relative al documento stesso, come ad esempio il titolo e i metadata presenti.
- *<body>:* tag contenente tutti gli elementi che saranno poi visibili in pagina e che verranno mostrati sul front-end.

L'HTML, inoltre, è un linguaggio di markup statico, per cui la pagina web viene creata in base al codice HTML e non può essere modificata e manipolata dall'utente: per poter inserire delle animazioni o per poter introdurre degli elementi interattivi all'interno della pagina, è necessario utilizzare degli altri strumenti, come il CSS (Cascading Style Sheet) e degli script Javascript.

2.1.2 CSS (Cascading Style Sheet)

Il CSS Cascading Style Sheet è un linguaggio di stile utilizzato per definire in maniera più dettagliata rispetto al semplice HTML l'aspetto e il layout degli elementi appartenenti a una pagina Web definiti all'interno del documento HTML.

Esso è stato introdotto nel 1996 dal World Wide Web Consortium (W3C), l'organizzazione che si occupa di sviluppare standard per il World Wide Web, con lo scopo di separare la presentazione di una pagina web dal suo contenuto strutturale.

Prima dell'avvento del CSS, infatti, le pagine web erano scritte principalmente in HTML, e le modalità di presentazione dei contenuti erano descritte direttamente all'interno del documento HTML stesso, rendendone difficile la manutenzione e l'aggiornamento dato che, ogniqualvolta si voleva modificare la presentazione della pagina, era necessario intervenire sul codice HTML stesso.

L'introduzione del CSS ha permesso quindi di separare la presentazione della pagina web dal contenuto strutturale della stessa, permettendo quindi di avere una maggiore flessibilità e facilità di manutenzione.

Il funzionamento del CSS consiste nella creazione di regole di stile per i singoli elementi che compongono la pagina web nel documento HTML. Tali stili possono essere inseriti direttamente inline all'interno del documento HTML, ma è buona prassi utilizzare una sezione dedicata allo stile all'interno della pagina (inglobando le regole all'interno del tag <style>) o, soluzione preferibile in termini di manutenibilità, utilizzare un foglio di stile esterno di cui si effettua l'import.

Per quanto riguarda la sintassi del CSS, invece, il selettore ha lo scopo di specificare gli elementi della pagina web a cui dovranno essere applicate le regole di stile, e all'interno della regola si definiscono le singole proprietà con i loro valori. Inoltre, il CSS utilizza il concetto di "cascata", che permette di definire diverse regole per gli stessi elementi della pagina web stabilendo delle priorità di applicazione delle regole, sulla base del selettore, dell'ordine di dichiarazione e dell'ereditarietà dei valori degli stili.

```css
h1 {
    font-family: courier, courier-new, serif;
    font-size: 20pt;
    color: blue;
    border-bottom: 2px solid blue;
}
p {
    font-family: arial, verdana, sans-serif;
    font-size: 12pt;
    color: #6B6BD7;
}
.red_txt {
    color: red;
}
```

Figura 2.3 struttura di un file CSS

2.1.3 Javascript

Per avere la possibilità di inserire elementi di interattività per l'utente all'interno delle pagine web, nel 1995 viene sviluppato da Brendan Eich il linguaggio di programmazione Javascript, che nasce inizialmente per un utilizzo limitato all'ambito del web development per la realizzazione di interfacce web interattive, ma che oggi è diventato uno dei linguaggi più utilizzati al mondo e che viene utilizzato non soltanto per lo sviluppo front-end, ma anche per lo sviluppo back-end e per molti altri scopi.

Il linguaggio Javascript è stato progettato per essere un linguaggio di scripting leggero, inizialmente destinato a funzionare esclusivamente lato client, senza quindi la necessità di un server di applicazioni, all'interno del browser web principalmente per la validazione di form e per le animazioni semplici: tale linguaggio di programmazione, quindi, è nato proprio con lo scopo di manipolare il DOM (Document Object Model) delle pagine web, ovvero la rappresentazione del contenuto HTML della pagina web.

L'esecuzione del linguaggio Javascript direttamente lato client, infatti, è fondamentale per lo sviluppo web in quanto elimina la necessità di richiedere un server di applicazioni che esegua il codice lato server e che invii i dati al browser, determinando dunque la possibilità di creare pagine estremamente reattive, in cui gli utenti possono interagire con l'applicazione browser che si occupa di gestire real-time le interazioni dell'utente, senza il passaggio sul server.

La maggior parte dei primi siti web erano ospitati su server statici, che non erano in grado di elaborare codice dinamico o di eseguire operazioni complesse: ciò significa che tutte le operazioni di elaborazione dei dati dovevano essere eseguite lato client, utilizzando linguaggi di scripting come JavaScript, che all'epoca era in fase iniziale di sviluppo.

Oggi, circa il 95,2% dei siti web, corrispondente 1,52 miliardi di siti web, utilizzano Javascript come linguaggio di programmazione per lo sviluppo.

2.2 Anni 2000 – Pagine dinamiche, framework e librerie

Durante i primi anni 2000, il web vive una fase di rapida evoluzione tecnologica e architetturale: in questo periodo, infatti, Javascript si afferma come linguaggio di programmazione del web e vengono introdotti diverse librerie e diversi framework che hanno permesso di creare applicazioni web dinamiche e interattive in maniera molto più efficiente.

Una libreria è un insieme di codice predefinito che può essere importato all'interno del progetto e utilizzato per eseguire delle specifiche operazioni, con lo scopo di aiutare gli sviluppatori nella fase di sviluppo dell'applicazione web, fornendo un insieme di funzioni e metodi che possono essere utilizzati per semplificare la scrittura del codice.

Una delle principali librerie utilizzate durante l'inizio del terzo millennio è jQuery, una libreria JavaScript open-source sviluppata nel 2006 da John Resig, che ha la funzione di semplificare l'interazione con il DOM aggiungendo, tra le altre cose, elementi di interattività all'interno della pagina.

Tramite jQuery, gli sviluppatori hanno avuto per la prima volta la possibilità di selezionare gli elementi del DOM tramite una sintassi semplificata, creare animazioni, effettuare chiamate AJAX e altro ancora tramite una sintassi semplificata rispetto a Javascript "puro", permettendo quindi di ridurre in maniera significativa il codice necessario per la creazione delle applicazioni web, spianando quindi la via verso librerie Javascript più avanzate e complesse, come ad esempio React.

I punti di forza che hanno determinato l'affermarsi delle librerie Javascript come jQuery rispetto all'utilizzo di Javascript puro sono i seguenti:

- Semplificazione della sintassi: le librerie Javascript semplificano la sintassi Javascript standard, agevolando quindi l'interazione con il DOM e con gli elementi all'interno della pagina e la gestione degli eventi scaturiti dall'interazione degli utenti;
- Compatibilità cross-browser: le librerie come jQuery offrono un'API unificata per l'interazione con il DOM funzionante per tutti i browser, eliminando quindi il problema della compatibilità tra i vari browser;
- Ampia documentazione e supporto: spesso dietro allo sviluppo delle librerie vi è una vasta community che alimenta la documentazione della libreria stessa, fornendo suggerimenti e chiarimenti ai vari membri della community all'interno degli spazi di condivisione sul web;
- Utilizzo di plugin: jQuery, ad esempio, è stata utilizzata per la realizzazione di plugin e widget personalizzabili, come ad esempio slider, menù e caroselli, che possono essere facilmente integrabili all'interno delle pagine web semplificando quindi lo sviluppo di elementi avanzati come quelli descritti;
- Performance: molto spesso le librerie sono ottimizzate per offrire prestazioni più elevate in termini di caricamento delle pagine, rendendo le applicazioni web più veloci e reattive.

Le librerie, quindi, permettono di avere un insieme di funzioni e metodi per eseguire delle operazioni specifiche all'interno dell'applicazione web.

I framework, invece, offrono un'architettura completa per lo sviluppo web, fornendo una struttura di base per lo sviluppo, definendo un'architettura e uno schema ben definito: ciascun framework, infatti, assume una serie di convenzioni di default per la struttura dell'applicazione e includendo una serie di strumenti per la gestione della logica di business, dell'interfaccia utente, della gestione dei dati, della sicurezza e dell'accesso ai servizi web.

Uno dei primi framework più utilizzati nello sviluppo delle applicazioni web è stato Ruby on Rails, sviluppato nel 2004, un framework open source scritto in Ruby che ha introdotto per la prima volta l'approccio *"convention over configuration",* un principio di design del software secondo cui bisogna ridurre il numero di decisioni che lo sviluppatore che sta utilizzando il framework deve prendere, limitando le decisioni dello sviluppatore relativamente allo sviluppo della logica dell'applicazione, mentre il framework si deve

occupare delle parti ripetitive e convenzionali senza dover quindi richiedere una configurazione esplicita per ogni aspetto del software.

Ruby on Rails ha reso lo sviluppo di applicazioni molto più rapido ed efficiente, in quanto includeva diverse funzionalità predefinite per semplificare una moltitudine di operazioni, come ad esempio l'interazione con i database e la gestione della migrazione dei dati, ispirando lo sviluppo di altri framework web come ad esempio Spring e Laravel.

2.3 Architettura client-server

L'architettura client-server è la prima architettura utilizzata nella storia del web development per la realizzazione di applicazioni web. In questa architettura, le funzionalità dell'applicazione web sono suddivise in due componenti principali: il client e il server.

Il client, che è il software eseguito sul dispositivo dell'utente, si occupa della user interface dell'applicazione web, utilizzando il browser per gestire quindi la visualizzazione dell'applicazione web e le interazioni con l'utente.

Il server, d'altra parte, è responsabile della gestione della logica dell'applicazione e dei dati che vengono visualizzati in pagina. In particolare, il server fornisce le risposte alle richieste del client, gestisce la sicurezza dell'applicazione e si interfaccia con i database e/o altri sistemi esterni.

Nell'architettura client-server, il client e il server comunicano attraverso una rete (generalmente una rete Internet) utilizzando il protocollo HTTP (Hypertext Transfer Protocol): il client invia una richiesta http al server, che a sua volta risponde al client con una risposta HTTP, che contiene informazioni quali l'indirizzo della pagina web richiesta, i dati dell'utente e le informazioni di sessione.

2.3.1 Il client

In un'architettura client-server, il client è il software o l'applicazione che viene eseguita dal dispositivo di fruizione dell'utente finale e che interagisce con il server per accedere ai dati o alle altre funzionalità offerte dall'applicazione web.

Le funzionali del client sono diverse, tra cui:

1) Fornire un'interfaccia utente che consenta all'utente di interagire con l'applicazione web;
2) Gestire gli input dell'utente all'interno della pagina web;
3) Inviare le richieste al server per ottenere o eseguire delle operazioni specifiche, come ad esempio cambiare pagina all'interno dell'applicazione o richiedere/caricare dati;
4) Ricevere, analizzare e presentare le risposte dal server all'utente;
5) Memorizzare i dati dell'applicazione sul dispositivo dell'utente;

Le tecnologie utilizzate per la realizzazione del client dipendono dalle esigenze e dalle scelte degli sviluppatori, tuttavia, per quanto riguarda lo sviluppo web, le principali utilizzate sono l'HTML per la creazione della struttura della pagina, il CSS per definire lo stile degli elementi contenuti all'interno della pagina web e Javascript, associato a framework e/o librerie, per implementare la logica dell'applicazione e le interazioni con il server.

2.3.2 Il server

In un'architettura client-server, il server è il componente che gestisce le richieste dei client e fornisce i dati e le funzionalità dell'applicazione web.

Le funzioni del server sono diverse, tra cui:

1) Gestire ed elaborare le richieste HTTP dei client inviate tramite la rete internet;
2) Fornire i dati e le risorse richieste dai client, come ad esempio le pagine web, i file multimediali, i dati dal database o le API;
3) Eseguire operazioni specifiche richieste dai client, come ad esempio l'invio di e-mail, l'interazione con il database o l'elaborazione di pagamenti;
4) Gestire la sicurezza e l'autenticazione dei client che accedono all'applicazione web usando metodi di autenticazione e autorizzazione;
5) Mantenere lo stato dell'applicazione e dei dati tra le diverse richieste dei client, ad esempio utilizzando tecniche di sessione o di caching.

2.3.3 Comunicazione client-server: HTTP

Il protocollo di comunicazione utilizzato per lo sviluppo di applicazioni web è HTTP, Hyper Text Protocol, un protocollo di comunicazione per il trasferimento di dati su internet, in

particolare per il recupero di risorse web come pagine HTML, file, immagini, dati e molto altro.

Tale protocollo è stato sviluppato agli inizi degli anni ’90 da Tim Berners Lee stesso con lo scopo di creare un protocollo di comunicazione standardizzato per la comunicazione tra server e client, in maniera tale da definire come e con quale formato dovesse avvenire la comunicazione tra i due elementi. Inizialmente, HTTP supportava solamente il recupero di pagine statiche, ma in seguito è stato esteso per supportare anche il trasferimento di dati dinamici all’interno di applicazioni web interattive.

Le applicazioni web, infatti, sono costituite da un insieme di pagine web che vengono trasmesse al client in seguito a una comunicazione tra client e server che avviene attraverso il protocollo HTTP.

Il protocollo HTTP si basa essenzialmente sul modello di richiesta/risposta: il client, ad esempio il browser web, invia una richiesta HTTP al server, che a sua volta risponde al client con una risposta HTTP.

La richiesta HTTP, in particolare, contiene:

1) Il metodo HTTP, che specifica il tipo di azione che il client desidera eseguire sulle risorse del server;
2) L’URL (Uniform Resource Locator), che consiste nell’indirizzo web che consente di individuare la risorsa su internet ed è composta da diversi parametri quali il protocollo di comunicazione, il nome del server, il percorso della risorsa e, se necessario, dei parametri aggiuntivi;
3) Eventuali dati della richiesta.

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:92.0) Gecko/20100101 Firefox
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Figura 2.4 GET HTTP

La risposta del server, invece, contiene:

1) Lo stato della richiesta (ad esempio 200 per una risposta valida, 404 per una risorsa non trovata);
2) I dati della risposta;
3) Le informazioni di intestazione.

```
HTTP/1.1 200 OK
Date: Wed, 23 Feb 2023 12:00:00 GMT
Server: Apache/2.4.48 (Unix)
Last-Modified: Tue, 22 Feb 2023 10:00:00 GMT
ETag: "abcdef1234567890"
Content-Type: text/html
Content-Length: 1234
Connection: close
```

```
<!DOCTYPE html>
<html>
<head>
<title>Example Website</title>
</head>
<body>
<h1>Welcome to the Example Website!</h1>
<p>This is an example web page.</p>
</body>
</html>
```

Figura 2.5 Risposta HTTP

I passaggi fondamentali che caratterizzano il funzionamento della comunicazione HTTP all'interno di un'architettura client-server in ambito web sono i seguenti:

1) Il client invia una richiesta HTTP al server: tale richiesta contiene l'indirizzo URL (Uniform Resource Locator) della pagina web richiesta e altre informazioni come i cookie e i parametri di query, se necessari;
2) Il server riceve la richiesta HTTP e cerca la pagina web richiesta all'interno della sua infrastruttura dati. Trovata la pagina, il server genera una risposta HTTP contenente il codice di stato HTTP e il contenuto della pagina web richiesta;
3) Il server invia la risposta HTTP al client tramite la rete;

4) Il client riceve la risposta HTTP e ne analizza il codice di stato e il contenuto: se il codice di stato indica che la richiesta è andata a buon fine, allora il client analizza il contenuto della pagina web e lo visualizza all'interno del browser.

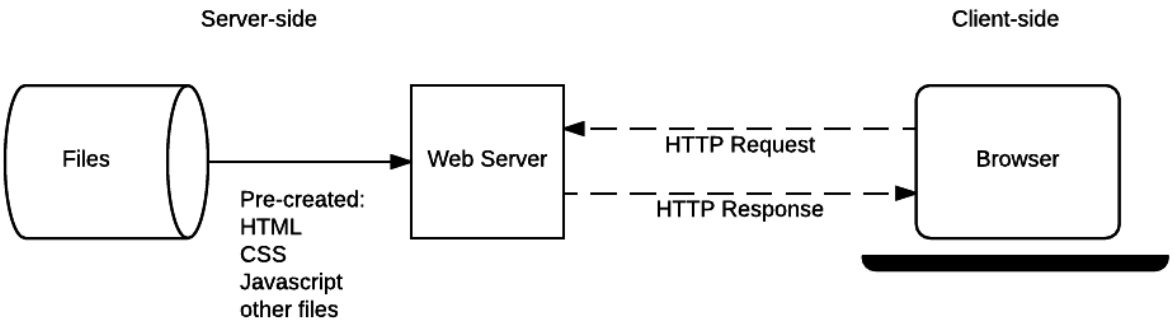


Figura 2.6 Schema comunicazione HTTP

3. Architetture web

Con il termine “architettura” in ambito informatico, ci si riferisce alla struttura complessiva di un software e alle modalità tramite cui i componenti dell’applicativo comunicano tra di loro, rappresentando quindi la struttura di base su cui verranno implementati i vari elementi dell’applicazione.

La definizione dell’architettura di un’applicazione web è un processo fondamentale in quanto influenza tutte le fasi del ciclo di vita dell’applicazione stessa, dalla fase di sviluppo alla fase effettiva di utilizzo da parte dell’utente finale. Essa costituisce la base per ogni elemento dell’applicazione web, andando ad influenzare la qualità del prodotto finito: inevitabilmente, un’applicazione con una pessima architettura in termini di progettazione avrà un impatto negativo sulla realizzazione e l’utilizzo dell’applicazione stessa secondo diversi punti di vista.

Il modello architetturale di tipo client-server per sistemi distribuiti, ovvero sistemi informatici costituiti da processi interconnessi tra di loro, è quello dell’architettura multistrato, in cui le varie funzionalità del software sono suddivise su diversi strati in comunicazione tra loro.

La separazione dell’applicazione in più livelli permette di avere un modello per sviluppare in maniera vantaggiosa applicazioni flessibili e scalabili, permettendo di suddividere le funzionalità su vari livelli specifici e di effettuare modifiche sui singoli livelli senza dover andare a modificare necessariamente l’applicazione nella sua interezza, andando quindi a garantire una maggiore semplicità di progettazione e implementazione.

3.1 Architettura three-tier

L’architettura Three-Tier è il modello di architettura software principalmente utilizzato nell’ambito dello sviluppo web.

Nel modello dell’architettura Three-Tier, l’applicazione è organizzata in tre livelli logici di elaborazione principali indipendenti tra di loro ma comunicanti, ciascuno dei quali svolge una specifica funzione: il livello di presentazione, il livello di applicazione e il livello dei dati.

Il livello di presentazione, chiamato anche “livello di interfaccia utente”, è il livello più alto dell’applicazione, e si occupa della presentazione delle informazioni e dell’interfaccia utente al consumatore finale, nonché di gestire gli eventi e gli input dell’utente stesso. Il livello di presentazione è quello che viene eseguito, nel caso delle applicazioni web, all’interno del browser web.

Il livello di applicazione, chiamato business logic tier, si occupa della logica dell’applicazione stessa, elaborando le informazioni che vengono raccolte nel livello di presentazione, validando i dati inseriti dagli utenti e gestendo le autorizzazioni dei consumatori finali.

Il livello di dati, anche chiamato livello di persistenza, è il livello più basso dell’architettura three-tier, e si occupa di gestire la creazione, l’aggiornamento e l’eliminazione dei dati dell’applicazione stessa, gestendo l’accesso ai sistemi di archiviazione dei dati.

