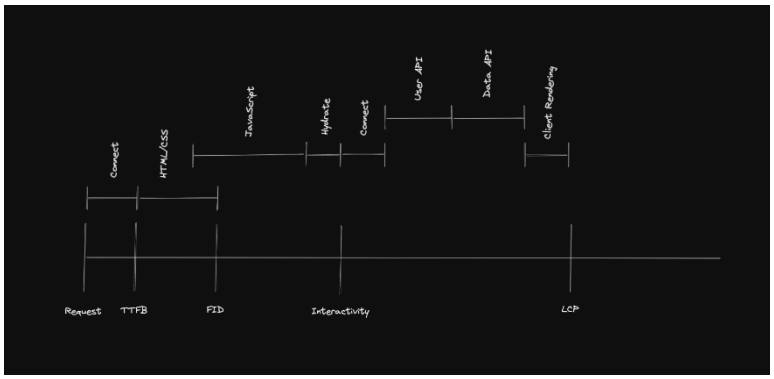
Struttura Front-end

Server-Side e Content-Side

Introduzione

Client side rendering (CSR)

Con una soluzione di rendering lato client, reindirizzi la richiesta a un singolo file HTML e il server la consegnerà senza alcun contenuto (o con una schermata di caricamento) finché non recupererai tutto il JavaScript e lascerai che il browser compili tutto prima di eseguire il rendering del contenuto.



LEGENDA

LCP: Largest Contentful Paint — misura il tempo necessario per eseguire il rendering del contenuto principale
FID: First Input Delay — misura la latenza della prima interazione che un utente ha effettuato con la pagina
CLS: Cumulative Layout Shift — misura la stabilità della pagina
TTFB: Time To First Byte — misura il tempo di configurazione della connessione e la reattività del server
FCP: First Contentful Paint — misura il tempo necessario per eseguire il rendering del primo contenuto
INP: Interaction to Next Paint — misura la latenza di tutte le interazioni che un utente ha effettuato con la pagina
TTI: Time To Interactive - misura il tempo in cui una pagina diventa interattiva (eventi collegati, ecc.).

I passaggi di questa architettura (CSR) sono:

- Client invia richiesta
- Avviene la connessione
- Viene eseguito il rendering del codice HTML/CSS
- Vengono eseguiti gli script (Scaricati JS ed eseguiti)
- “Hydration” il sito viene reso interattivo
- Recupero dati (richieste API per i contenuti)
- Rendering vero e proprio (LCP)

Problemi

Qui si può vedere che il CSR presenta alcuni problemi che può tradursi in prestazioni scadenti.
La parte più problematica è la necessità per i browser di scaricare, analizzare ed eseguire il pacchetto generato e solo successivamente essere in grado di recuperare i dati e infine visualizzare il contenuto sulla pagina. Quindi il tempo necessario per gestire questo processo **dipende dal dispositivo (e connessione) dell’utente**. Questo può essere un enorme collo di bottiglia in termini di prestazioni.

Diventa molto complicato migliorare LCP per questo tipo di applicazioni, perché è necessario eseguire tutti i passaggi uno dopo l’altro prima del rendering, anche se si prova ad ottimizzare l’applicazione con suddivisone codice e memorizzazione nella cache. Per non dimenticare che più aumenta la grandezza del nostro bundle più rallentiamo e danneggiamo l’esperienza dell’utente finale.

Il SEO è un problema con le applicazioni di rendering lato client perché i web crawler possono leggere facilmente l'HTML renderizzato dal server, ma potrebbero non aspettare per scaricare tutti i bundle JavaScript, eseguirli e attendere il completamento del recupero dei dati lato client, il che può portare a un'indicizzazione errata.

