Gojob

React Server Components: La performance à quel prix ?

Certified B Corporation

Nassim Yagoub

Create React App est mort

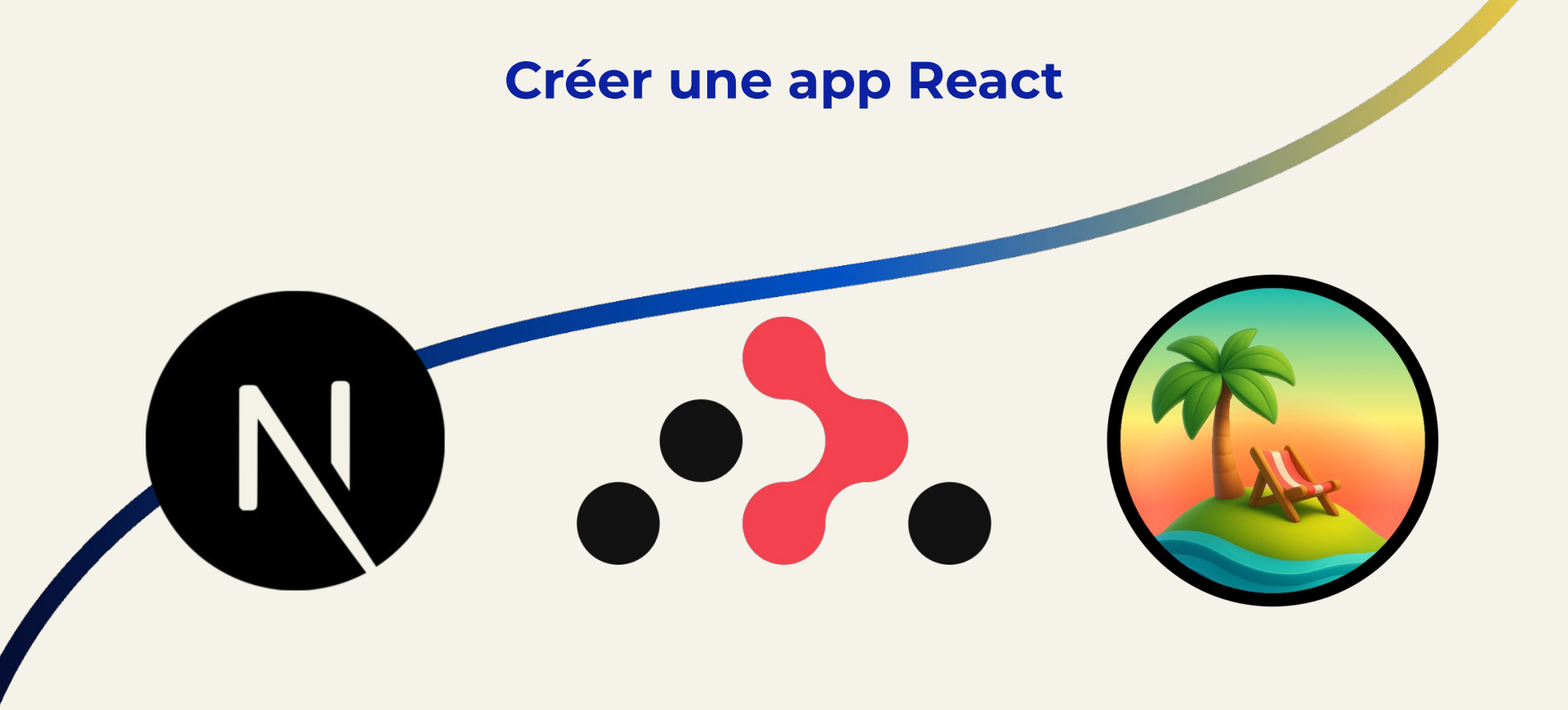
Sunsetting Create React App

February 14, 2025 by Matt Carroll and Ricky Hanlon

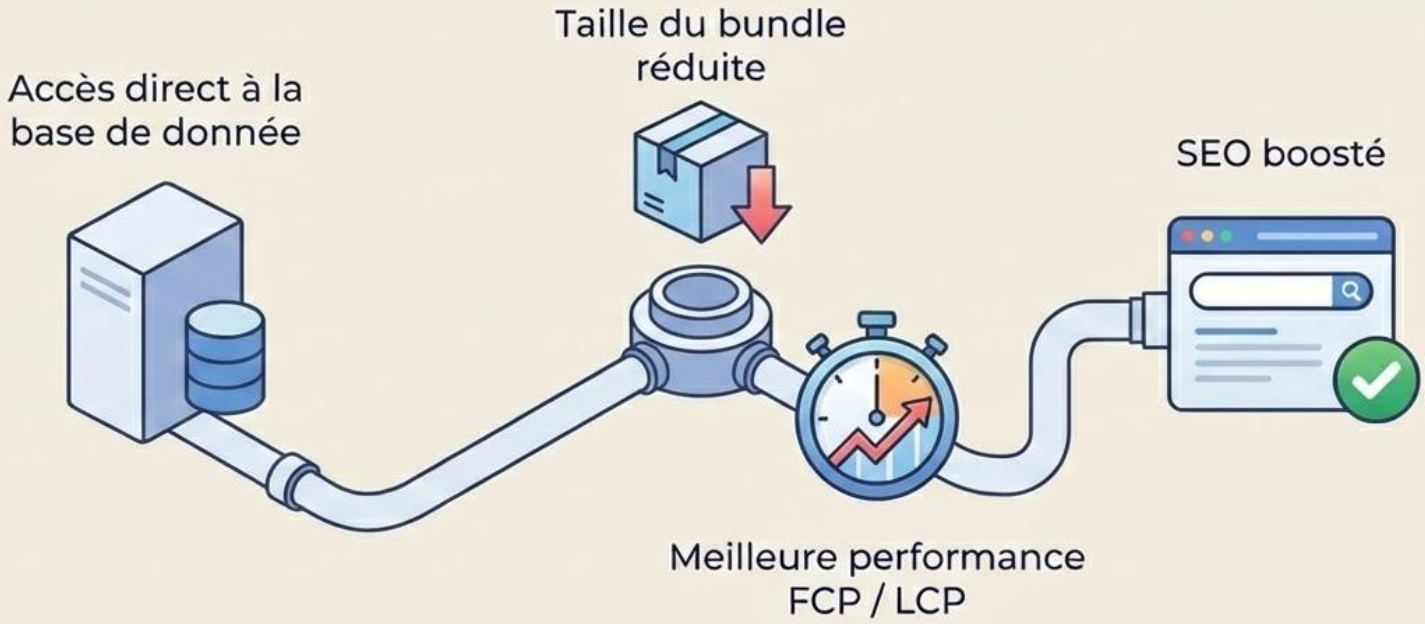
Today, we're deprecating Create React App for new apps, and encouraging existing apps to migrate to a framework, or to migrate to a build tool like Vite, Parcel, or RSBuild.