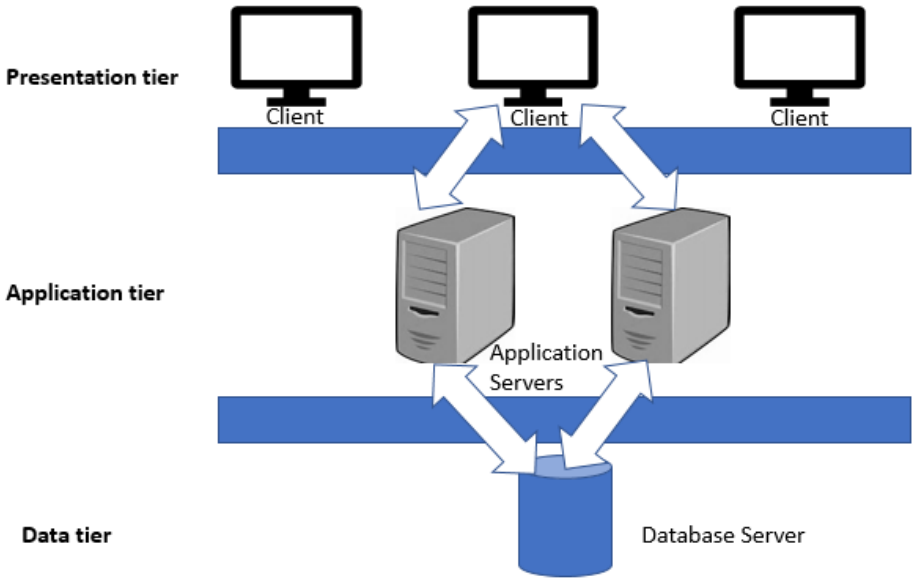


Figura 3.1 Schema dell'architettura three tier

L’architettura three-tier si è consolidata come modalità di progettazione principale delle architetture web in quanto presenta numerosi vantaggi in termini di sviluppo e gestione dell’applicazione web:

1) Separazione logica e fisica delle funzionalità: ogni livello può essere eseguito su un sistema operativo o su un server separato, che viene scelto sulla base dei requisiti

funzionali dell'applicazione. Inoltre, ogni livello viene eseguito su server e hardware dedicati, in modo che le funzioni di ciascun livello si possano personalizzare e ottimizzare senza impatti sugli altri;

2) Benefici sulla fase di sviluppo: ogni livello può essere sviluppato simultaneamente da team diversi, permettendo quindi di portare l'applicazione sul mercato in tempi ridotti e di utilizzare, per ciascun livello, linguaggi e tecnologie diverse. La separazione dei livelli, inoltre, permette di avere una maggiore flessibilità nell'aggiornamento o nell'aggiunta di nuove funzionalità;
3) Scalabilità: suddividendo l'applicazione in tre parti distinte, ciascuna parte può essere gestita e scalata separatamente, permettendo di distribuire i livelli dell'applicazione su server diversi, aumentando la capacità di gestione dell'applicazione, migliorandone le prestazioni e ottimizzandone l'utilizzo delle risorse distribuendo il carico di lavoro in maniera più efficiente;
4) Affidabilità: la separazione dei livelli consente di isolare i problemi in modo più efficace, senza la necessità di bloccare l'intera applicazione per risolvere una problematica legata a un singolo livello;
5) Sicurezza: la suddivisione in livelli permette di applicare misure di sicurezza e politiche di accesso specifiche per ciascun livello. Il livello di presentazione, ad esempio, potrebbe prevedere delle misure di sicurezza per prevenire degli accessi non autorizzati; il livello di applicazione potrebbe implementare dei controlli di validazione, mentre il livello dei dati potrebbe venire protetto con delle misure di sicurezza per prevenire l'accesso non autorizzato ai dati.

Dopo aver analizzato i vantaggi e il funzionamento dell'architettura a tre strati, è dunque importante effettuare un ulteriore approfondimento circa le architetture dei singoli strati e come esse si siano evolute nel corso degli anni con l'evoluzione di internet e con l'introduzione di nuove tecnologie di sviluppo. In particolare, per quanto riguarda il business logic tier, le due architetture fondamentali sono l'architettura monolitica e l'architettura a microservizi, mentre per il presentation layer verranno posta particolare attenzione verso le Multi Page Application (MPA), le single Page Application (SPA) e l'architettura a micro front-end.

3.2 Dall'architettura monolitica ai microservizi

L'evoluzione delle metodologie di sviluppo software ha, nel corso degli anni, portato a una grande trasformazione nella progettazione e nella realizzazione delle architetture back-end utilizzate in ambito web, che hanno poi avuto importanti conseguenze anche sullo sviluppo delle architetture front-end.

In passato, infatti, lo sviluppo software è stato caratterizzato da un approccio monolitico, in cui i singoli componenti dell'applicazione venivano sviluppati come se fossero un'unica entità. Negli ultimi anni, le innovazioni tecnologiche hanno determinato la nascita di nuovi pattern architetturali per lo sviluppo delle infrastrutture software, come ad esempio l'architettura a microservizi, che ha posto le basi per l'architettura a micro front-end.

3.2.1 Architettura monolitica

L'architettura monolitica rappresenta il modello tradizionale dello sviluppo software, ed è caratterizzato dallo sviluppo delle funzioni del software stesso all'interno di una codebase unificata, come indicato dal termine "monolitico".

L'applicazione all'interno di un'applicazione che segue tale modello architetturale, infatti, è sviluppata come se fosse un'unica unità di codice, che include tutte le funzionalità dell'applicazione. Le funzioni del software monolitico, infatti, sono strettamente legate tra loro facendo parte di un'unica grande applicazione.

Le applicazioni monolitiche sono sviluppate e distribuite come se fossero un'unica applicazione, indipendentemente dal modo in cui l'utente finale la utilizza, e i moduli dell'applicazione stessa sono strettamente legati tra di loro.

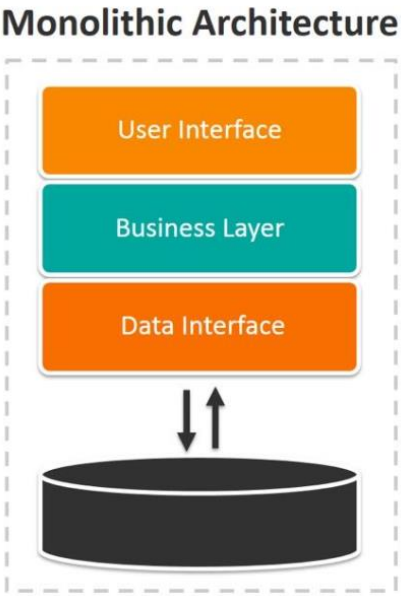


Figura 3.2 Schema dell'architettura monolitica

Per comprendere il funzionamento dell'architettura monolitica, può essere utile prendere come esempio un'applicazione web di un ipotetico e-commerce: tale applicazione dovrà occuparsi di autorizzare gli utenti, effettuare il log-in all'area personale e permettergli di trasferire denaro per acquistare degli oggetti direttamente dall'inventario del negozio; all'interno di questo processo, sono coinvolti un grande numero di altri componenti, sia dal punto di vista della presentazione che dal punto di vista dell'interazione con il database e con i servizi.

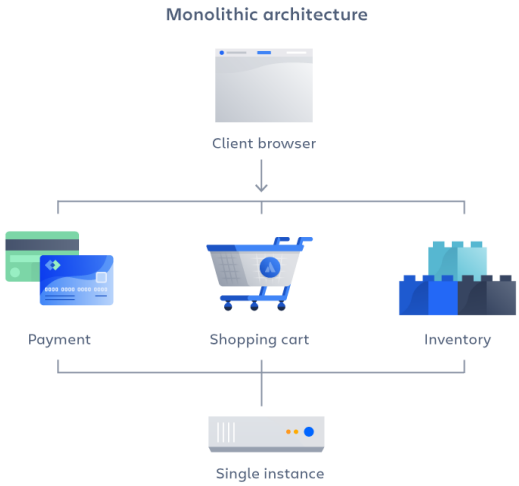


Figura 3.3 Ipotetico modello di un e-commerce con architettura monolitica

Se l'applicazione è progettata seguendo il pattern dell'architettura monolitica, essa sarà sviluppata e distribuita come se fosse una singola applicazione, a prescindere da come l'utente la utilizzerà, sia che acceda da un dispositivo mobile che da un dispositivo desktop.

Dal punto di vista della distribuzione e dello sviluppo, invece, facendo parte di un'unica codebase i componenti dovranno condividere le scelte tecniche, come ad esempio il linguaggio di programmazione utilizzato e/o i framework per lo sviluppo scelti, e nel caso in cui dovesse essere necessaria una modifica su un singolo componente, tale modifica riguarderà inevitabilmente anche gli altri, soprattutto dal punto di vista della distribuzione.

Quando si sviluppa utilizzando il pattern dell'architettura monolitica, il vantaggio principale è la velocità di sviluppo, dovuta alla semplicità di avere un'applicazione basata su un'unica codebase.

L'architettura monolitica viene spesso utilizzata per applicazioni di piccole e medie dimensioni, dove la complessità dell'applicazione non richiede la suddivisione in moduli separati, semplificando quindi le modalità di sviluppo e di distribuzione dell'applicazione, in quanto essa viene distribuita all'interno di un singolo pacchetto. Per queste tipologie di applicazioni, l'unificazione del flusso di lavoro per lo sviluppo, il testing e il rilascio non comporta eccessive problematiche in termini di manutenzione avendo un ristretto numero di componenti.

I vantaggi dell'architettura monolitica, per le applicazioni di piccole dimensioni, includono:

1) Facilità di distribuzione: dato che l'applicazione si basa su una codebase unificata, la distribuzione dell'applicazione è necessariamente semplificata. L'applicazione monolitica, infatti, richiede un singolo processo di esecuzione per l'intera applicazione;
2) Facilità di sviluppo: quando un'applicazione è costruita con un'unica codebase, essa è più facile da sviluppare. Ad esempio, l'utilizzo di una singola codebase semplifica la gestione delle dipendenze, in quanto i pacchetti e le librerie esterne vengono gestite all'interno di un unico file di configurazione;
3) Performance: in una codebase unificata, un'API unica può spesso svolgere la stessa funzione che numerose API svolgono con i microservizi. I componenti dell'applicazione inoltre comunicano attraverso un'interfaccia unificata e un

protocollo comune, riducendo quindi l'overhead di comunicazione tra i diversi componenti dell'applicazione. In termini di risorse, invece, all'interno dell'architettura monolitica i componenti dell'applicazione condividono le stesse risorse.

4) Test semplificati: dato che l'applicazione monolitica consiste di un'unica unità centralizzata, i test end-to-end possono essere eseguiti in modo molto più rapido rispetto a un'applicazione distribuita.
5) Facilità di debugging: poiché il codice si trova tutto all'interno di un'unica repository, è tendenzialmente più semplice andare ad effettuare il debug di un problema.

Tuttavia, nel momento in cui un'applicazione diventa molto grande in termini di componenti e di funzionalità, riuscire ad avere una buona scalabilità diventa una grande sfida. Infatti, per apportare delle singole modifiche per una singola funzione, è necessario compilare e testare l'intera piattaforma, il che va contro l'approccio agile che al giorno d'oggi è preferito dagli sviluppatori.

Infatti, nel caso di applicazioni di grandi dimensioni, i vantaggi dell'architettura monolitica citati precedentemente rischiano di diventare degli svantaggi che influenzano negativamente l'esperienza di sviluppo dell'applicazione.

Per applicazioni di grandi dimensioni, l'architettura monolitica non è dunque la soluzione ideale, in quanto la codebase unificata rende più complicata la manutenzione e la modifica del codice, dato che la modifica di un singolo componente impatta necessariamente gli altri componenti dell'applicazione: ad esempio, all'interno di un ipotetico applicativo per e-commerce, se si volesse effettuare una modifica, e quindi un aumento di versione, al componente che si occupa di autenticare gli utenti, in fase di distribuzione non sarà distribuita semplicemente una nuova versione del componente di autenticazione, ma dovrà essere distribuito un pacchetto unificato che comprenderà, oltre al componente che ha subito la modifica, anche tutti gli altri componenti.

Inoltre, la complessità di un'applicazione aumenta con la dimensione e diventa sempre più difficile da gestire, soprattutto in termini organizzativi all'interno del team di sviluppo, dato che ogni singolo sviluppatore dovrebbe avere una conoscenza trasversale di tutto l'applicativo.

Di seguito un'analisi degli svantaggi dell'architettura monolitica nel caso sopracitato:

1) Rallentamento nella velocità di sviluppo: un'applicazione monolitica di grandi dimensioni rende lo sviluppo più complesso e lento. Infatti, con l'aumentare delle dimensioni dell'applicativo, aumentano necessariamente i tempi di sviluppo e di debugging, dato che i cambiamenti all'interno del codice di un componente possono avere effetti collaterali su altri componenti. Inoltre, i componenti spesso rischiano di diventare altamente interdipendenti, rendendone difficile l'aggiornamento;
2) Effetti negativi sulla developer experience: una grande applicazione monolitica può essere estremamente complessa da comprendere, soprattutto nel caso di sviluppatori che vengono inseriti all'interno del team in una fase avanzata del progetto. Inoltre, la pipeline di sviluppo del software monolitico prevede necessariamente che tutti gli sviluppatori conoscano il funzionamento complessivo dell'applicazione, anche se il lavoro di uno sviluppatore potrebbe essere circoscritto ad alcuni componenti, dato che i componenti stessi sono dipendenti tra loro;
3) Scarsa flessibilità: l'architettura monolitica richiede l'utilizzo di un unico linguaggio di programmazione per l'intero applicativo, dato che quest'ultimo è contenuto all'interno di una codebase unificata. Questa caratteristica, infatti, è poco strategica data la grande varietà tecnologica disponibile sul mercato del web development.
4) Scarsa affidabilità: eventuali errori all'interno di un componente potrebbero influenzare la disponibilità e la fruibilità dell'intera applicazione. Ad esempio, se pensiamo ad un'applicazione front-end con architettura SPA, se in pagina sono presenti due componenti di cui uno con un errore, inevitabilmente la pagina non potrà essere fruita;
5) Scalabilità ridotta: i componenti all'interno di un'architettura monolitica sono spesso altamente dipendenti, per cui è complesso scalare l'applicazione in modo selettivo.

Per ovviare alle criticità dovute all'adozione dell'architettura monolitica in progetti di grandi dimensioni, nel corso degli anni 2000 è stato introdotto il concetto di architettura a microservizi, che ha determinato una rivoluzione nell'industria del software. Tale architettura è stata ideata per offrire una modalità di sviluppo e distribuzione del software più agile e scalabile, in linea con le nuove metodologie di sviluppo che sono state sviluppate nel corso degli anni (come Agile e DevOps), in modo tale da superare i limiti dell'architettura monolitica.

Tale tipologia di architettura è stata ideata per offrire una modalità di sviluppo e distribuzione del software più agile e scalabile, per superare i limiti dell'architettura monolitica e per adattarsi all'esigenza di nuove metodologie di sviluppo, come Agile e DevOps.

3.2.2 Architettura a microservizi

L'architettura a microservizi, indicata semplicemente con il termine "microservizi", è un design architetturale che consiste nella suddivisione dell'applicazione in componenti modulari, ciascuno dei quali svolge una specifica funzione e comunica con gli altri mediante API. In questo modo, ciascun microservizio può essere sviluppato, testato e distribuito in maniera indipendente dagli altri, semplificando quindi la gestione delle dipendenze tra i componenti e consentendo dunque una maggiore agilità nello sviluppo e nel rilascio effettivo del software.

Il pattern dell'architettura a microservizi nasce inizialmente per essere applicato all'interno di quello che, nell'architettura three-tier, è indicato come "Business Logic Tier", dove la modularità e la scalabilità sono fondamentali, soprattutto all'interno di progetti di grandi dimensioni.

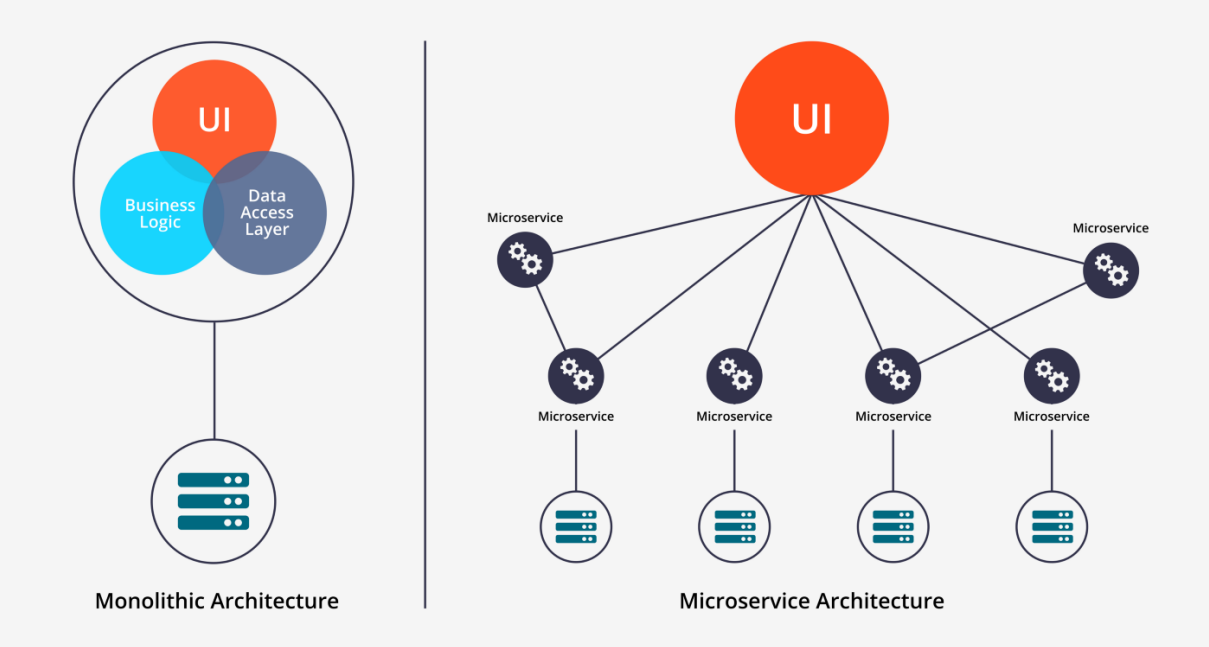


Figura 3.5 Architettura monolitica vs Architettura a microservizi

I microservizi si basano sul concetto di *"separation of concerns",* uno dei principi chiave dello sviluppo software. Questo principio si basa sulla separazione delle funzionalità dell'applicazione web in moduli indipendenti, separati e specializzati, che possono essere gestiti separatamente e scalati selettivamente. Tale approccio consente quindi alle grandi aziende di creare applicazioni altamente scalabili e distribuite su larga scala.

Ogni componente, secondo questa filosofia, deve possedere un singolo scopo e non deve dipendere dagli altri servizi, con cui può eventualmente comunicare tramite delle API specifiche.

Basandosi sulla decomposizione dell'applicazione tramite il disaccoppiamento dei suoi componenti, ciascun componente, che indicheremo con il termine "microservizio", potrà essere quindi sviluppato e modificato indipendentemente dagli altri microservizi, come se fosse un'applicazione autonoma.

Per questa ragione, ciascun microservizio può essere gestito da team di sviluppo diversi e specializzati. Inoltre, può essere sviluppato con tecnologie e linguaggi di programmazione diversi, dato che i microservizi non vengono generati a partire da un'unica codebase e le varie funzioni che concorrono alla creazione della pagina web non sono dipendenti tra loro. Ciascun microservizio può quindi essere modificato e ridistribuito in maniera autonoma, senza dover necessariamente effettuare un deploy dell'intera applicazione web.

Un esempio molto utile per comprendere le differenze tra l'architettura monolitica e i microservizi è quello dell'applicazione web di un e-commerce.

Nella sua configurazione più semplice, un e-commerce deve essere costituito da un'interfaccia front-end, un carrello, un catalogo e un configuratore di prodotti e una tecnologia per la gestione dei pagamenti, oltre ad altre integrazioni con i sistemi aziendali.

In un'architettura a microservizi, ciascun microservizio corrisponde ad una delle funzioni sopracitate: ciascuno di questi componenti viene aggiornato e modificato in modo molto variabile, dato che, ad esempio, l'interfaccia utente e il catalogo prodotti potrebbero avere tempi di deploy molto più frequenti rispetto al carrello o al sistema di generazione degli ordini.

Se in questo contesto utilizzassimo un'architettura monolitica, ad ogni aggiornamento di un componente sarebbe necessario rilasciare nuovamente l'intera applicazione, mentre con un'architettura monolitica si avrebbe la possibilità di agire in modalità puntuale, lavorando sui singoli componenti interessati e senza coinvolgere ciò che non dovrebbe essere oggetto di modifiche.

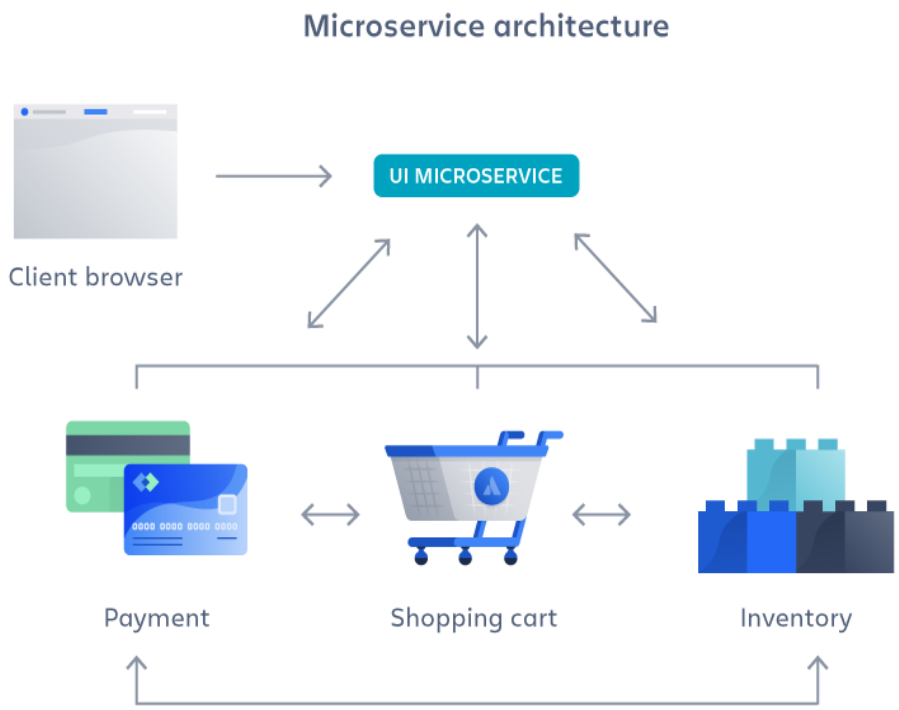


Figura 3.6 Schema e-commerce con architettura a microservizi

In contesti di grandi applicazioni, infatti, le funzioni previste dall'applicazione spesso non sono gestite e rilasciate da un unico soggetto, dato che ciascun fornitore agisce in maniera indipendente dagli altri: chi sviluppa l'UX design dell'applicazione non ha infatti bisogno di conoscere il flusso di sviluppo, ad esempio, del servizio per la gestione degli ordini, e chi si occupa dello sviluppo del front-end non necessita di conoscere i dettagli relativi agli asset del catalogo, ma solo una struttura base dei dati su cui andare ad effettuare gli sviluppi.