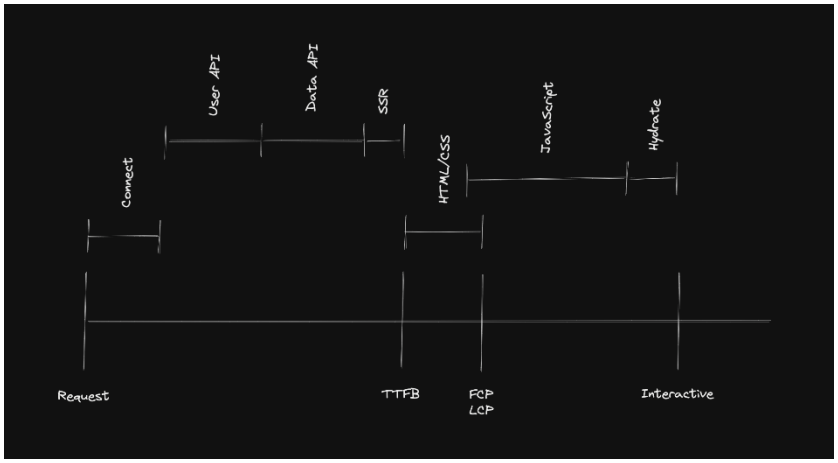
Vantaggi

Alcuni dei vantaggi del CSR sono:

- non è necessario un server per gestire l'applicazione.
- Puoi facilmente configurare l'app e pubblicarla su una CDN.
- A seconda di come l'app è strutturata e implementata, dispone di navigazioni soft che forniscono una UX migliore.

Server side rendering (SSR)

Diversamente dal rendering lato client, SSR gestisce il rendering sul server, quindi tutto (recupero dei dati, HTML generato, ecc.) viene eseguito sul server e non più sul client. SSR viene utilizzato per recuperare dati e precompilare una pagina con contenuti personalizzati, sfruttando la connessione Internet affidabile del server. Cioè, la connessione Internet del server è migliore di quella di un utente, quindi è in grado di prelevare e unire i dati prima di consegnarli all'utente.



LEGENDA

LCP: Largest Contentful Paint — misura il tempo necessario per eseguire il rendering del contenuto principale
FID: First Input Delay — misura la latenza della prima interazione che un utente ha effettuato con la pagina
CLS: Cumulative Layout Shift — misura la stabilità della pagina
TTFB: Time To First Byte — misura il tempo di configurazione della connessione e la reattività del server
FCP: First Contentful Paint — misura il tempo necessario per eseguire il rendering del primo contenuto
INP: Interaction to Next Paint — misura la latenza di tutte le interazioni che un utente ha effettuato con la pagina
TTI: Time To Interactive - misura il tempo in cui una pagina diventa interattiva (eventi collegati, ecc.).

I passaggi per questa architettura sono:

- Sul server
  - Recupera i dati per l'intera app (dati utente, dati sui contenuti)
  - Visualizza l'intera app in HTML
- Sul client
  - Carica il codice JS
  - "Hydration" (collega logica JS con HTML)

La parte **importante** è che ogni passaggio deve terminare prima che possa iniziare il passaggio successivo. Non c'è parallelismo e tutto avviene nell'ordine riportato sopra.

Vantaggi

Il **miglioramento** delle prestazioni è:

- Il recupero dei dati avviene sul server (macchina più potente e più performante del client)
- SSR genera l'HTML dei componenti sul server e lo manda al client

L'utente può già vedere un contenuto prima di caricare il bundle.

Il recupero dei dati lato server può essere più veloce del recupero dei dati lato client, in particolare per set di dati di grandi dimensioni o query complesse. Questo perché i server spesso hanno più risorse disponibili rispetto ai dispositivi client, come maggiore potenza di elaborazione, memoria e connessioni Internet più veloci:

- Dopo che la richiesta è stata inviata, abbiamo il TTFB ovvero il tempo necessario per attendere la risposta del server e il download del contenuto (risorsa)
- TTFB: 1 andata e ritorno di latenza e il tempo impiegato dal server per preparare la risposta. Si basa sul server per rispondere al sito web
- Download del contenuto: la quantità totale di tempo necessaria per leggere il corpo della risposta. Si basa sulla macchina (server e browser/dispositivi utente)

Problemi

Ma, come abbiamo visto, l'SSR anche lui alcuni colli di bottiglia in termini di prestazioni. È necessario attendere che l'intera app recuperi i dati per poter eseguire il rendering dell'HTML. E deve fare un'intera idratazione DOM.
Questo può essere un grosso problema per le metriche di interattività come FID (First Input Delay) e INP (Interaction to Next Paint).