Di seguito un'analisi dei vantaggi che derivano dall'utilizzo di un'architettura a microservizi:

1) Scalabilità

La scalabilità è uno dei vantaggi principali di un'applicazione web con architettura a microservizi. In un'applicazione monolitica, infatti, l'aumento del traffico si ripercuote inevitabilmente sulla performance dell'applicazione: se i visitatori del sito web di un e-commerce acquistano più prodotti contemporaneamente, l'aumento del carico del sito potrebbe causare il blocco dell'applicazione, dato che tutti i componenti del monolite utilizzano le stesse risorse, e quindi il front-end, il back-end e il database vengono sovraccaricati contemporaneamente.

Al contrario, all'interno di un'applicazione web a microservizi, ogni componente del sistema ha delle risorse dedicate. Pertanto, se il traffico verso il sito web aumenta, solamente i servizi front-end, utilizzeranno più risorse rispetto ai servizi back-end.

Per ottimizzare ancora di più l'allocazione delle risorse, inoltre, sono stati creati degli strumenti di orchestrazione che allocano dinamicamente le risorse ai servizi in base alle necessità del momento.

La scalabilità, inoltre, facilita anche la distribuzione di nuove funzionalità e le modifiche ai singoli microservizi senza necessariamente mettere offline l'intera applicazione.

2) Robustezza

Con un'architettura a microservizi, la presenza di un errore in un servizio non ha un impatto negativo sugli altri, dato che ciascun microservizio è isolato dagli altri. Tuttavia, nel caso di applicazioni di grandi dimensioni, è probabile che dei microservizi abbiano delle dipendenze, per cui è necessario implementare dei sistemi per proteggere l'applicazione da un arresto dovuto a un errore in una dipendenza.

3) Software agnostico rispetto alle tecnologie e ai linguaggi di programmazione

I diversi microservizi possono essere sviluppati con qualsiasi linguaggio di programmazione e con qualsiasi framework. Questo garantisce una migliore flessibilità all'interno dei team di sviluppo, che scelgono le tecnologie da utilizzare per lo sviluppo del microservizio a seconda delle necessità del team e del servizio stesso.

4) Sicurezza

L'architettura a microservizi consente di adottare un approccio selettivo in termini di sicurezza dei dati: ciascun microservizio, infatti, è responsabile di un compito specifico, per cui è molto più semplice realizzare dei sistemi di sicurezza dei dati. Per collegare i microservizi, infine, vengono utilizzate API sicure che elaborano i dati assicurandosi che essi siano accessibili solo a elementi autorizzati.

5) Velocità di sviluppo e manutenibilità

I microservizi consentono di sviluppare e mantenere un'applicazione in maniera molto più semplice e veloce: grazie all'indipendenza tra microservizi, infatti, questi ultimi possono essere costruiti e modificati con meno rischi di conflitti e/o interruzioni dell'applicazione.

Al giorno d'oggi, l'architettura a microservizi per la gestione dei servizi back-end è diventata una scelta molto popolare, soprattutto all'interno di grandi aziende come ad esempio Amazon, Netflix, Spotify e Uber, che si sono rese, nel corso degli anni, promotori di tale modello architetturale. Tuttavia, l'architettura a microservizi non deve essere considerata sempre la soluzione migliore da attuare, soprattutto quando il team di sviluppo è piccolo e l'applicazione non necessita di tutte le funzionalità descritte precedentemente.

Inoltre, nello sviluppare un'applicazione con un'architettura a microservizi, bisogna sempre considerare le seguenti criticità:

1) Latenza nella comunicazione tra microservizi: più servizi vengono messi in comunicazione tra di loro, più bisogna gestire la comunicazione tra questi, cercando di garantire bassi livelli di latenza;
2) Utilizzo di più tecnologie: la scelta della tecnologia utilizzata per la realizzazione del microservizio è consigliata solamente se si hanno team separati che si occupano esclusivamente di una specifica tecnologia; quando si dispone di un piccolo team di back-end, infatti, sarebbe opportuno scegliere un unico stack tecnologico per lo sviluppo dei singoli microservizi;
3) Difficoltà nell'utilizzo delle utilities: all'interno dell'architettura a microservizi non è possibile condividere funzioni o librerie tra i vari microservizi a differenza, invece, dell'architettura monolitica;

4) Curva di apprendimento e di produttività: la scelta dell'architettura a microservizi piuttosto che dell'architettura monolitica deve tenere conto della complessità dell'applicazione stessa, della dimensione e delle capacità del team di sviluppo.

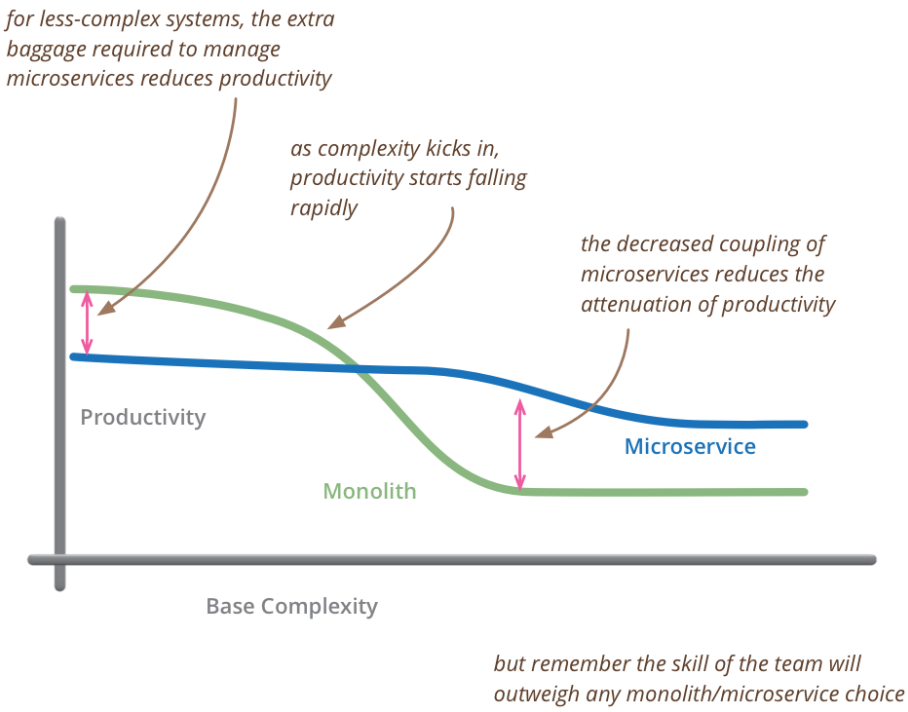


Figura 3.7 Grafico complessità-produttività legato alla curva di apprendimento dell'architettura a microservizi

L'architettura a microservizi rappresenta quindi una soluzione agile e scalabile per lo sviluppo e la distribuzione del software, focalizzandosi sulla scomposizione del sistema in moduli indipendenti e garantendo una migliore scalabilità e facilità di gestione.

I principi di questo design architetturale, inoltre, hanno avuto anche degli impatti sull'architettura interna del Presentation Layer, in quanto nel corso degli ultimi anni è stato sviluppata, per quanto riguarda l'ambito del front-end, un modello architetturale che si ispira ai principi dell'architettura a microservizi: l'architettura a micro front-end.

L'architettura a micro front-end rappresenta un'ulteriore evoluzione nelle modalità di progettazione e sviluppo di applicazioni web moderne basate su un'architettura modulare e altamente scalabile, dato che permette di scomporre l'interfaccia utente in moduli

indipendenti, chiamati micro front-end, che, come i microservizi, possono essere sviluppati, testati e distribuiti indipendentemente l'uno dall'altro, oltre a poter essere sviluppati con stack tecnologici differenti.

3.3 Architetture front-end

Nella fase di sviluppo dell'applicazione web, anche la scelta dell'architettura front-end da utilizzare è una componente cruciale per migliorare la scalabilità, la manutenibilità e la flessibilità dell'applicazione web dal punto di vista dell'interfaccia utente, per cui è estremamente importante organizzare e strutturare il lato client dell'applicazione in modo efficace ed efficiente.

In questa sezione della tesi, verranno inizialmente analizzate le tecniche di rendering per le interfacce web, per poi analizzare le architetture front-end che nel corso degli anni sono diventate degli standard per lo sviluppo e l'organizzazione delle applicazioni web, come le Multi Page Application, le Single Page Application e l'architettura a micro front-end.

Le MPA (Multi-Page-Application) sono applicazioni web che si basano su pagine web separate, ciascuna delle quali possiede una propria URL. Le SPA (Single-Page-Application), invece, sono delle applicazioni web che si basano su una singola pagina che viene aggiornata in maniera dinamica in base alle richieste e senza la necessità di riaggiornare la pagina. L'archittettura a micro front-end, che è emersa negli ultimi anni e che prende spunto dall'architettura a microservizi, infine, è una metodologia di sviluppo che permette di suddividere l'interfaccia utente in componenti indipendenti, ciascuno dei quali viene sviluppato, testato e rilasciato in maniera indipendente.

Il focus della seguente sezione sarà l'analisi delle architetture che, nel corso degli anni, sono diventate degli standard per lo sviluppo e l'organizzazione del front-end di un'applicazione, andando a delineare dei pattern frequenti per diverse tipologie di applicazione.

La scelta dell'architettura front-end da utilizzare per il proprio progetto dipende chiaramente dalle esigenze specifiche del progetto stesso, dal contesto in cui esso verrà utilizzato e dalle disponibilità e risorse del team di sviluppo, oltre ad avere un impatto significativo sulla qualità dell'applicazione e sulla sua manutenibilità nel tempo.

3.3.1 Server Side Rendering (SSR)

Il Server Side Rendering è una tecnica di rendering per contenuti web che prevede che la pagina web venga generata direttamente dal server e inviata al browser già pronta per la visualizzazione.

Il Server Side Rendering prevede che il server si occupi quindi della generazione dell'HTML, che viene fornito al browser in maniera tale che esso possa essere visualizzata senza la necessità di dover attendere l'esecuzione di codice JavaScript lato client. Ogni qual volta che l'utente effettua un'interazione con la pagina, il browser manda una richiesta al server che genera e fornisce la nuova pagina HTML per ogni interazione che l'utente effettua.

Il Server Side Rendering permette di realizzare applicazioni web molto performanti in termini di velocità e SEO (Search Engine Optimization), in quanto ad ogni richiesta dell'utente corrisponde una singola pagina che è sia più rapida da caricare, data la scarsa quantità di codice Javascript lato client, sia personalizzabile secondo le metodologie per l'ottimizzazione SEO per garantire una migliore indicizzazione della pagina attraverso i motori di ricerca.

La generazione delle pagine sul server, quindi, permette di evitare l'invio di una grande quantità di codice Javascript al client, permettendo quindi di diminuire il tempo necessario affinché la pagina sia completamente interattiva.

3.3.2 Client Side Rendering (CSR)

Il Client Side Rendering è una tecnica di rendering delle pagine web che prevede la generazione e il popolamento della pagina all'interno del codice Javascript: tutte le logiche di routing, di gestione dei dati e di generazione delle interfacce utente vengono quindi gestite sul client piuttosto che sul server.

L'aspetto negativo principale del rendering lato client è che la quantità di codice Javascript necessario aumenta con l'aumentare delle dimensioni e delle funzionalità dell'applicazione. Ad esempio, l'utilizzo di librerie aggiuntive per l'integrazione di funzionalità all'interno dell'applicazione può determinare un rallentamento non trascurabile del caricamento della

pagina web, in quanto molto spesso le librerie devono essere elaborate prima di visualizzare il contenuto in pagina.

Il Client Side Rendering è un approccio che è divenuto estremamente popolare con la nascita dei moderni framework front-end come ReactJS e Angular.

Quando l'utente accede a una pagina web renderizzata sul client, il server risponde con una pagina HTML vuota che viene popolata dal client stesso, che deve innanzitutto effettuare il download del codice Javascript per la gestione di tutte le funzionalità e occuparsi della creazione dell'interfaccia utente stessa; solo dopo aver compilato il codice, quindi, la pagina web diventa visibile e interattiva. Durante la compilazione del codice, è inoltre buona prassi prevedere dei sistemi per informare l'utente circa lo stato dell'applicazione, ad esempio mostrando un loader nell'attesa della compilazione del codice. Successivamente al primo caricamento della pagina, inoltre, la pagina web non viene mai re-renderizzata o rigenerata da zero in seguito all'interazione dell'utente: in tal caso, infatti, il client invia una richiesta al server che risponde con dei dati che vengono utilizzati per la modifica della pagina web generata al primo caricamento.

3.3.3 Approccio Ibrido

Nel corso degli ultimi anni sono stati invece sviluppati dei framework che permettono di avere un rendering misto dell'applicazione web, come ad esempio Next.js, un framework basato su React.js che permette di effettuare delle modalità di rendering misto.

Next.js, ad esempio, permette di definire delle pagine per cui effettuare un pre-render, generando quindi l'HTML delle suddette pagine in anticipo, a ciascuna delle quali viene associato un bundle di codice Javascript da eseguire per la pagina stessa, che viene processato solo quando la pagina è caricata dal browser. Inoltre, Next.js permette di sviluppare componenti specificamente pensati per essere renderizzati lato Server o lato Client, a seconda delle necessita specifiche dell'applicazione: se un componente necessita dell'accesso alle risorse back-end, ad esempio, sarebbe meglio utilizzare un Server Component, mentre se il funzionamento del componente è strettamente legato all'interazione dell'utente, questo dovrebbe essere un Client Component.

3.3.4 MPA: Multi Page Application

L'approccio tradizionale alle costruzioni delle pagine web è conosciuto come MPA (Multi Page Application). In questa architettura, ogni pagina web è un documento HTML separato che viene caricato dal server in risposta alle singole richieste del client: quando un utente interagisce con l'applicazione, ad esempio facendo un click su un link, il browser invia una richiesta al server, che a sua volta risponde con un documento HTML che viene mostrato e caricato all'interno del browser con le relative risorse.

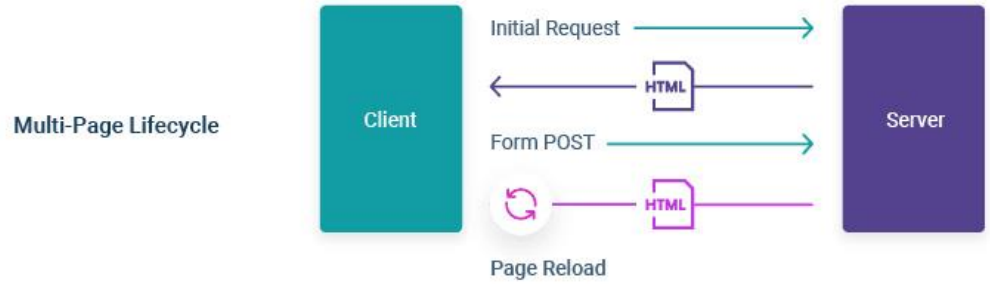
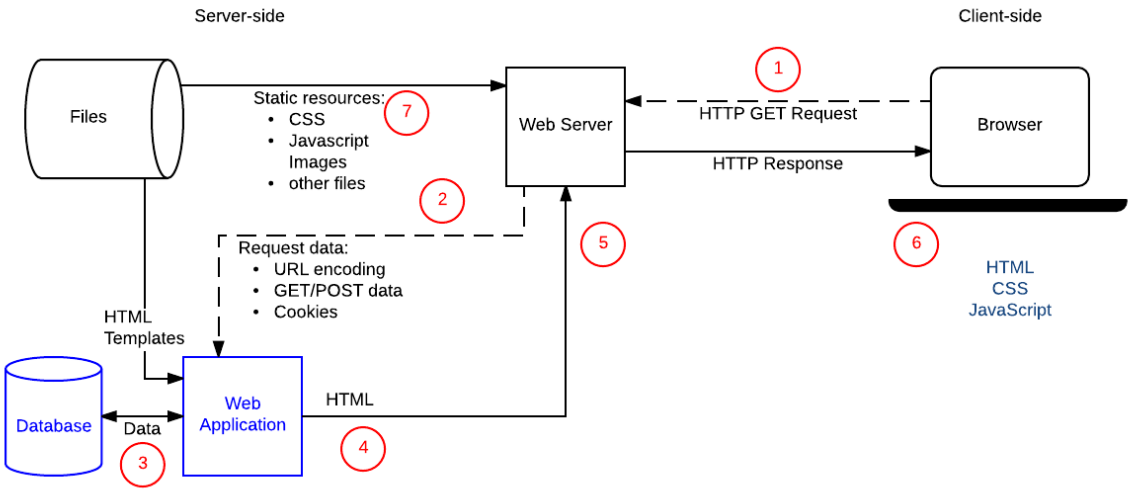


Figura 3.8 Schema del ciclo di comunicazioni client-server di una MPA



Nelle Multi Page Application, la maggior parte delle pagine HTML viene renderizzata direttamente dal server secondo l'approccio del SSR. Le MPA che utilizzano questa tecnica di rendering eseguono il rendering di tutto l'HTML direttamente sul server, e spesso non

richiedo l'esecuzione di codice Javascript lato client. In questo modo, le MPA sono molto più veloci nel primo caricamento della pagina web.

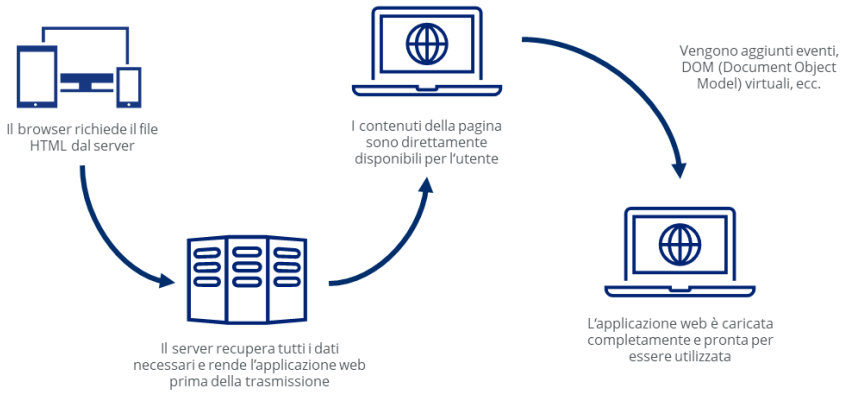
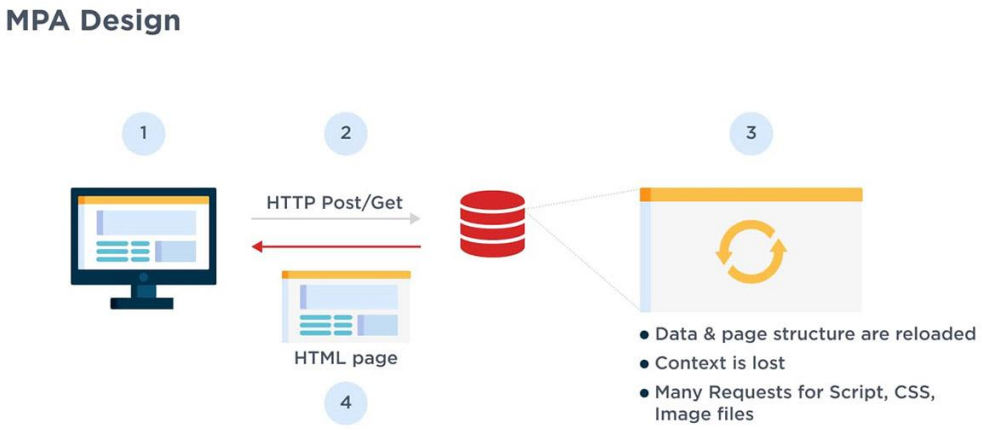


Figura 3.9 Ciclo di esecuzione di una MPA



L'utilizzo di una MPA rispetto ad altre architetture, come le SPA, comporta diversi vantaggi legati all'architettura stessa dell'applicazione, come ad esempio:

1) Ottimizzazione SEO

Le Multi Page Application consente una migliore gestione del SEO (Search Engine Optimization) in quanto ogni pagina può essere ottimizzata separatamente per i motori di ricerca, andando a definire i metadata necessari all'interno di ciascun file HTML. Le pagine

possono essere quindi indicizzate in maniera migliore, così da poter raggiungere posizioni migliori all'interno dei risultati di ricerca.

2) Scalabilità

Le Multi Page Application consentono di creare nuovi contenuti e di inserirli in nuove pagine create appositamente con lo scopo di visualizzare dei contenuti specifici. Per tale ragione, le MPA risultano essere particolarmente adatte per applicazioni web di grandi dimensioni. Inoltre, dato che ogni pagina può essere sviluppata e ottimizzata separatamente, è possibile avere una maggiore scalabilità e flessibilità. Infine, non vi è limite al numero di pagine che si possono inserire all'interno dell'applicazione.

3) Data Analytics approfonditi

Per le MPA esistono molti strumenti di analisi dati che permetto di avere informazioni approfondite sul comportamento degli utenti sulle singole pagine.

Tuttavia, le Multi Page Application presentano diversi svantaggi dovuti alla struttura stessa della codebase generata e alle modalità di creazione delle singole pagine web all'interno del browser:

1) Prestazioni

Nelle Multi Page Application, a ogni interazione il server deve ricaricare la maggior parte delle risorse, come l'HTML, il CSS e gli script Javascript. Quando si accede a un'altra pagina attraverso un link, infatti, il browser ricarica completamente i dati della pagina e scarica nuovamente tutte le risorse a seguito della richiesta HTTP da parte del client, includendo anche quei componenti che si ripetono all'interno di tutte le pagine, come ad esempio l'Header e il Footer. Tutto ciò, quindi, influisce negativamente sulla velocità e sulle prestazioni, oltre a fornire un'esperienza utente meno fluida rispetto ad altre architetture front-end.

2) Tempi di sviluppo e criticità nell'aggiornamento e manutenzione

Lo sviluppo delle MPA necessita di più tempo rispetto a quello con altre architetture front-end (come SPA). Tale complessità è dovuta al fatto che bisogna creare e gestire più pagine

contemporaneamente. Poiché ogni pagina può essere gestita e creata separatamente, aggiornamenti ad una singola pagina potrebbero richiedere la modifica di più parti dell'applicazione.

3.3.5 SPA: Single Page Application

Le Single Page Application (SPA) sono applicazioni web che, al contrario delle Multi Page Application, consistono di una singola pagina HTML che viene aggiornata dinamicamente in base alle richieste del client senza la necessità di riaggiornare e ricaricare completamente la pagina.

Le Single Page Application hanno avuto una rapida evoluzione negli ultimi anni, ma le prime idee risalgono alla fine degli anni 90: Nel 1998, per esempio, Microsoft ha rilasciato la prima versione di Internet Explorer con il supporto di XTMLHttpRequest, un oggetto Javascript che permetteva di effettuare richieste asincrone ad un server web senza dover ricaricare l'intera pagina. Nel 2005, poi, Google ha rilasciato Google Maps, una delle prime SPA che ha attirato l'attenzione dell'industria informatica, dimostrando che le SPA possono essere utilizzate per la realizzazione di applicazioni web altamente reattive senza la necessità di dover ricaricare la pagina e influire negativamente sulla fluidità dell'esperienza utente.

Una SPA, quindi, è un'applicazione web in cui, dopo aver caricato la pagina HTML base, il server invia solamente i dati richiesti dal client, che a sua volta si occuperà di renderizzare le nuove informazioni senza ricaricare completamente la pagina.

Le SPA si basano su Ajax, "Asynchronous Javascript and XML", una tecnologia di programmazione web che consente di aggiornare la pagina in modo asincrono senza doverla necessariamente ricaricare, permettendo quindi di inviare e ricevere dati dal server in maniera asincrona, senza quindi interrompere l'esperienza utente.

Dopo aver caricato la pagina, ogni volta che l'utente interagisce con l'applicazione, ad esempio compilando un form o interagendo con un link, l'applicazione invia una richiesta AJAX al server per ottenere i dati necessari all'aggiornamento della pagina. Una volta ottenuti i dati sotto forma di file JSON, il client utilizza Javascript per aggiornare

dinamicamente il contenuto della pagina, senza dover effettuare nuove richieste HTTP per caricare una nuova pagina (come avviene nelle Multi Page Application).

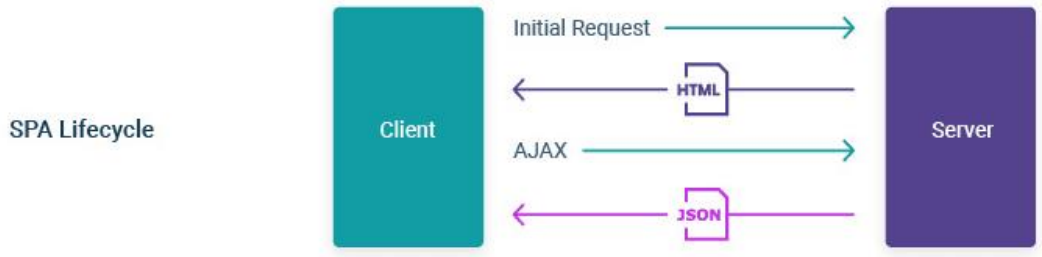


Figura 3.10 Ciclo di comunicazioni client-server in una SPA

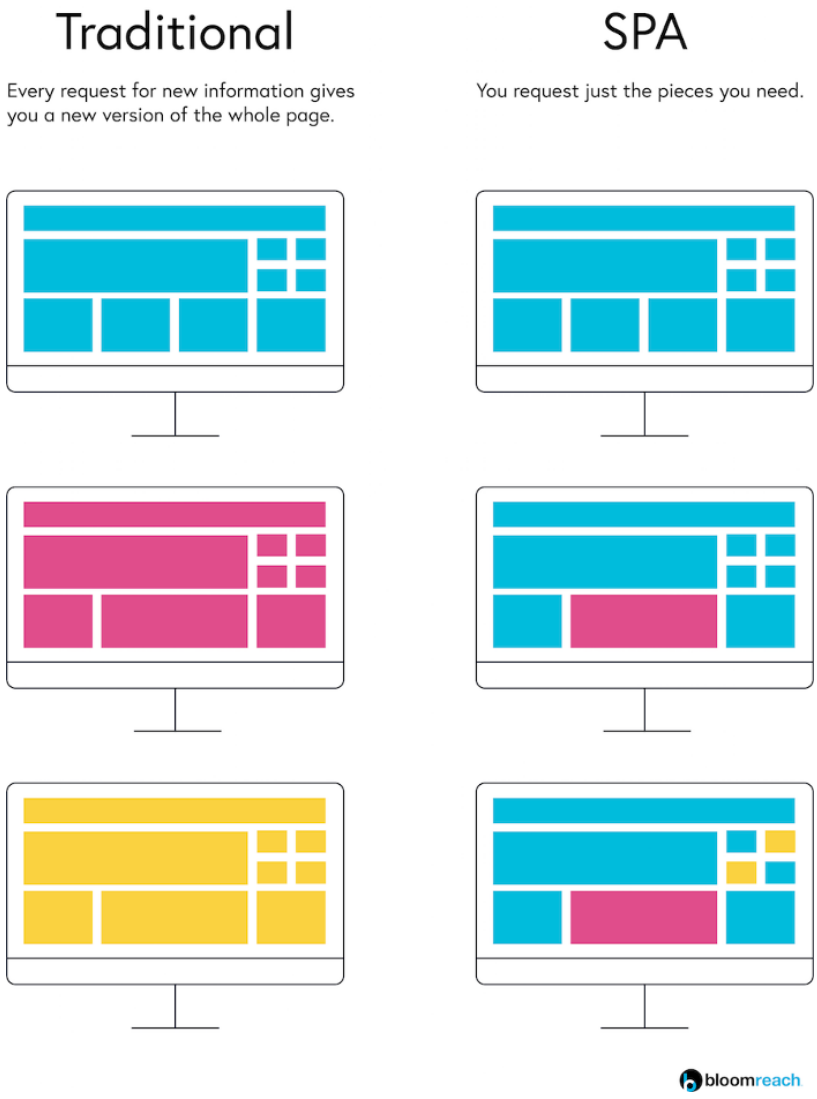


Figura 3,11 Confronto layout MPA vs SPA

Le Single Page Application, inoltre, oggi si basano fortemente sull'utilizzo di framework o librerie per accelerare e semplificare tale processo di sviluppo e creazione della pagina, fornendo una serie di strumenti e di componenti predefiniti che vengono utilizzati per la creazione dell'applicazione. In particolare, i framework SPA offrono funzionalità per il routing, per la gestione degli stati e per la gestione delle chiamate API.

Il primo framework per le Single Page Application è stato Angular.js, sviluppato e rilasciato da Google nel 2010, ed ha avuto un'importanza fondamentale in quanto fornisce una struttura organizzativa del progetto e diverse funzionalità predefinite. Nel corso degli anni a seguire, sono stati sviluppati poi altri framework e altre librerie per lo sviluppo di SPA, come React, Vue.js e Ember.js, che forniscono anch'essi funzionalità predefinite e pattern semplificati per l'organizzazione del codice rispetto alle SPA costruite solamente con Javascript.

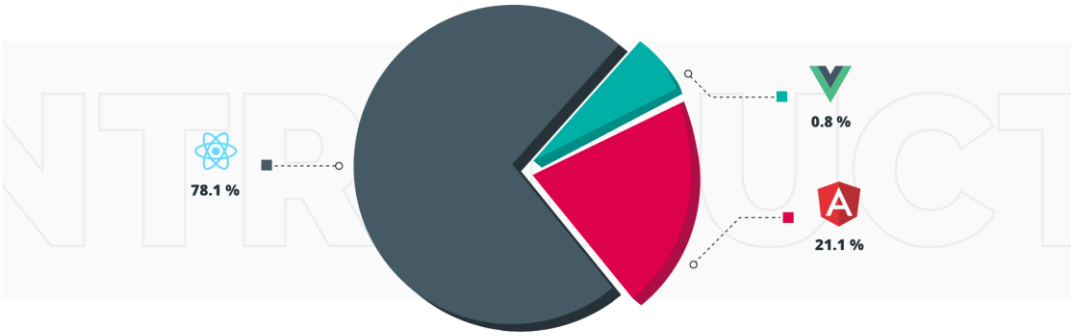


Figura 3.12 Grafico a torta contenente percentuali di adozione dei framework front-end più diffusi.

L'utilizzo delle Single Page Application permette di avere numerosi vantaggi sia in fase di sviluppo dell'applicazione sia per quanto riguarda l'esperienza utente in fase di navigazione.

1) Performance: poiché le SPA non aggiornano l'intera pagina, ma solo il contenuto necessario in seguito all'interazione con i contenuti da parte dell'utente, il risultato sarà una pagina web più veloce rispetto a una MPA;
2) User Experience: non necessitando di ricaricare completamente la pagina, le SPA offrono agli utenti un'esperienza d'uso più lineare e comprensibile. Inoltre, l'utilizzo di framework e librerie permette di costruire interfacce più flessibili e reattive;

3) Data caching: dopo la prima richiesta al server, le SPA memorizzano nella cache tutti i dati locali necessari, permettendo agli utenti di poter lavorare anche in modalità offline;
4) Velocità di sviluppo: nel caso di librerie component based come ad esempio React, è possibile sviluppare e riutilizzare parti di codice in pagine differenti, oltre a poter testare un minor numero di elementi dell'applicazione;
5) Debugging: debuggare e testare una SPA realizzata con framework come Angular o librerie come React risulta essere molto semplice, in quanto esistono dei tools da poter installare direttamente all'interno del browser per monitorare lo stato dell'applicazione.

Tra gli svantaggi e le criticità delle SPA, invece, abbiamo:

1) Scarso supporto al SEO: dato che all'interno delle SPA non esistono effettivamente pagine separate ma un solo documento HTML che viene aggiornato dinamicamente in base alle richieste del client, non è possibile ottimizzare le varie pagine per il SEO. Per poter fare ciò bisogna quindi utilizzare meccanismi di Server Side Rendering.
2) Tempi di download se la piattaforma è complessa e poco ottimizzata;
3) Supporto Javascript: poiché le SPA si basano sull'utilizzo di Javascript lato client ogni qualvolta viene fatta una richiesta al server, un requisito fondamentale per l'utilizzo delle SPA è che Javascript sia abilitato all'interno del browser;
4) Mancanza di cronologia: il funzionamento di base di un browser consente, tramite l'utilizzo del back button, di tornare alle pagine precedenti. Dato che il concetto di "pagina" in senso stretto non esiste nelle SPA, tale funzionalità del browser non avrebbe senso all'interno di un client realizzato con tale architettura, in quanto servirebbe tornare a uno stato precedente dell'applicazione (e non ad una pagina specifica). Per ovviare a tale problematica, tuttavia, esistono delle API specifiche e delle funzionalità interne ai vari framework e alle librerie.

4. Architetture a micro front-end

4.1 Contesto

Con la nascita e il successo dell'architettura a microservizi, molte organizzazioni hanno iniziato ad utilizzare questo stile architetturale in maniera tale da evitare le limitazioni, in termine di gestione e di scalabilità, dei servizi back-end monolitici e di grandi dimensioni. D'altro canto, per quanto riguarda la parte front-end dei progetti web, la transizione a modelli più agevoli in termini di manutenzione e scalabilità non è andata di pari passo con la transizione dal back-end monolitico ai microservizi, in quanto molti team e molte aziende, seppur appoggiandosi per il livello logico dell'applicazione su un'architettura a microservizi, hanno continuato a implementare il front-end seguendo le logiche delle architetture monolitiche.

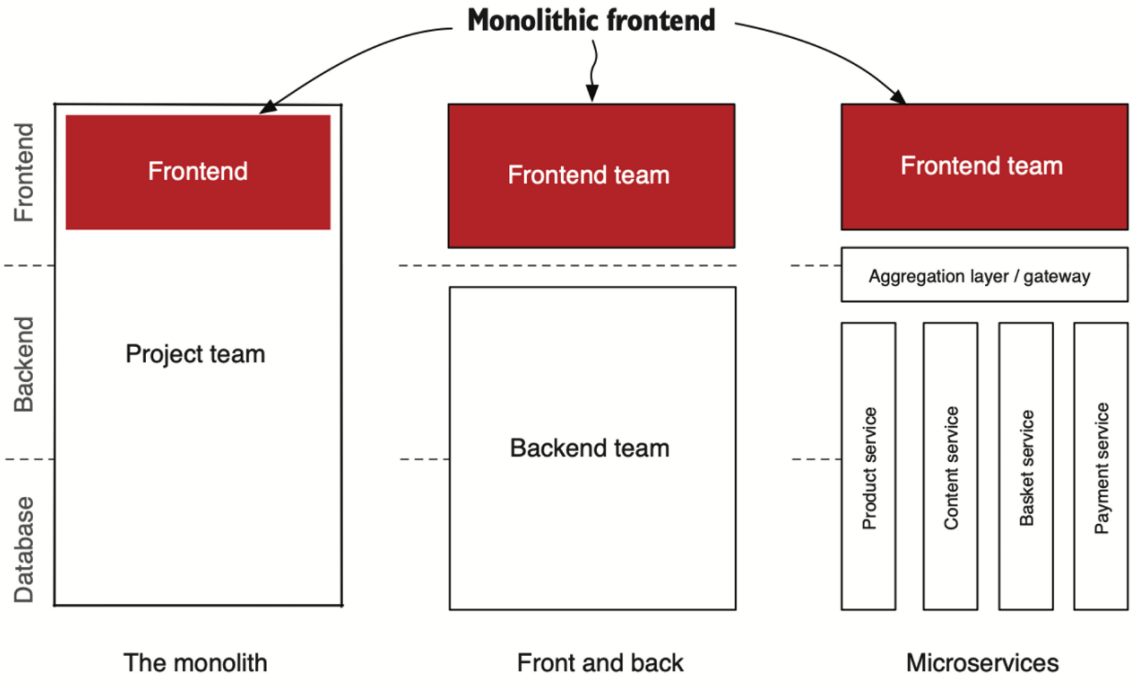


Figura 4.1 Evoluzione delle architetture web, dall'architettura monolitica all'architettura a microservizi.

L'immagine in figura mostra in maniera molto semplice come, nel corso del tempo, si sono evolute le modalità di sviluppo web, partendo da un modello monolitico in cui il back-end era strettamente legato al front-end, passando a un modello in cui vi è stata una separazione tra il livello di presentazione e il livello logico, per arrivare, infine, al modello iniziale dell'architettura a microservizi.

In questo modello, infatti, mentre i servizi back-end più grandi sono suddivisi in termini di responsabilità in microservizi, i client che consumano i dati forniti da questi sono ancora dei monoliti.

Per molte situazioni, un'architettura front-end monolitica può essere la scelta migliore per lo sviluppo dell'applicazione, soprattutto per piccoli progetti o aziende con piccoli team di sviluppo. Tuttavia, nel momento in cui il progetto cresce in termini di dimensioni e complessità, si complica di molto la gestione del codice e del team, portando il "monolite" ad essere un ostacolo per l'azienda stessa per diversi motivi:

1) Problematiche in termini di sviluppo, distribuzione e scalabilità

Dato che il front-end monolitico è distribuito e rilasciato come se fosse un unico blocco, vi sono degli inevitabili rallentamenti in termini di sviluppo e distribuzione del progetto. Dato che tutte le funzionalità dell'applicazione sono implementate in un unico blocco di codice, qualsiasi aggiornamento o correzione richiede il rilascio dell'intera applicazione, che può essere lento e oneroso nel caso di applicazioni complesse.

Inoltre, dato che nel front-end monolitico tutti i componenti sono strettamente integrati tra di loro, qualsiasi interruzione o problema in uno dei componenti presenti all'interno di una pagina potrebbe causare l'interruzione dell'intera applicazione web, determinando quindi un impatto significativo sulla disponibilità dell'applicazione in rete e sull'esperienza degli utenti.

Infine, poiché tutte le funzionalità dell'applicazione sono integrate all'interno dello stesso progetto, risulta molto complicato aggiornare le dipendenze o le librerie utilizzate senza compromettere l'integrità dell'applicazione stessa, portando a problemi di compatibilità e di sicurezza.

2) Mancanza di flessibilità

Nelle applicazioni monolitiche, il fatto che l'applicazione sia sviluppata basandosi su una codebase unificata i cui componenti sono strettamente integrati tra di loro in termini di dipendenze e configurazione implica diverse limitazioni per quanto riguarda le scelte tecniche e tecnologiche che possono essere effettuate in fase di sviluppo del progetto.

Infatti, un'applicazione monolitica è strettamente vincolata dalle tecnologie che sono state utilizzate precedentemente all'interno dell'applicazione: se, per esempio, un'applicazione è basata sul framework React, tutti i componenti dovranno funzionare secondo le logiche dettate dal framework, e non è possibile decidere di utilizzare un altro framework per la realizzazione di nuove funzionalità e di nuovi componenti, in quanto non è possibile introdurre nuove tecnologie nella vecchia architettura.

3) Gestione del team e developer experience

Nel caso di progetti di piccole dimensioni gestiti da un piccolo team di sviluppo specializzato in una specifica tecnologica e che è in grado di conoscere le singole funzionalità dell'intera applicazione, sviluppare un'applicazione basandosi su un front-end monolitico potrebbe essere la scelta ideale in termini di tempi di sviluppo.

Se parliamo di progetti di grandi dimensioni gestiti da un numero elevato di sviluppatori, la gestione del progetto stesso potrebbe essere estremamente complicata, in quanto è praticamente impossibile che ciascuno sviluppatore sia in grado di conoscere alla perfezione ogni singola parte del progetto, e anche nel caso di team dedicato a specifiche funzionalità dell'applicazione, il rischio di riscontrare divergenze o conflitti tra i team nella fase di sviluppo del codice è elevato.

Inoltre, per quanto riguarda la gestione delle risorse, in progetti di grandi dimensioni gestiti con un front-end monolitico, la produttività del team stesso rischia di essere influenzata negativamente a causa delle risorse necessarie per formare nuove figure che siano in grado di conoscere le singole funzionalità dell'applicazione.

Questo pattern organizzativo finisce quindi per peggiorare l'esperienza di sviluppo degli stessi sviluppatori, che devono potenzialmente conoscere ogni aspetto del sistema oltre ad essere necessariamente limitati in termini di scelte tecnologiche da poter effettuare a causa della scarsa flessibilità dell'architettura stessa.

L'architettura a micro front-end nasce dunque per ovviare a tali criticità, dando origine ad un nuovo approccio per la creazione del front-end di un'applicazione, prendendo ispirazione dalla già consolidata architettura a microservizi così da poter scomporre il "monolite" del front-end in tanti moduli separati e indipendenti tra di loro.

4.2 Dall'architettura monolitica all'architettura a micro front-end

La tendenza ricorrente per lo sviluppo delle applicazioni web è quella di sviluppare una SPA monolitica per il front-end che si appoggia ad una struttura back-end basata sull'architettura a microservizi, in cui il client front-end comunica con questi ultimi o tramite degli endpoint pubblici accessibili dal client front-end stesso o tramite dei gateway API che assumono il ruolo di intermediari tra i client e i microservizi.