Una **possibile soluzione** sarebbe quella di reindirizzare le parti statiche dal server senza attendere il recupero dei dati. Le parti dell'interfaccia utente che sono ancora in attesa del recupero dei dati possono visualizzare un indicatore di caricamento (spinner, skeleton) e al termine del recupero, il server esegue nuovamente lo "streaming" delle parti mancanti → Questo si chiama **Streaming SSR.**

Nuovi utilizzi

Informazioni su Streaming SSR

Lo streaming riguarda la parallelizzazione delle operazioni. L'idea è di inviare HTML in blocchi e visualizzarlo progressivamente così come viene ricevuto. Quindi, possiamo parallelizzare il lavoro sul server e inviare piccoli blocchi HTML nel tempo al client. Con questo metodo possiamo risolvere, o almeno migliorare, i due principali problemi del SSR:

- attesa del completamento del recupero dei dati per eseguire il rendering sul client
- "Hydration" come collo di bottiglia

Quindi si può eseguire già un rendering delle parti statiche mentre il server recupera i dati necessari. Pertanto il rendering della prima parte di HTML diventa piuttosto veloce e migliora il TTFB.

Naturalmente è possibile dividere l'HTML in più blocchi per poi ritrasmetterli al client appena questi sono pronto per essere reindirizzati (tramite ad esempio un **"Hydration" Selettiva**).

"Hydration" Selettiva

SSR può eseguire lo streaming di HTML in blocchi nel tempo, quindi ogni blocco HTML consegnato e sottoposto a rendering sul client può essere idratato immediatamente. Non è necessario attendere il rendering del secondo e dei blocchi successivi.

È molto **IMPORTANTE** però, avere a mente che quando ogni blocco arriva al browser, affinché possa essere renderizzato e idratato, il thread principale deve essere libero di gestire il lavoro. Se è occupato, non è possibile lavorarci sopra, a meno che non venga data la priorità.

Vantaggi

- Server parallelizzato che esegue il rendering dell'HTML in blocchi più piccoli
- Idratazione selettiva per ogni blocco

- Migliori TTFB e FCP a causa del blocco HTML più piccolo trasmesso in streaming
- Migliori TTI, FID e INP grazie all'idratazione selettiva dei pezzi più piccoli

In termini di implementazione, utilizziamo una simpatica astrazione chiamata “React Suspense”.

Server-Side Generation

La generazione lato server (SSG) è un approccio ibrido che combina i vantaggi di SSR e CSR. In questo approccio, il server genera file HTML statici per ogni pagina, ma include anche JavaScript lato client che può essere utilizzato per aggiornare la pagina secondo necessità.

Un esempio di un popolare framework SSG è Gatsby. Con **Gatsby**, puoi scrivere codice React e generarlo automaticamente in file HTML statici, offrendo i vantaggi di SSG senza dover gestire tu stesso il server.

Vantaggi

- Tempi di caricamento iniziale rapidi
- Aggiornamenti dinamici secondo necessità
- Può fornire un'esperienza utente migliore agli utenti con connessioni Internet più lente o dispositivi meno potenti

Svantaggi

- Può essere più complesso da configurare e gestire
- Potrebbe non essere adatto per applicazioni che richiedono aggiornamenti in tempo reale

Utilizzo

Quando un utente richiede una pagina, il server genera un file HTML statico per quella pagina, insieme a tutti i file JavaScript richiesti. Il client può quindi visualizzare immediatamente la pagina, senza dover attendere ulteriori richieste del server. Il JavaScript lato client può essere utilizzato per aggiornare la pagina secondo necessità.