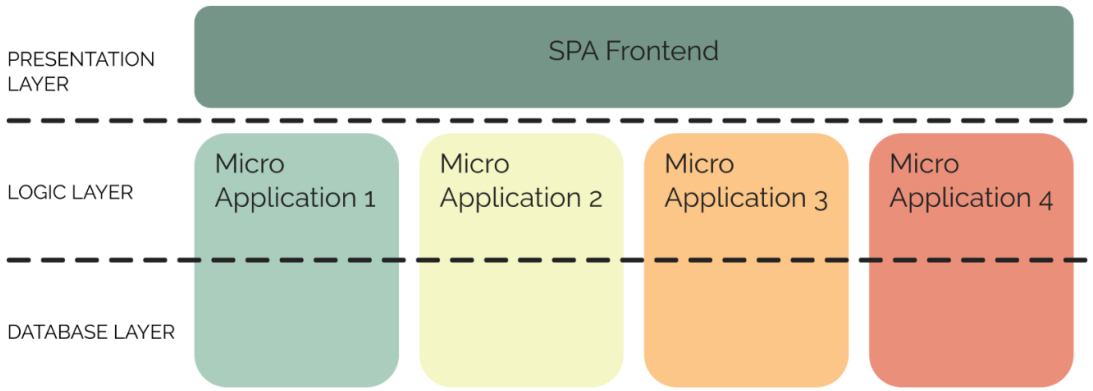


Figura 4,2 Schema dell'architettura di una SPA

Il fatto che il front-end sia presente, all'interno di questo diagramma, come un unico blocco, indica che la parte dell'applicativo che si occupa di gestire l'interfaccia utente viene distribuita a partire da un'unica codebase su cui effettivamente un solo team può lavorare in maniera organizzata e ragionevole, conoscendo il progetto nella sua interezza.

Inoltre, questa struttura logica, dato che il front-end e il back-end hanno responsabilità separate e sono implementati seguendo logiche e pattern di sviluppo diversi, ha dato origine alla nascita di una organizzazione dei team di sviluppo secondo una struttura orizzontale che prevede la separazione del team in team specializzati nella realizzazione dei microservizi e in team specializzati nella realizzazione del front-end dell'applicativo web.

L'idea dell'architettura a micro front-end, invece, è quella di estendere la metodologia di sviluppo dell'architettura a microservizi anche al front-end, suddividendo quindi il Presentation Layer in tanti moduli separati che provvedono alla realizzazione e alla gestione di una specifica funzionalità dell'applicazione web. Il front-end viene quindi suddiviso in tanti piccoli front-end indipendenti tra di loro, ciascuno dei quali è sviluppato e distribuito in maniera autonoma dagli altri: essendo delle vere e proprie microapplicazioni, infatti,

possono essere realizzati con soluzioni tecnologiche diverse tra di loro, proprio come nel caso dei microservizi.

Il concetto di architettura a micro front-end, dunque, non introduce unicamente un nuovo approccio alla realizzazione delle architetture web, ma una vera e propria rivoluzione in termini organizzativi e strutturali nella gestione dei team di sviluppo.

Poiché ogni singolo micro front-end comunica solamente con i microservizi necessari per la realizzazione della funzionalità ad esso dedicata, l'applicazione risulta suddivisa in tante piccole microapplicazioni full-stack: trattandosi di applicazioni separate, queste possono essere sviluppate e distribuite secondo delle pipeline anch'esse separate, oltre a poter essere sviluppate utilizzando tecnologie diverse, non essendoci vincoli e dipendenze tra le singole microapplicazioni.

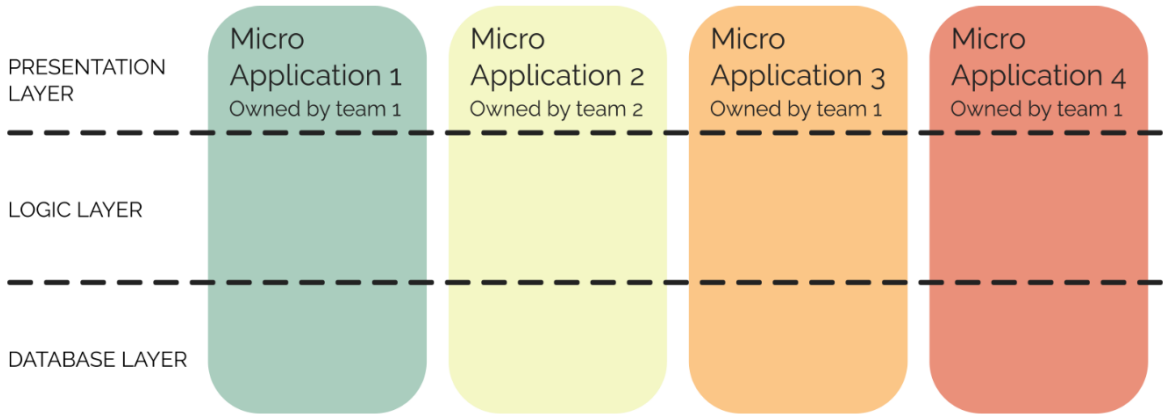


Figura 4.3 Schema dell'architettura a micro front-end

La rivoluzione in termini organizzativi e gestionali, invece, consiste nel fatto che l'architettura a microfront-end e la presenza di microapplicazioni full-stack separate tra di loro favorisce la nascita di un nuovo modo di organizzare i gruppi in team di sviluppo verticali, in cui ciascun team si occupa di una singola funzionalità dell'applicazione. All'interno di questi team, inoltre, le conoscenze non sono specifiche e relative a un singolo ambito, come nel caso della suddivisione in team front-end e back-end, ma verticali e relative a ciascuna fase dello sviluppo della funzionalità a cui la microapplicazione è dedicata.

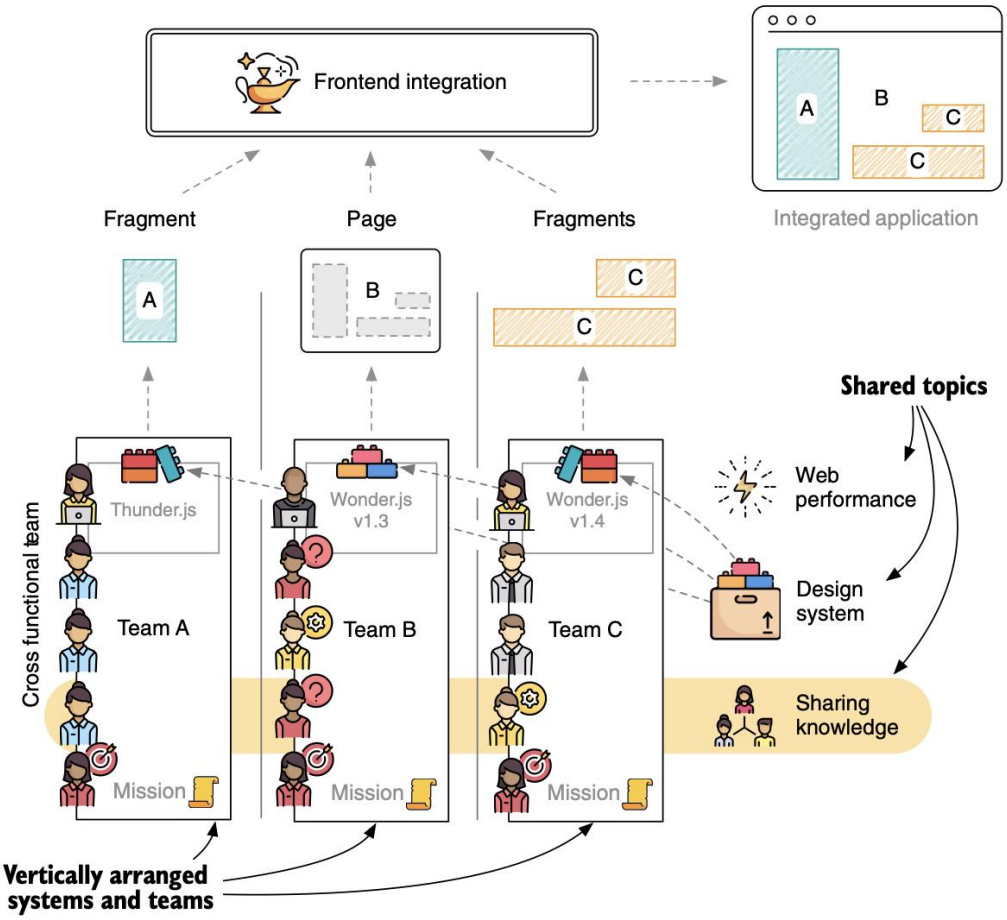


Figura 4.4 Suddivisione in team verticali all'interno di un progetto con architettura a micro front-end

Ciascuna funzionalità è quindi di proprietà di un singolo team, che avrà un'unica area di competenza e una specifica missione relativamente allo scopo dell'applicazione web stessa e alle necessità degli utenti finali dell'applicazione.

Ciascun team si occuperà quindi di realizzare la propria microapplicazione indipendentemente dalle scelte tecnologiche degli altri team, per sviluppare il proprio modulo che dovrà poi essere integrato in pagina con tutti gli altri moduli appartenenti ad altri team.

Lo sviluppo delle singole microapplicazioni da parte di team full-stack con competenze trasversali che combinano individui con formazioni professionali diverse (sviluppo back-end, front-end UX) permette inoltre di analizzare una funzionalità da diversi punti di vista, favorendo quindi la possibile nascita di soluzioni più creative e interessanti. Inoltre, il fatto che vi sia una squadra unicamente dedicata alla progettazione di un singolo componente permette invece di diminuire la necessità di coordinamento tra team diversi, in quanto

all'interno dello stesso team sono presenti tutti gli elementi necessari all'ideazione e alla realizzazione della microapplicazione.

Nel caso di un ipotetico e-commerce, ad esempio, potremmo definire tre team, ciascuno con uno scopo specifico relativo alla customer journey dell'utente:

1) Team "Inspire": ispirare il cliente che naviga nell'applicazione, presentando i prodotti che potrebbero essere interessanti;
2) Team "Decide": aiuta a prendere decisioni d'acquisto informate e complete, fornendo immagini dei prodotti, specifiche e recensioni;
3) Team "Checkout": guidare il cliente verso il percorso d'acquisto.

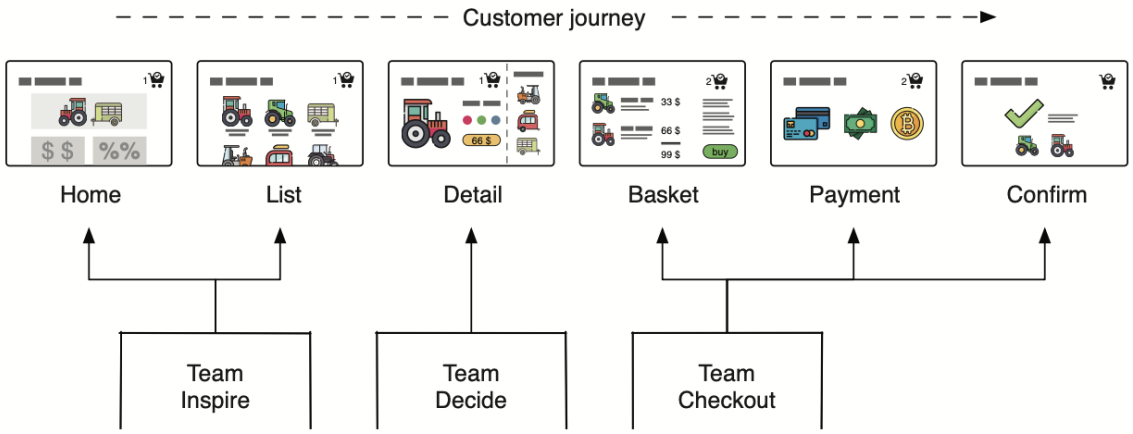


Figura 4.5 Suddivisione dei componenti in base al team di riferimento

Le singole pagine dell'e-commerce possono essere gestite da team diversi, sulla base dello step di riferimento della customer journey, ma all'interno delle pagine è possibile avere anche dei moduli appartenenti a team differenti, per cui uno step fondamentale nella realizzazione dell'architettura è quello dell'integrazione a front-end dei componenti appartenenti a team differenti, che, come abbiamo detto, possono essere sviluppati utilizzando tecnologie di sviluppo, framework o librerie diverse tra di loro.

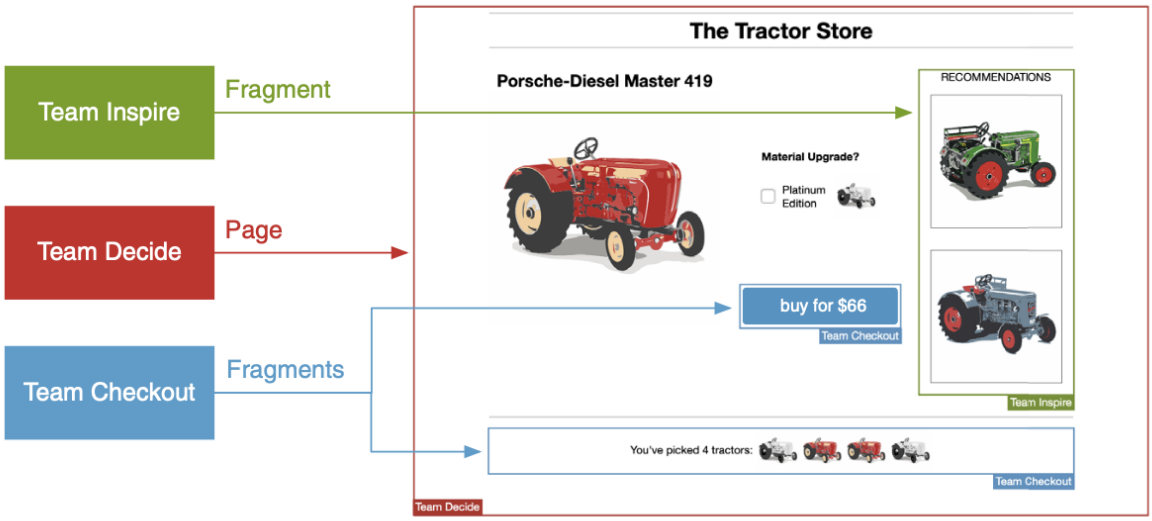


Figura 4.6 Ipotetico schema per una progettazione di un e-commerce con architettura a micro front-end.

Ad esempio, all'interno di un'ipotetica pagina di dettaglio prodotto, che potenzialmente apparterebbe al Team Decide, potrebbe essere presente una sezione legata ai prodotti correlati al prodotto selezionato (Team Inspire) o informazioni circa l'acquisto del prodotto stesso e/o informazioni relative ai prodotti che si trovano all'interno del carrello (Team Checkout).

Un team può quindi decidere di includere delle funzionalità appartenenti alle microapplicazioni di altri team, aggiungendoli all'interno delle pagine di propria gestione sotto forma di "Fragments", senza però avere effettivamente accesso ai suoi dettagli implementativi.

Ciascun team, inoltre, crea la propria repository e gestisce la propria pipeline di sviluppo in maniera indipendente da quella degli altri componenti, che viene eseguita ogni qual volta viene effettuata una modifica al codice, distribuendo una nuova versione della microapplicazione relativa al team. Le pipeline dei singoli team vengono quindi eseguite in maniera autonoma, e un cambiamento all'interno della microapplicazione del Team A con l'esecuzione della relativa pipeline, conseguentemente, non causerà mai alcuna interruzione delle singole fasi della pipeline della microapplicazione di un ipotetico Team B.

4.3 Vantaggi dell'architettura a micro front-end

L'adozione dell'architettura a micro front-end apporta dunque numerosi benefici al progetto, sia dal punto di vista organizzativo che dal punto di vista tecnico.

4.3.1 Developer Experience e Tempi di sviluppo

Una delle ragioni principali per cui le grandi aziende decidono di intraprendere la strada dell'architettura front-end è la diminuzione del Time To Go Live dovuta alla maggiore coordinazione degli elementi coinvolti nella realizzazione di una determinata funzionalità del progetto: con l'applicazione del modello dei team verticali e cross-funzionali, infatti, tutte le persone coinvolte nello sviluppo di una funzionalità lavorano nello stesso team, per cui vi è necessariamente una maggiore coordinazione e quindi una notevole riduzione dei tempi di comunicazione per effettuare delle decisioni sulla funzionalità stessa, nonostante il lavoro da dover effettuare sia lo stesso.

Dal punto di vista della Developer Experience, un elemento fondamentale risiede nella possibilità di poter effettuare le scelte tecnologiche che si ritengono più idonee con le specifiche necessarie per lo sviluppo della microapplicazione, a prescindere dalle scelte tecnologiche, come ad esempio framework o librerie: l'isolamento delle microapplicazioni, che sono del tutto indipendenti le une dalle altre, permette quindi di sviluppare i singoli componenti con le tecnologie che si ritengono più idonee, a prescindere dalle scelte degli altri team.

Infine, l'architettura a micro front-end semplifica di molto lo sviluppo, l'aggiornamento e la distribuzione delle singole microapplicazioni, dato che la codebase dei singoli componenti è completamente separata e indipendente da quella degli altri componenti. Mantenere il codice di ciascun tema separato, infatti, garantisce cicli di sviluppo, test e distribuzione molto più efficienti e veloci.

La possibilità di distribuire i micro front-end in maniera indipendente riduce la portata e i rischi dovuti al deployment dell'applicazione: ogni microapplicazione dovrebbe avere una propria pipeline di distribuzione, per cui dovrebbe essere possibile distribuire ogni micro front-end a prescindere dallo stato degli altri microfront-end che verranno inclusi all'interno del progetto.

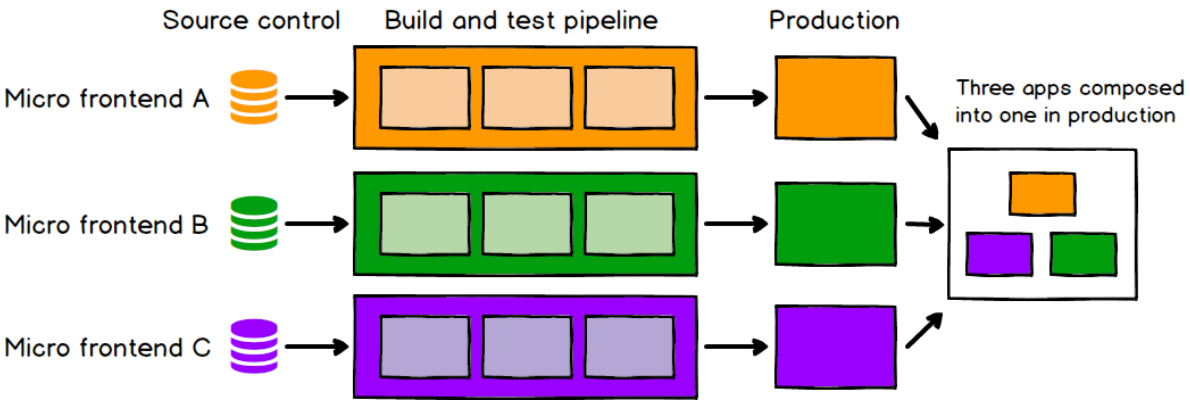


Figura 4.7 Suddivisione delle pipeline di sviluppo, test e distribuzione dei singoli micro front-end

4.3.2 Scalabilità

L'architettura a micro front-end permette di realizzare un'applicazione altamente scalabile, in quanto, dato che ciascun micro front-end è autonomo e indipendente, ciascuna parte del sistema può essere modificata a seconda delle necessità a prescindere dalle altre. I team di sviluppo di ciascun micro front-end, inoltre, possono introdurre cambiamenti alle singole microapplicazioni di riferimento senza intaccare le performance degli altri micro front-end.

4.4 Criticità dell'architettura a micro front-end

I benefici menzionati sono chiaramente accompagni da alcuni costi che riguardano diversi ambiti.

4.4.1 Complessità

L'architettura a micro front-end richiede la scomposizione delle applicazioni in moduli più piccoli, aumentando così la complessità generale dell'architettura e del sistema, motivo per cui è necessario un maggiore livello di coordinamento e collaborazione più ad alto livello per il mantenimento dell'integrità del sistema.

Inoltre, in termini di sviluppo e distribuzione, poiché i micro front-end vengono spesso sviluppati in ambienti indipendenti e diversi da quello che poi sarà effettivamente rilasciato in produzione, vi è il rischio che, nel momento in cui il micro front-end viene integrato all'interno dell'ambiente di produzione, siano presenti bug o errori che potrebbero interferire con il corretto funzionamento dell'applicazione. Per queste ragioni, quindi, è fondamentale

effettuare dei test mirati in ambienti simili a quello finale, così da poter individuare eventuali problemi di integrazione il prima possibile.

4.4.2 Ridondanza

Il fatto di avere più team che lavorano su microapplicazioni full-stack determina che, più ad alto livello, vi sia un elevata ridondanza in termini di codice, librerie e ambienti di sviluppo, dato che ciascun team deve gestire e mantenere il proprio server, il proprio database e la propria pipeline di distribuzione.

4.5 Composizione della pagina web

Un elemento fondamentale nello sviluppo di un'applicazione web basata sull'architettura a micro front-end è la definizione delle modalità con cui le singole microapplicazioni devono comporre la pagina principale dell'applicazione.

Esistono diverse modalità di implementazione dell'architettura a micro front-end e, in particolare, nel quinto capitolo verrà effettuata un'analisi della realizzazione di un'applicazione basata sui micro front-end tramite il framework Astro.

4.5.1 Collegamento ipertestuale tra applicazioni

La modalità più semplice di realizzazione di un'architettura a micro front-end consiste nella navigazione tra diverse applicazioni tramite l'utilizzo di collegamenti ipertestuali, che utilizzano le URL per reinderizzare l'utente verso una pagina o un contenuto specifico.

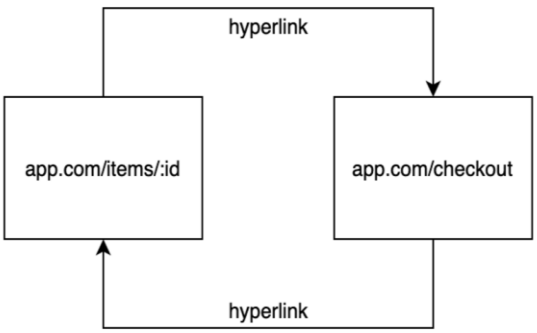


Figura 4.8 Schema collegamento ipertestuale tra applicazioni

Il problema principale di questo approccio è che è necessario, per effettuare il reindirizzamento, conoscere l'URL della specifica risorsa, di cui non è possibile garantire che sia sempre la stessa indicata.

```
<a href="path/url">About</a>
```

4.5.2 Iframes

Un iframe è un elemento HTML che permette di caricare l'HTML di un'altra pagina web nel documento in cui questo viene inserito. Gli iframe vengono spesso utilizzati per la realizzazione di ADV, per effettuare l'embedding dei video all'interno della pagina, ad esempio tramite le API di Youtube o Vimeo, o per l'inserimento di contenuti interattivi.

```
<iframe src="https://www.w3schools.com" title="W3Schools"></iframe>
```

Quando il browser incontra il tag *<iframe/>,* avviene la creazione di un nuovo documento HTML all'interno del quale viene caricato il contenuto richiesto, compreso di codice Javascript e CSS. Il documento contenitore, inoltre, può definire delle grandezze CSS per l'aspetto dell'iframe, ma non può accedere direttamente agli elementi, ai tag e al codice che viene renderizzato all'interno dell'iframe.

L'utilizzo degli iframes è spesso frequente per la realizzazione di applicazioni web basate sull'architettura a micro front-end, data la loro abilità di fornire indipendenza e un forte isolamento tra i vari micro front-end dato che, come detto precedentemente, se si utilizza un iframe all'interno di un container non è possibile accedere al codice contenuto all'interno dell'iframe: l'indipendenza delle varie microapplicazioni, infatti, è uno dei punti di forza dell'architettura a micro front-end.

Dunque, l'utilizzo degli iframe per la realizzazione di un'applicazione web basata sull'architettura a micro front-end presenta degli svantaggi significativi in termini di integrazione e flessibilità.

Infatti, l'utilizzo degli iframe determina delle forti limitazioni sulla possibilità di gestione dei layout, dato che un componente che incorpora un'altra microappicazione tramite iframe non ha accesso al codice e al CSS del componente inglobato.

Inoltre, l'utilizzo degli iframe può portare a un incremento della complessità e e ad un decremento in termini di performance, dato che ogni iframe deve essere caricato e gestito separatamente.

4.5.3 Server-side composition

Un'altra possibile metodologia di sviluppo di un'applicazione web a micro front-end consiste nella composizione dei differenti widget o componenti sul server: la composizione della pagina web viene effettuata sul server da un servizio che si pone come intermediario tra il server e il browser, a cui fornisce la pagina web completamente assemblata.

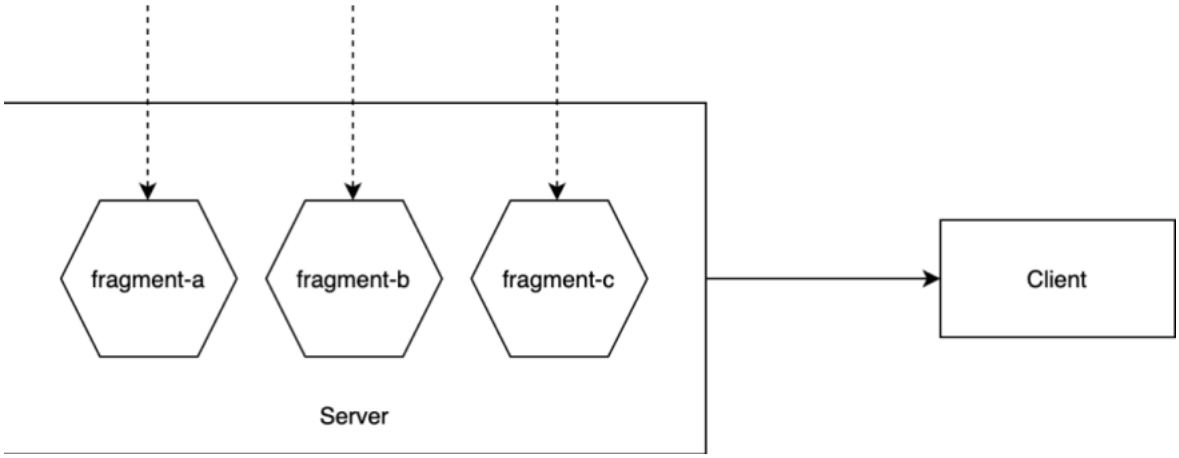


Figura 4.9 Server side composition

Secondo questa modalità di composizione della pagina, il server gestisce e compone la pagina inserendo i singoli micro front-end realizzati dai singoli team, per poi restituire al client l'HTML, il CSS e il codice Javascript necessario.

Questa tipologia di rendering può essere, ad esempio, realizzata tramite l'utilizzo della libreria Tailor.js, una libreria da integrare su un server Node.js che permette di utilizzare un servizio che si occupa specificamente di comporre la pagina finale andando a comporre i singoli frammenti front-end.

Un frammento, secondo questo pattern, non è altro che un server HTTP che si occupa del render di una parte della pagina specifica, oltre a fornire le risorse Javascript e il CSS.

Il Server Side Composition è una tecnologia molto utile in quanto, basandosi sul Server Side Rendering, garantisce una buona ottimizzazione per il SEO, che per alcune applicazioni è fondamentale, oltre a delle ottime performance in termini di caricamento della pagina, dato che il browser riceve la pagina già assemblata dal server.

D'altro canto, invece, bisogna considerare che l'utilizzo di questa metodologia comporta un aumento delle risorse necessarie sul server, per cui vi è il rischio di diminuire la scalabilità e le performance dell'applicazione stessa, soprattutto nel caso di applicazioni di grandi dimensioni. Inoltre, la composizione lato server senza l'utilizzo di librerie specifiche non garantisce un'ottima isolazione nel browser in termini di layout, per cui bisogna ottimizzare il CSS sul front-end in maniera tale da evitare collisioni tra frammenti diversi.

4.5.4 Client-side composition

Per molte applicazioni, il tempo di caricamento al primo accesso non è uno dei requisiti fondamentali per la distribuzione dell'applicazione web, come ad esempio nel caso di applicazioni che necessitano di gestire numerosi input da parte dell'utente: il Server Side Rendering, infatti, non è l'ideale per questa tipologia di applicazioni, in quanto ad ogni input dell'utente corrisponde un caricamento dell'intera pagina, che potrebbe influenzare negativamente l'User Experience del consumatore finale.

Per questi motivi, sono stati sviluppati diversi framework e librerie che permettono di effettuare la composizione della pagina direttamente sul client, che si occupa di generare il codice HTML e di aggiornarlo a seguito delle interazioni dell'utente.

Nell'architettura front-end basata sul client-side composition, l'obiettivo è gestire all'interno della pagina le singole microapplicazioni sviluppate e fornite dai singoli team dedicati, che saranno renderizzate e aggiornate in maniera indipendente dal resto degli altri componenti della pagina, indipendentemente dal framework o dalla libreria scelta per lo sviluppo del componente.

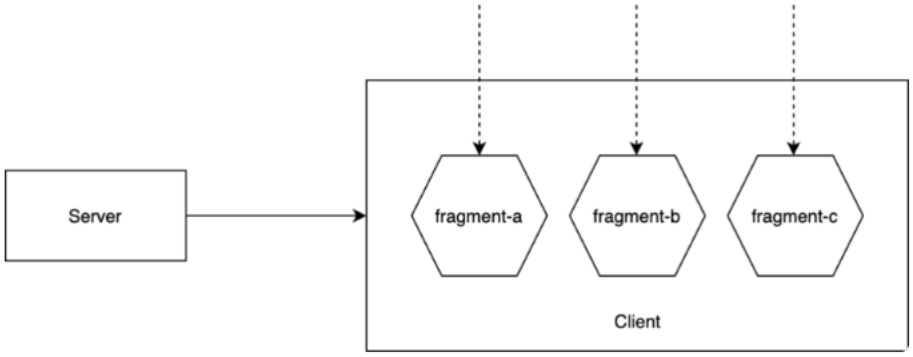


Figura 4.10 Client Side Composition

Una metodologia per realizzare il client side composition è quelle dei Web Component, che permettono di incapsulare i componenti che compongono l'interfaccia utente, fornendo delle API browser che consentono di definire l'interfaccia finale della pagina.

Tramite l'utilizzo dei Web Component, infatti, viene definito uno standard per la creazione dei componenti che utilizza HTML e un'API specifica chiamata DOM API, che ciascun micro front-end deve utilizzare per gestire la coordinazione con gli altri microfrontend.

L'utilizzo dei Web Component prevede l'utilizzo di specifiche API Javascript che definiscono le modalità di creazione dell'HTML dei singoli elementi, oltre che l'esistenza di uno "Shadow DOM" che appartiene ai singoli componenti che devono essere renderizzati. Inoltre, i Web Component prevedono l'utilizzo di tag HTML specificamente creati per la realizzazione del template della pagina.

I singoli componenti vengono quindi creati con la tecnologia ritenuta più idonea dai singoli team, per poi venire incapsulati dentro dei Web Components che possono essere utilizzati all'interno del contenitore padre dell'applicazione renderizzata sul client, dopo che questi sono stati dichiarati come elementi da poter inserire all'interno dei Web Component tramite la specifica API Javascript.

Per definire un Web Component, il componente deve essere innanzitutto definito come estensione di HTMLElement, per poi renderlo disponibile tramite le funzioni dell'API.

```
class CustomButton extends HTMLElement{
...
}

customElements.define("custom-button", Custom);
```

Dopo aver definito il componente come tale, si può procedere con l'inserimento del componente all'interno del progetto principale tramite l'utilizzo del tag <custom-button> definito con la funzione customElements.define.

La possibilità di realizzare degli Shadow DOM, inoltre, permette di isolare e collegare dei DOM "nascosti" collegati al DOM principale; gli elementi del DOM principale, però, non hanno accesso agli elementi del DOM nascosto, per cui è possibile isolare il CSS e il codice Javascript senza che vi siano interferenze, ad esempio, da parte del CSS definito nel contenitore padre.

L'utilizzo di questa tipologia di composizione dell'applicazione web permette di realizzare applicazioni i cui componenti possono essere realizzati con più framework, dato che i Web Components possono convivere all'interno della stessa pagina a prescindere dalle tecnologie utilizzate per la loro realizzazione; inoltre, dato che i componenti vengono poi trasformati in semplice codice Javascript, HTML e CSS, essi sono compatibili con tutti i browser moderni. Infine, data la natura dei Web Components, è possibile riutilizzarli all'interno di pagine diverse.

Tuttavia, dato che i Web Components vengono gestiti e renderizzati sul client, il caricamento della pagina potrebbe richiedere tempi più lunghi, dato che la gestione dei componenti è gestita direttamente sul client. Inoltre, senza l'utilizzo di librerie specifiche, i Web Components non sono ottimizzati per il SEO.

4.5.5 Framework

Nel corso degli ultimi anni, sono stati sviluppati diversi framework per ottimizzare e velocizzare la creazione di applicazioni a micro front-end secondo approcci diversi, ciascuno dei quali con le proprie convenzioni e best practices per lo sviluppo dell'applicazione.

5. Il Framework Astro: un esempio applicativo

5.1 Astro

Astro è un framework per la realizzazione di MPA (Multi Page Application) che permette di renderizzare componenti sviluppati tramite l'utilizzo di diversi framework all'interno della stessa pagina.

Tramite Astro, infatti, è possibile generare pagine HTML statiche renderizzati lato server. I componenti che necessitano di esecuzione di codice JavaScript lato client, invece, vengono caricati separatamente insieme alle loro dipendenze e vengono gestiti come se fossero all'interno di una SPA.

5.2 Island Architecture

Le pagine realizzate tramite il framework Astro si basano sul pattern architetturale delle "Astro Island", termine che si riferisce ai singoli componenti interattivo su una pagina HTML: all'interno della pagina, possono coesistere più isole contemporaneamente, e il render di ciascuna isola è separato dal render delle altre isole. Il concetto di "Island Architecture" è stato coniato da Katie Sylor-Miller, uno sviluppatore front-end di Etsy, un marketplace online.

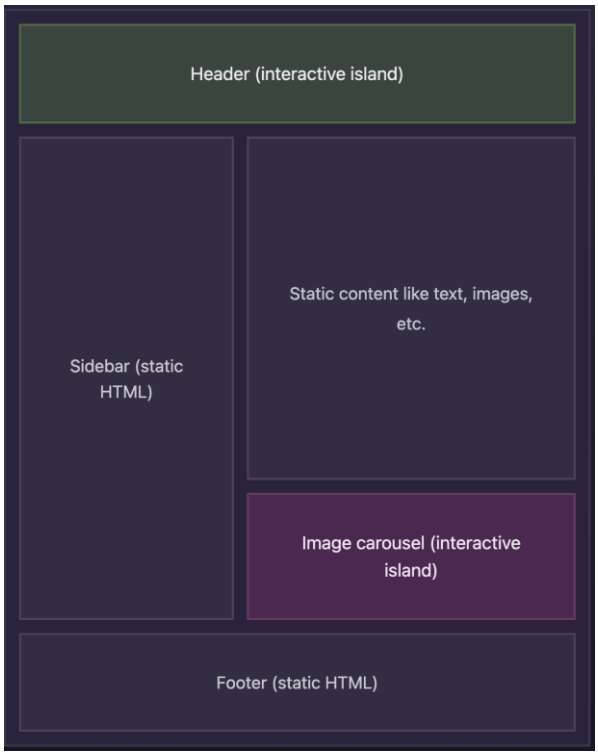


Figura 5.1 Island Architecture: isole dinamiche e isole statiche

Le isole, inoltre, permettono di utilizzare framework UI diversi all'interno della stessa pagina per renderizzare i componenti, che possono quindi essere sviluppati con framework differenti (React, Vue, Svelte, SolidJs…).

L'idea dell'Island Architecture è, quindi, quella renderizzare le pagine HTML sul server, iniettando all'interno dell'HTML stesso delle regioni dedicate a contenere l'output dei corrispondenti widget che vengono inseriti in pagina. Tali regioni, possono quindi essere poi popolate lato client con dei widget autonomi, riutilizzando l'HTML iniziale renderizzato dal server.

Renderizzare delle pagine utilizzando l'architettura ad isole permette quindi di avere delle porzioni di pagina altamente dinamiche che vengono inizializzate separatamente: le singole aree della pagina diventano quindi interattiva senza necessariamente attendere il caricamento di altri componenti in pagina.

Tramite quest'architettura, è quindi possibile creare delle applicazioni web che trovino il giusto compromesso tra l'interattività di un'applicazione il cui codice viene eseguito lato client e i benefici in termini di ottimizzazione SEO tipici delle applicazioni renderizzate sul server.

5.3 Caratteristiche del framework

Astro è un framework che nasce per lo sviluppo specifico di pagine web rapide e focalizzate sulla visualizzazione di molti contenuti, come ad esempio siti di marketing, blog, portfolio ed e-commerce. A differenza dei moderni framework, che sono ideali per la creazione di web application molto più complesse, Astro si concentra principalmente sui contenuti. Essendo un framework content-based, Astro offre delle prestazioni molto superiori rispetto a framework concentrati più sulla struttura dell'applicazione.

A tal fine, Astro privilegia il più possibile il rendering lato server e il modello architetturale delle MPA rispetto al rendering lato client. Infatti, il rendering lato client, tipico delle SPA, oltre ai suoi numerosi vantaggi già citati presenta delle criticità in termini di performance e SEO che non avrebbe senso introdurre all'interno di pagine web content-based in cui il caricamento iniziale e la realizzazione di un SEO ottimale sono essenziali.

Astro genera di default i siti web senza l'utilizzo di Javascript lato client. Se un componente è realizzato con un framework (React, Svelte, Vue ecc.), questo viene quindi trasformato in semplice codice HTML da Astro, permettendo quindi di mantenere il sito web rapido e veloce rimuovendo tutto il codice Javascript inutilizzato dalla pagina, migliorando di molto le performance della pagina web in quanto il codice Javascript è uno degli asset più pesanti da caricare e gestire.

Se, invece, è necessario del codice lato client per la creazione di un'interfaccia interattiva, Astro richiede la creazione di un'isola: solo per il componente renderizzato all'interno dell'isola, quindi, verrà fornito anche il codice Javascript da eseguire lato client per la gestione degli input dell'utente. Tale processo è definito come "partial hydration", e consiste nel fornire il codice Javascript eseguibile lato client solamente a dei componenti specifici.

5.4 Progettazione

Con lo scopo di realizzare una dimostrazione pratica sulla realizzazione di un'applicazione basata su un'architettura a micro front-end, è stato scelto di utilizzare il framework Astro per lo sviluppo di alcune delle pagine appartenenti ad un ipotetico e-commerce, sulla base del modello visualizzato nel cap 4.2.

Per i dati utilizzati per lo sviluppo, poiché l'obiettivo del progetto è quello di dimostrare le potenzialità, la robustezza e le modalità di comunicazione appartenenti a un'applicativo front-end sviluppato con architettura a micro front-end tramite il framework Astro, è stata utilizzata un'API online gratuita con dei contenuti pseudo-reali per la realizzazione di un'e-commerce, i cui dati sono stati reperiti tramite una chiamata REST al seguente endpoint: https://fakestoreapi.com/products/, che restituisce un elenco di prodotti in formato json contenenti i dati principali per la realizzazione del prototipo, quali nome del prodotto, descrizione, prezzo, categoria e immagine. A partire da questa base di prodotti, quindi, è stata sviluppata l'applicazione front-end.

A tale scopo, sono state quindi sviluppate la pagina contenente l'elenco di prodotti disponibili all'acquisto e la pagina con i dettagli del prodotto selezionato, ipotizzando un'ipotetica suddivisone in tre team di sviluppo concentrandosi sulle relative fasi di acquisto della customer journey:

1) Team Decide: al team Decide sono collegati tutti quei componenti che si occupano di fornire informazioni dettagliate del prodotto selezionato, permettendo quindi all'utente di ottenere maggiori informazioni per effettuare un acquisto consapevole;
2) Team Inspire: al team Inspire, invece sono correlati i componenti che hanno l'obiettivo di fornire dei suggerimenti di acquisto all'utente, permettendo quindi di avere una visione più ampia della gamma di prodotti disponibili e correlati al prodotto scelto;
3) Team Checkout: team dedicato allo sviluppo e alla realizzazione dei componenti relativi alla fase di acquisto vero e proprio, dall'aggiunta al carrello alla modalità di pagamento.

Secondo la filosofia dell'architettura a micro front-end, inoltre, ogni team utilizzerà lo stack di sviluppo che ritiene più opportuno, senza vincolarsi alle scelte tecnologiche degli altri team.

In particolare, è stato scelto di sviluppare i componenti relativi al team Decide utilizzando il framework Svelte.js, quelli del team Inspire tramite il framework Preact.js, mentre quelli appartenenti al team Checkout saranno sviluppati con la libreria React.js.

Tutti i componenti sviluppati sono stati quindi integrati tramite il framework Astro all'interno delle singole pagine, permettendo quindi la convivenza dei vari componenti sviluppati con tecnologie diverse e indipendenti grazie all'inizializzazione di isole dedicate, oltre all'inserimento di componenti statici realizzati direttamente tramite la sintassi di Astro.

Per la pagina relativa alla lista di prodotti disponibili è stato ideato il seguente wireframe, seguito dalla corrispondente struttura dell'architettura secondo il modello delle Island Architecture precedentemente descritto.