Vercel Commerce

Informazioni

Vercel commerce utilizza:

- React Server Components (RSCs) and Suspense
- New fetching and caching paradigms

React Server Component

Ti consente di scrivere un'interfaccia utente di cui è possibile eseguire il rendering e, facoltativamente, memorizzare nella cache del server.

In Next.js, il lavoro di rendering è ulteriormente suddiviso in segmenti di percorso per abilitare lo streaming e il rendering parziale e esistono tre diverse strategie di rendering del server:

- Rendering statico
- Rendering dinamico
- Streaming

Come spiegato in precedenza il SSR (con streaming) ci dà la possibilità di avere diversi vantaggi come: recupero dati dal server (miglior performance), sicurezza (token e chiavi salvate sul server senza esporle), caching (risultato del server può essere inserito in cache), dimensione bundle ridotti, primo contenuto disponibile più velocemente, ottimizzazione dell’indicizzazione dei motori di ricerca, streaming (caricamento dei blocchi di lavori e trasmessi in streaming al client, al loro completamento)

Next.js

Anche loro devono seguire lo stesso processo che abbiamo visto sopra per SSR. L'unica differenza è che non soffrono di un byte Time To First (TTFB) lento perché l'HTML viene generato in fase di compilazione oppure viene generato e memorizzato nella cache in modo incrementale man mano che arrivano le richieste.

Conclusione

Nelle app CSR, gli utenti devono scaricare tutto il JavaScript necessario ed eseguirlo per visualizzare/interagire con la pagina.

Con SSR risolviamo alcuni di questi problemi generando l'HTML sul server. Tuttavia non è ottimale poiché prima dobbiamo attendere che il server recuperi tutti i dati e generi l'HTML. Quindi il cliente deve scaricare JavaScript per l'intera pagina. Infine, poiché l'idratazione è un singolo passaggio in React, dobbiamo eseguire JavaScript per connettere l'HTML generato dal server e la logica JavaScript in modo che la pagina possa essere interattiva.

Quindi il problema principale è che dobbiamo aspettare il completamento di ogni passaggio prima di poter iniziare con quello successivo. Come abbiamo visto per risolvere questo problema si sta iniziando ad utilizzare lo **streaming SSR**.

Il rendering in streaming non è qualcosa di completamente nuovo in React 18. In effetti, esiste da React 16. React 16 aveva un metodo chiamato renderToNodeStream che, a differenza di renderToString, rendeva il frontend come un flusso HTTP al browser.

Utilizzando ora le Suspense molte cose che accadevano in serie ora accadono in parallelo.

Comparazione

Approach	Advantages	Disadvantages	Working	Uses
SSR	Fast initial load times, better SEO optimization	Increased server load, limited interactivity	The server renders HTML, sends to the client for display	Content-heavy websites, better SEO optimization
CSR	More interactive and dynamic web applications, smoother user experience	Slower initial load times, poor SEO optimization	times, poor SEO optimization The server sends the initial HTML, client updates with JavaScript	Web applications, SPAs
SSG	Fast initial load times and dynamic updates, better SEO optimization	Limited interactivity and dynamic updates, increased server load	The server generates static HTML, the client uses JavaScript to update	Websites that require both fast initial load times and dynamic updates, better SEO optimization

Fonte

Siti utilizzati e utili

- https://prateeksurana.me/blog/future-of-rendering-in-react/
- https://www.iamtk.co/the-evolution-of-react-rendering-architectures-and-web-performance
- https://www.geeksforgeeks.org/server-side-rendering-vs-client-side-rendering-vs-server-side-generation/
- https://www.searchenginejournal.com/client-side-vs-server-side/482574/
- https://www.callstack.com/blog/understanding-react-server-components