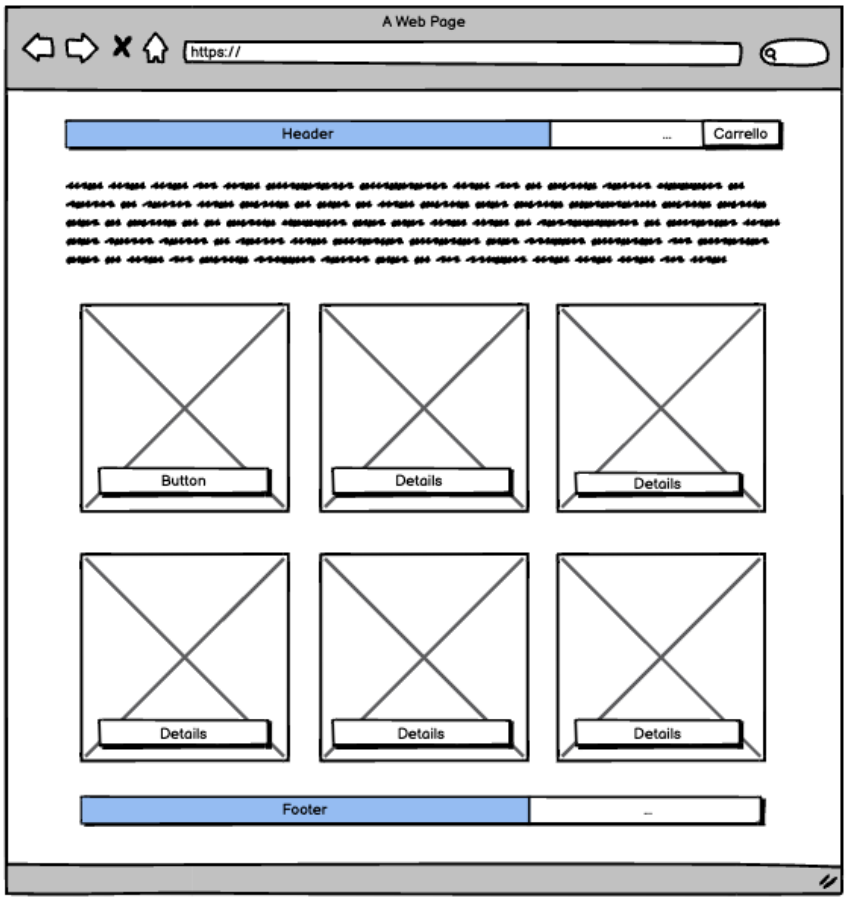


Figura 5.2 Wireframe pagina elenco prodotti

Seguendo il pattern architetturale di Astro, dunque, all'interno della seguente pagina sono stati istanziati tre componenti statici (Header, Componente testuale e Footer), e due componenti dinamici, uno per la visualizzazione delle singole card relative ai vari prodotti, con i quali si potrà interagire per selezionare il prodotto di cui si vorrà visualizzare i dettagli, e l'altro per il componente relativo al carrello.

Il primo componente appartiene all'area di lavoro del Team Inspire, in quanto si occupa della visualizzazione dei vari prodotti, ed è stato dunque sviluppato utilizzando Preact.js; il secondo, invece, appartiene al team Checkout, ed è stato sviluppato in React.

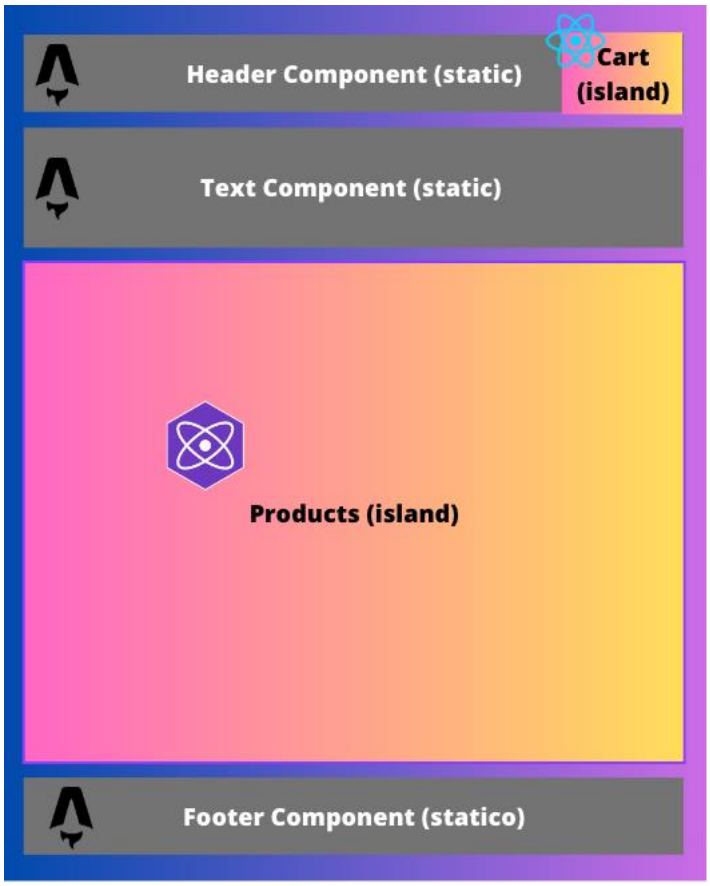


Figura 5.3 Island Architecture - pagina elenco prodotti

Lo stesso ragionamento in termini di layout è stato poi effettuato relativamente alla pagina di dettaglio prodotto, all'interno del quale saranno presenti componenti appartenenti a ciascuno dei team indicati. Infatti, all'interno della pagina di dettaglio, sono stati inseriti un componente dedicato alla visualizzazione dei dettagli del prodotto (Team Decide), un componente relativo ai prodotti correlati alla categoria del prodotto selezionato (Team Inspire) e il componente dedicato all'aggiunta e alla visualizzazione del carrello (Team Checkout). Inoltre, come nella pagina dedicata all'elenco dei prodotti disponibili, dei componenti statici sviluppati tramite il framework Astro che non richiedono l'istanziamento di un'isola dinamica, ovvero l'Header, il Footer e un componente testuale.

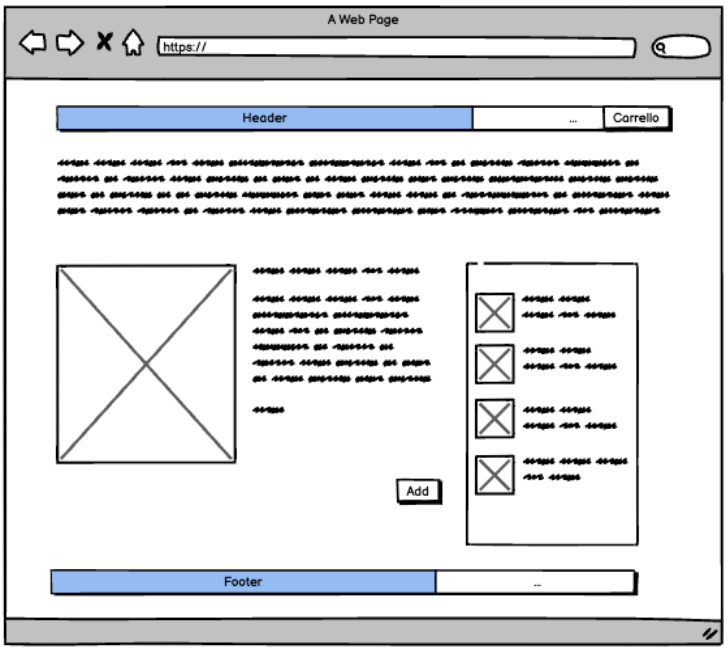


Figura 5.4 Wireframe pagina dettaglio prodotto

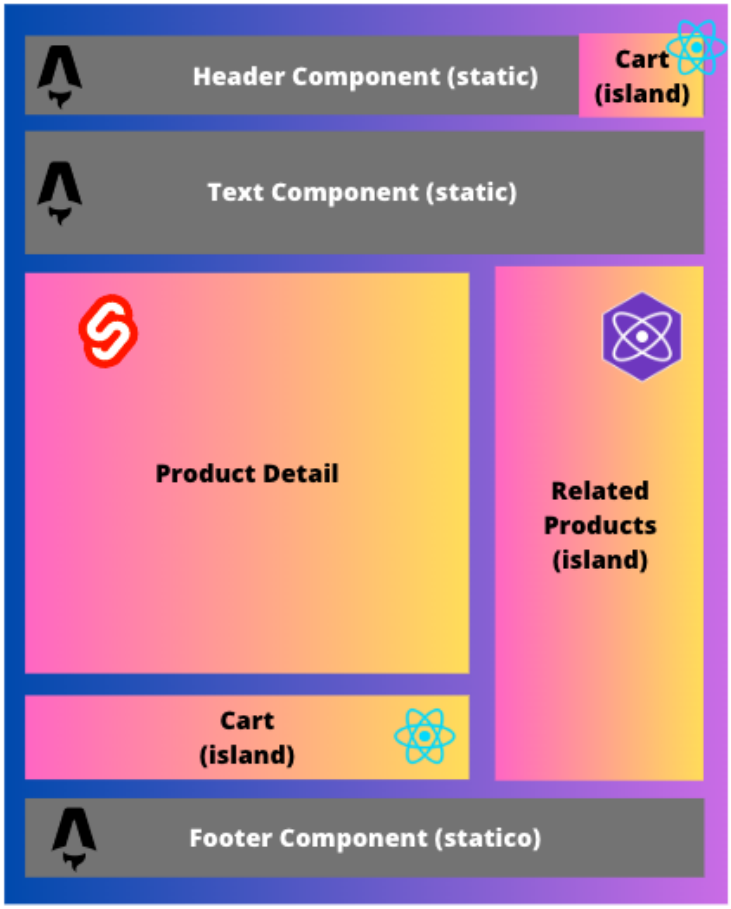


Figura 5.5 Island Architecture pagina dettaglio prodotto

5.5 Implementazione

Per l'inizializzazione del progetto, è stato utilizzato npm (Node Package Manager) e il rispettivo comando per l'inizializzazione di un progetto Astro, che permette di configurare diversi parametri e di selezionare tra JavaScript e TypeScript come linguaggio di programmazione. Il progetto in questione, in particolare, è stato realizzato usando JavaScript.

```
samuelemannala@MacBook-Air-di-Samuele astro % npm create astro@latest
```

Per inizializzare un server Astro per l'esecuzione del codice in locale, invece, si utilizza il comando "npm run dev". Il server Astro, di default, verrà inizializzato sulla porta 3000, per cui, per visualizzare il programma sul browser, è necessario collegarsi a https://localhost:3000.

La struttura base di un progetto Astro è composta dalle seguenti cartelle e dai seguenti file:

- src/: codice sorgente del progetto (componenti, pagine, stile);
- public/: cartella contenente assets;
- package.json: file contenete le dipendenze del progetto;
- astro.config.mjs: file di configurazione Astro;

La struttura della cartella src è stata organizzata in maniera tale da suddividere semanticamente le aree di sviluppo degli ipotetici team, per cui, all'interno della cartella components, sono state create tre sotto-cartelle "decide", "inspire", "checkout", contenenti i componenti appartenenti alle relative aree di sviluppo e sviluppati con la corrispondente libreria o il corrispondente framework, oltre ai componenti statici sviluppati secondo la sintassi di Astro.

Figura 5.6 Struttura cartella src contenente i componenti dell'applicazione

All'interno della cartella src, inoltre, è fondamentale per l'esecuzione del progetto l'esistenza della

cartella “pages” contenente le singole pagine, che possono essere o dei file HTML o dei componenti Astro. Per il progetto di tesi, è stato scelto di sviluppare le singole pagine secondo la sintassi dei componenti di Astro, così da poter effettuare gli import dei componenti e snellire la scrittura del codice di markup tramite l’utilizzo di un Layout Astro, che ha permesso di definire un template base condiviso tra le varie pagine, contenente l’Header e il Footer dell’applicazione.

5.5.1 Componenti Astro

I componenti Astro costituiscono la base fondamentale dei progetti sviluppati tramite Astro, e sono dei componenti che vengono letti dal browser come dei semplici file HTML, senza la possibilità di eseguire del codice a runtime sul client, oltre a venire renderizzati esclusivamente sul server secondo i principi del SSR.

I componenti Astro sono composti da due aree principali: lo Script e il Template. L’area dedicata agli script è delimitata da “---” ed è dedicata all’implementazione di codice JavaScript per l’esecuzione di codice Javascript sul server, come import di altri componenti (che possono essere anche sviluppati tramite framework o librerie), di fogli di stile CSS o per il fetch di dati. La sezione dedicata al template di pagina, invece, è dedicata alla definizione dell’HTML del componente, all’interno del quale è possibile utilizzare Javascript Expressions, componenti importati o tag specifici.

I componenti Astro, inoltre, sono progettati per essere riutilizzaabili e componibili, e possono prevedere la definizione di proprietà da fornire al componente per la sua renderizzazione.

All’interno del progetto, ad esempio, è stato definito il componente “TextComponent.astro”, un semplice componente per la visualizzazione di un blocco testuale. Tale componente, ad esempio, si aspetta che gli venga passata una props chiamata “text”, che verrà poi utilizzata per il render del componente.

```
// src/components/TextComponent.astro
const {text = 'This is my microfrontend project'} = Astro.props;

<div class='text-component'>
   {text}
</div>


<style>
   .text-component{
      background-color: #f2f2f2;
      margin: 12px 12px 24px 12px;
      font-size: 16px;
      font-weight: 300;
      padding: 8px
   }
</style>
```

All'interno dei componenti Astro, inoltre, è possibile utilizzare il tag <slot/>, che è un placeholder per contenuti HTML esterni che possono essere inseriti all'interno del componente stesso.

Nel Base Layout che compone le pagine del progetto, ad esempio, è stato inserito il tag <slot/> per il posizionamento degli elementi che andranno a comporre le singole pagine.

5.4.2 Componenti sviluppati mediante framework o librerie

All'interno del progetto, data l'ipotetica differenza tra gli stack tecnologici di ciascun team, sono stati inseriti componenti sviluppati con diversi framework: Astro, infatti, consente l'import, all'interno delle pagine e all'interno dei componenti .astro stessi, di componenti sviluppati con librerie o framework esterni.

Poiché Astro, di default, effettua il rendering sul server, per rendere un componente realizzato, ad esempio, tramite React, è necessario istanziare un'isola nel momento in cui si decide di utilizzare il componente tramite il comando "client:" da inserire all'interno del tag del componente.

Esistono diversi comandi sulla base del funzionamento desiderato: ad esempio, se si vuole che il caricamento del codice Javascript venga eseguito nel momento in cui termina il caricamento della pagina, bisogna utilizzare la direttiva client:load; se, invece, si vuole rendere disponibile il codice da eseguire lato client solo nel momento in cui il componente in questione diventa disponibile in pagina, bisogna utilizzare la direttiva client:visible.

All’interno del progetto, poiché sono stati utilizzati tre framework diversi, è stato necessario installare le dipendenze per le varie librerie tramite linea di comando npm apposito:

```
npx astro add svelte
npx 74stro add react
npx 74stro add preact
```

Effettuata tale operazione, è stato quindi possibile creare i componenti seguendo le sintassi dei singoli framework o librerie scelti all’interno delle cartelle dedicate ai singoli team. Per lo styling dei componenti, inoltre, è stato utilizzato SCSS, un preprocessore CSS.

5.4.3 Gestione e condivisione dei dati

Per la gestione e la condivisione dei dati restituiti dall’API, è stato creato un file shared.js che si occupa di effettuare la chiamata Rest all’endpoint per l’inizializzazione dei valori relativi ai singoli prodotti dell’e-commerce, che vengono resi disponibili all’interno dei singoli componenti e delle singole pagine.

All’interno della pagina di dettaglio prodotto, inoltre, è stato necessario implementare un meccanismo di comunicazione e di condivisione dei dati che permettesse il corretto funzionamento del componente dedicato alla visualizzazione dei prodotti correlati, nonché alla condivisione e all’aggiornamento di tale valore.

Per poter condividere tale valore, è stato implementato il meccanismo di condivisione dei dati direttamente sul client tramite la libreria Nano Stores, che permette di condividere valori tra componenti a prescindere dalla tecnologia con cui questi sono stati realizzati.

La libreria è stata quindi installata nelle singole versioni per React, Svelte e Preact tramite il comando apposito da linea di comando. Per ciascun framework, infatti, è presente una sintassi specifica per l’interazione con i dati condivisi istanziati tramite la libreria selezionata.

Per la gestione dello stato condiviso, è stato quindi implementato un persistentAtom, con lo scopo di memorizzare all’interno del localStorage del browser l’id del prodotto selezionato, che viene quindi letto dai componenti presenti in pagina per effettuare le dovute operazioni.

Il componente dedicato alla visualizzazione delle ipotetiche proposte alternative, SuggestedItem.jsx (renderizzato all'interno del componente RecommendedProducts), è stato configurato in maniera tale da visualizzare i prodotti correlati, dando la possibilità di modificare il componente di cui si visualizzano i dettagli tramite un meccanismo di modifica del persistentAtom condiviso.

```
Import { h } from 'preact';
import styles from './styles.scss';
import { idSelected } from '../../shared';


export default function SuggestedItem(props) {
  const {product} = props;

  return (
    <div>
      <div class='card-suggested'>
        <div class='left'>
        <img class='image' src={`${product?.image}`}/>
        </div>
        <div class='right'>
        <div class='title' onClick={()=> {idSelected.set(product?.id)}}>{product?.title}</div>
        <div class='price'>{product?.price} $</div>
        </div>
      </div>

    </div>
  );
}
```

Inoltre, anche il componente per l'aggiunta al carrello del prodotto ha accesso all'id del prodotto selezionato, che utilizza per effettuare l'aggiunta e le modifiche al carrello, memorizzato anch'esso all'interno dello storage del browser tramite la funzione Map() della libreria.

Infine, poiché l'obiettivo era quello di testare la possibilità di inserire più framework all'interno della pagina, non è stato realizzato fino in fondo il routing dinamico con l'inserimento dell'id all'interno dell'URL della pagina, ma i contenuti variano all'interno della stessa pagina con la stessa URL; tuttavia, l'implementazione del routing dinamico sarebbe chiaramente stata opportuna nella misura in cui l'obiettivo fosse stato quello di sfruttare al meglio le possibilità del framework con lo scopo di ottimizzare il SEO della

pagina web, in quanto permetterebbe al template di pagina di avere accesso ai singoli dati del prodotto con lo scopo di ottimizzare i metadati inseriti all'interno del documento HTML.

5.4.4 Implementazione delle pagine

Come indicato, sono state realizzate due ipotetiche pagine dell'applicazione web: una per la selezione di un prodotto dall'elenco dei prodotti disponibili, e l'altra che, sulla base del prodotto selezionato, mostra i dettagli del prodotto e i prodotti correlati per categoria.

Le singole pagine sono state sviluppate come componenti Astro, e importano i componenti necessari alla costruzione della pagina, oltre all'elenco dei prodotti disponibili, disponibile all'interno di file Javascript. Il codice inserito nel frontmatter, la parte dedicata agli script dei componenti Astro, verrà quindi eseguito sul server, che si occuperà di fornire al browser la pagina web già renderizzata, salvo l'istanziamento di eventuali isole dinamiche per l'inserimento di eventuali componenti che necessitano l'esecuzione di codice Javascript sul client, come, ad esempio, il componente dedicato alla visualizzazione del singolo prodotto, in homepage, ProductCard, che ha il compito di reindirizzare alla pagina dedicata al prodotto selezionato, modificando quindi il valore condiviso dell'id del prodotto selezionato oltre ad avviare il routing verso la pagina di dettaglio prodotto.

Le singole pagine, dunque, sono state create sulla base dei wireframe e dell'Island Architecture visualizzati all'interno del capito 5.4.

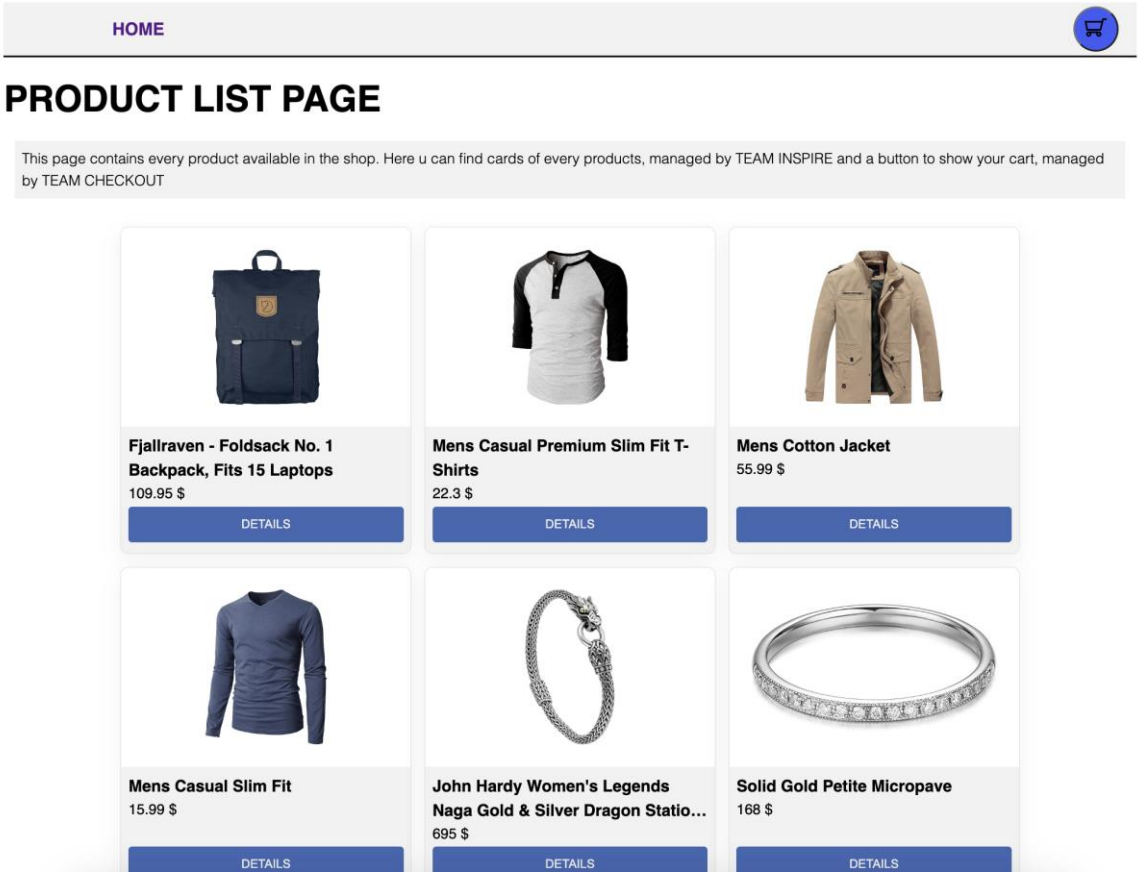


Figura 5.7 Output pagina elenco prodotti

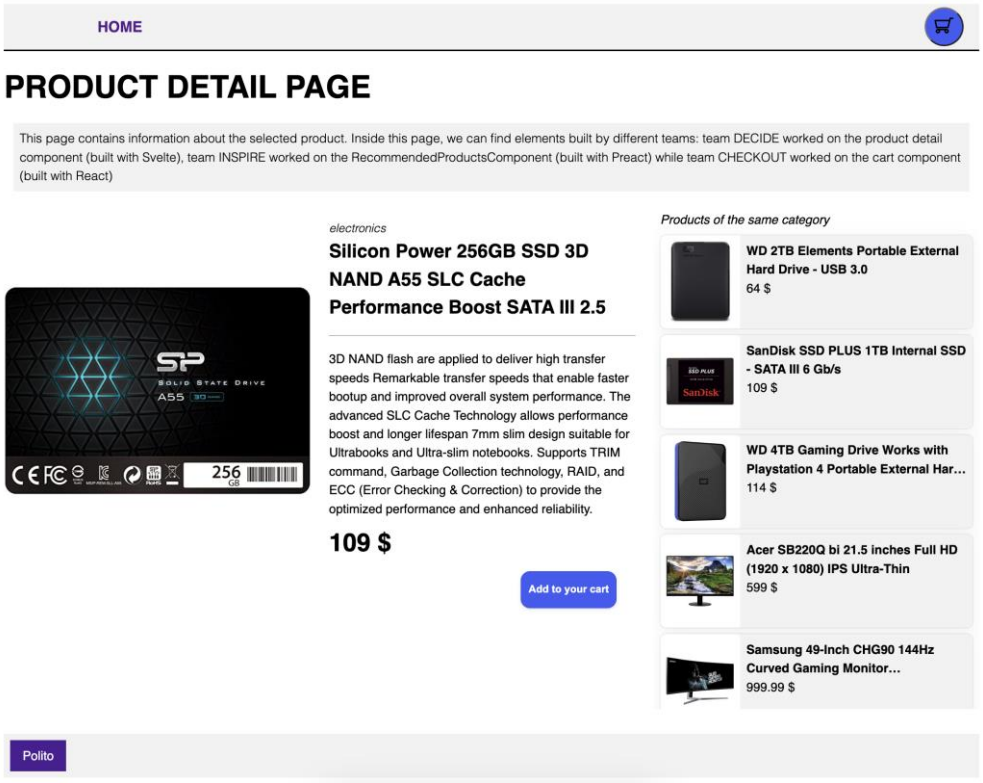


Figura 5.8 Output pagina dettaglio prodotto

Il layout, delle pagine, quindi, è stato realizzato prevedendo l'istanziamento, per i componenti sviluppati tramite Svelte, React e Preact, di isole dinamiche, data la necessità di eseguire codice Javascript sul client, per la modifica dei dati e la gestione di stati interni ai componenti. Ad esempio, il componente del carrello, inglobato all'interno dell'Header tramite un'isola specifica, presenta uno stato interno gestito tramite l'Hook specifico di React per l'apertura e la chiusura del popup dedicato alla visualizzazione dei prodotti all'interno del carrello.

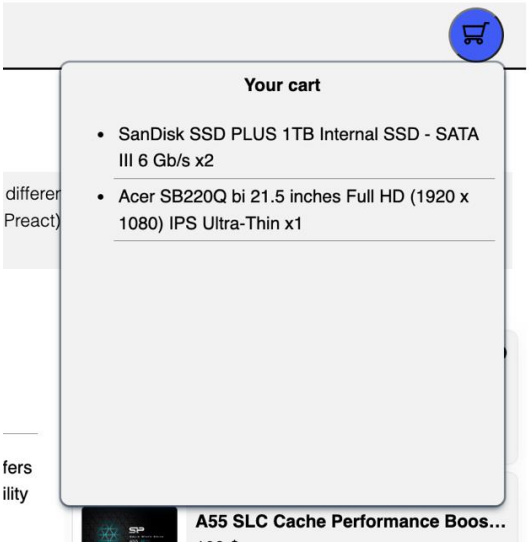


Figura 5.9 Dettaglio carrello aperto

5.4.5 Verifica della robustezza dell'applicazione

Come già spiegato, uno dei vantaggi principali dell'architettura a micro front-end è la grande indipendenza, in termini di deploy e gestione degli errori, dei singoli componenti che compongono l'architettura dell'applicazione.

Per avere una conferma di tale caratteristica, quindi, è stato volutamente inserito un errore all'interno del codice del componente del carrello, Cart.js (dentro la cartella di riferimento components/checkout), in particolare all'interno del corpo della funzione dedicata alla visualizzazione del popup contenente i prodotti inseriti nel carrello.

All'interno di un'applicazione tradizionale, ad esempio con un'architettura SPA basata interamente su React, ci aspetteremmo, giustamente che l'innescarsi di un errore dovuto ad un typo o a un errore nella scrittura del codice comporti l'interruzione dell'intera applicazione, in quanto i singoli componenti della pagina sono strettamente interconnessi.

Nel caso dell'applicazione sviluppata tramite il framework Astro, invece, l'innescarsi di un errore non determina l'interruzione dell'applicazione nella sua totalità, ma solamente l'eliminazione dal DOM della pagina dell'isola corrispondente al componente all'interno del quale è presente il suddetto errore. In particolare, nel caso del file Cart.js, l'interazione con il componente del carrello all'interno dell'Header eliminerebbe dal DOM il componente stesso, permettendo comunque di effettuare la navigazione all'interno della pagina e le restanti operazioni. Chiaramente, la presenza di un errore del genere, dovrebbe essere gestita

con un sistema di gestione degli errori, in quanto, nel nostro caso specifico, l'eliminazione del componente del carrello comporterebbe necessariamente un'interruzione del flusso di acquisto degli utenti.

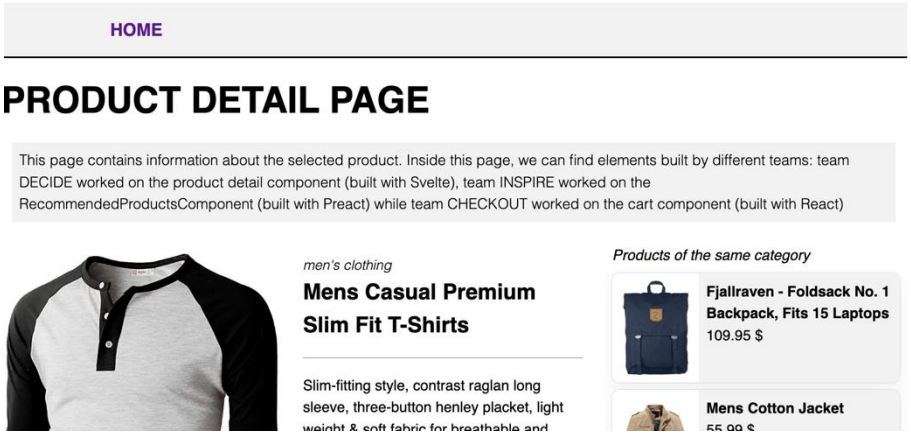


Figura 5.10 Output al rilevamento dell'errore sul carrello

6. Casi studio

Come già preannunciato, le tipologie di aziende in grado di trarre il maggior numero di benefici tramite l'utilizzo dell'architettura a micro front-end sono quelle con un'ampia presenza online e con un elevato traffico, come ad esempio e-commerce, piattaforme di prenotazioni online, social network e applicazioni web complesse: il vantaggio principale che queste realtà possono sfruttare è, soprattutto, quello della scalabilità dovuta alla possibilità di suddividere l'applicazione in moduli indipendenti, permettendo loro di effettuare modifiche e rilasciare in modo rapido e indipendente, oltre che un miglioramento significativo della modalità di gestione delle risorse.

Le grandi aziende della realtà digitale, spesso, si sono quindi fatte promotori dell'architettura a micro front-end, utilizzando metodologie già indicate all'interno del lavoro di tesi o sviluppando internamente dei framework per avere un controllo più personalizzato delle modalità di creazione della pagina web.

6.1 Netflix

Una grande azienda che fa utilizzo di un'architettura a micro front-end è Netflix, uno dei più importanti servizi di streaming in abbonamento, che ha deciso di adottare l'architettura a micro front-end sviluppando un framework interno a tale scopo: Lattice.

Lattice è un framework interno sviluppato e utilizzato da Netflix con lo scopo di snellire le dipendendenze interne al progetto permettendo un utilizzo più semplice di stati condivisi tramite l'utilizzo di Webpack, all'interno del loro client in cui lo stack principale è composto da React e Typescript. L'obiettivo, nello sviluppo del framework, è stato proprio quello di creare un framework che si basasse sullo stack utilizzato internamente all'azienda, per cui Lattice è un framework basato su React, in modo tale da poter adottare in maniera semplice e rapida tale tecnologia.

Lattice, dunque, è un framework per estendere applicazioni React, che prevede l'inserimento di un layer aggiuntivo, chiamato "Abstraction Layer", che ha lo scopo di iniettare i componenti o gli stati, andando inoltre a disaccoppiare il Business Layer dal Presentation Layer.

L'utilizzo di Lattice prevede che gli sviluppatori si concentrino sul loro componente principale, per poi inglobarlo in un plugin esterno, che viene utilizzato poi dal Framework per la composizione e la generazione dei singoli moduli in pagina.

6.2 Spotify

Le singole funzionalità dell'applicazione front-end desktop di Spotify sono sviluppate da team indipendenti che sviluppano da zero i singoli componenti relativi a quella determinata feature: ogni componente è quindi gestito da un team indipendente, e l'applicazione viene poi composta mediante l'utilizzo di un elevato numero di Iframes, che uniscono varie parti dell'applicazione e che comunicano tra di loro tramite dei bus di comunicazione specifici.

6.3 DAZN

Il team di sviluppo di DAZN, una piattaforma di streaming per servizi sportivi, è stata protagonista di un'elevata crescita in termini di risorse distribuite su diverse città: a causa di ciò, è stato deciso di rivedere completamente la struttura dei team di sviluppo cercando di suddividere i team di sviluppo per aree di competenza specifiche.

I punti chiave dell'architettura a micro front-end, secondo il team di sviluppo di DAZN, sono i seguenti:

1) Implementazioni indipendenti senza la condivisione di logiche;
2) Micro front-end modellati e sviluppati attorno a un Business Domain;
3) Gestione effettuata da un singolo team di sviluppo;

DAZN ha deciso di intraprendere la strada dell'architettura a micro front-end effettuando però delle premesse specifiche circa quello che è stato il suo obiettivo nell'implementazione dell'applicazione.

Innanzitutto, DAZN utilizza un singolo micro front-end per pagina, con lo scopo di non avere la necessita di gestire dipendenze tra più micro front-end che convivono contemporaneamente nello stesso contenitore. I micro front-end di DAZN, dunque, sono di medie dimensioni rispetto, ad esempio, a quelle implementate nel progetto Astro del capitolo precedente.

Per quanto riguarda quindi il routing, in DAZN viene effettuato effettuando il routing da un micro front-end a un altro, effettuando il caricamento di un micro front-end per volta. Inoltre, viene utilizzata un API specifica per la condivisione dei dati tra i vari micro front-end, così da poter gestire eventuali stati condivisi tra i componenti.

I singoli micro front-end, i cui output potenzialmente potrebbero essere delle SPA, sono gestiti e orchestrati da bootstrap, una libreria Javascript inclusa all'interno della pagina HTML principale e che quindi è disponibile in qualsiasi momento del ciclo di vita dell'applicazione, che si occupa del caricamento dei singoli micro front-end.

Bootstrap, dunque, funge da Client Side Orchestrator, in quanto è sempre presente in pagina e si occupa di rispondere alle richieste dell'utente effettuando il download del micro front-end richiesto a seconda degli input dell'utente.

7. Conclusioni

L'adozione dell'architettura a micro front-end per lo sviluppo di un'applicazione non si implica solamente l'esecuzione di una scelta tecnologica, ma anche di una scelta organizzativa che determina una rivoluzione nelle modalità di lavoro e di sviluppo che sono state lo standard all'interno delle aziende tech fino al giorno d'oggi.

Con l'adozione consapevole di tale architettura, infatti, la suddivisione in team di sviluppo, a ciascuno delle quali viene assegnata una specifica area di competenza oltre che la totale libertà in termini di scelte tecnologiche, permette la nascita di team esperti al cento per cento sulle singole funzionalità del progetto, oltre alla possibilità di ottimizzare al meglio le pipeline di test e distribuzione dei singoli componenti con maggiore facilità e libertà.

Chiaramente, l'architettura a micro front-end non deve essere vista come la soluzione definitiva per lo sviluppo delle applicazioni web, in quanto esistono tanti casi in cui probabilmente la realizzazione di un'architettura secondo tale filosofia risulterebbe superflua in termini di funzionalità e di dimensioni del progetto, e potrebbe determinare un rallentamento e un incremento delle difficoltà nello sviluppo, soprattutto durante le fasi iniziali del progetto.

Infatti, l'adozione di tale architettura potrebbe essere la soluzione ideale per applicazioni di grandi dimensioni, come ad esempio grandi e-commerce o social network, in cui è presente un elevato numero di sviluppatori dedicati all'applicazione, motivo per cui suddividere l'applicazione in tante micro-applicazioni potrebbe determinare una maggiore produttività e una maggiore capacità di comunicazione interna.

Per le grandi applicazioni, dunque, grazie ai suoi benefici, l'adozione dell'architettura a micro front-end rappresenta un'importante opportunità per ottenere un maggiore vantaggio competitivo all'interno del mercato digitale, costellato da un infinito numero di tecnologie sempre in continua evoluzione.

8. Bibliografia

- Signorelli, A. D. (2019, March 11). *Storia di internet e del world wide web*. Wired. https://www.wired.it/internet/web/2019/03/11/internet-world-wide-web-storia/?refresh_ce=
- *Web technology for developers | MDN*. (2023). https://developer.mozilla.org/en-US/docs/Web
- Wanyoike, M. (2019). History of front-end frameworks - LogRocket Blog. In *LogRocket Blog*. https://blog.logrocket.com/history-of-frontend-frameworks/
- N., & N. (2023, March 17). *What is Client Server Architecture?* Intellipaat Blog. https://intellipaat.com/blog/what-is-client-server-architecture/#:~:text=Client%20server%20architecture%20is%20a,are%20delivered%20over%20a%20network.
- contributori di Wikipedia. (2022, March 23). *Architettura multi-tier*. Wikipedia. https://it.wikipedia.org/wiki/Architettura_multi-tier
- *What is Three-Tier Architecture | IBM*. (n.d.). https://www.ibm.com/topics/three-tier-architecture
- La Trofa, F. (2022, January 11). *Cos'è e come funziona l'architettura monolitica*. UniverseIT. https://universeit.blog/architettura-monolitica/
- *Do You Need to Switch to a Microservices Architecture?* (2022, November 11). Lemberg Solutions. https://lembergsolutions.com/blog/do-you-need-switch-microservices-architecture
- Parravicini, C. (2022, October 14). *Decomposing Monolithic Applications using Separation of Concerns*. Intelligent Pathways.

https://intelligentpathways.com.au/decomposing-monolithic-applications-using-separation-of-concerns/

- *Rendering on the Web*. (2019, February 6). web.dev. https://web.dev/rendering-on-the-web/#:~:text=Client%2Dside%20rendering%20(CSR)%20means%20rendering%20pages%20directly%20in,and%20keep%20fast%20for%20mobile.
- robvet. (2022, November 28). *Comunicazione client front-end*. Microsoft Learn. https://learn.microsoft.com/it-it/dotnet/architecture/cloud-native/front-end-communication
- Coding Ninjas. (n.d.). *Code Studio*. https://www.codingninjas.com/codestudio/library/single-page-apps-vs-multi-page-apps
- Ly, A. (2022, March 30). *Pros and Cons Between Single Page and Multi-Page Apps*. Medium. https://andrewly.medium.com/pros-and-cons-between-single-page-and-multi-page-apps-8f4b26acd9c9
- Team, L. (2023, March 10). *Single-Page Application vs Multi-Page Application: Pros, Cons, and Which is Better?* Lvivity. https://lvivity.com/single-page-app-vs-multi-page-app
- *MPAs vs. SPAs*. (n.d.). Astro Documentation. https://docs.astro.build/en/concepts/mpa-vs-spa/#:~:text=In%20MPAs%2C%20most%20of%20your,it%20from%20a%20remote%20server.
- Jackson, C. (n.d.). *Micro Frontends*. martinfowler.com. https://martinfowler.com/articles/micro-frontends.html#PayloadSize

- Dziuba, A. (2021, August 27). *How to Scale Frontend with Micro Frontends Architecture*. Relevant Software. https://relevant.software/blog/scale-frontend-micro-frontends-architecture/#Microservices_in_the_frontend_-_why
- Mezzalira, L. (2020, September 4). *Lessons from DAZN: Scaling Your Project with Micro-Frontends*. InfoQ. https://www.infoq.com/presentations/dazn-microfrontend/#:~:text=What%20is%20a%20micro%2Dfrontend%20for%20DAZN%3F,single%2Dpage%20application%20is%20composed.
- Mezzalira, L. (2021c, December 9). *Micro-frontends, the future of Frontend architectures*. Medium. https://medium.com/dazn-tech/micro-frontends-the-future-of-frontend-architectures-5867ceded39a
- Mezzalira, L. (2021b, December 9). *Adopting a Micro-frontends architecture - DAZN Engineering - Medium*. Medium. https://medium.com/dazn-tech/adopting-a-micro-frontends-architecture-e283e6a3c4f3
- Ramirez, C. (2023, February 18). *Web Components. What are they? Pros, Cons and more.* Medium. https://medium.com/geekculture/web-components-what-are-they-pros-cons-and-more-77ad56711e49
- Verma, A. (2021, December 15). *Micro-Frontend using Web Components - The Startup - Medium*. Medium. https://medium.com/swlh/micro-frontend-using-web-components-e9faacfc101b#:~:text=Micro%20Frontend%20is%20a%20micro,cares%20about%20and%20specialises%20in.
- O. (2021, January 28). *Micro Frontends Patterns#12: Server Side Composition*. DEV Community. https://dev.to/okmttdhr/micro-frontends-patters-13-server-side-composition-1of5

- Maruti Techlabs. (2022, November 18). *What is Micro-Frontend? Benefits of Using Micro-Frontend Architecture*. Medium. https://medium.com/geekculture/what-is-micro-frontend-benefits-of-using-micro-frontend-architecture-f6d667edb03d
- *Astro Islands*. (n.d.). Astro Documentation. https://docs.astro.build/en/concepts/islands/
- *Islands Architecture - JASON Format*. (2020, August 11). https://jasonformat.com/islands-architecture/
- Gentile, V. (2022, July 8). How Micro-Frontend Architecture Improves Team Management and the Developer Experience. *Codemotion Magazine*. https://www.codemotion.com/magazine/frontend/micro-frontend-architecture/
- Geers, M. (2017, August 28). Micro Frontends. *Micro Frontends*. https://micro-frontends.org/
- Geers, M. (2020). *Micro Frontends in Action*. Manning Publications.
- Mezzalira, L. (2021a). *Building Micro-Frontends: Scaling Teams and Projects Empowering Developers*. O'Reilly Media.
- A. Montelius (2021, January 28). *An Exploratory Study of Micro FrontendsA;*