If you want to build a new app or website with React, we recommend starting with a framework.

Créer une app React



La promesse des RSC



Comment réaliser ça -> rendering hybride, sur le serveur quand c'est possible sinon sur le client

Déconstruire les RSC

Parlons performance

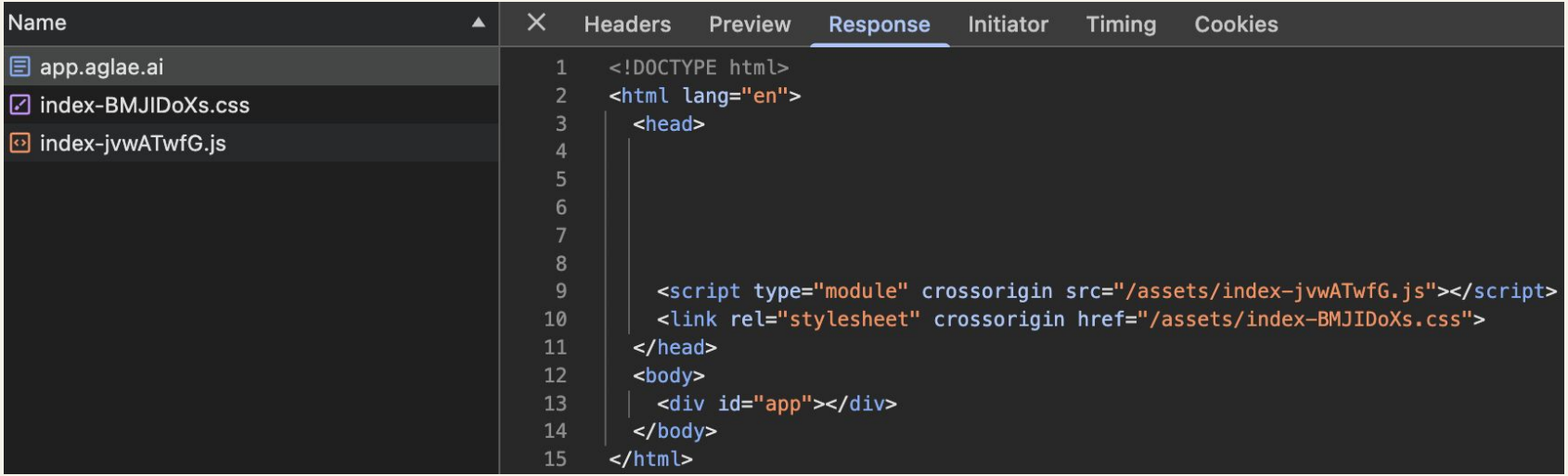
First Contentful Paint (FCP) : Le layout est apparu à l'écran, mais le contenu n'est pas forcément arrivé.

Time To Interactive (TTI) : React a pris la main, l'application a été hydratée, les éléments interactifs fonctionnent.

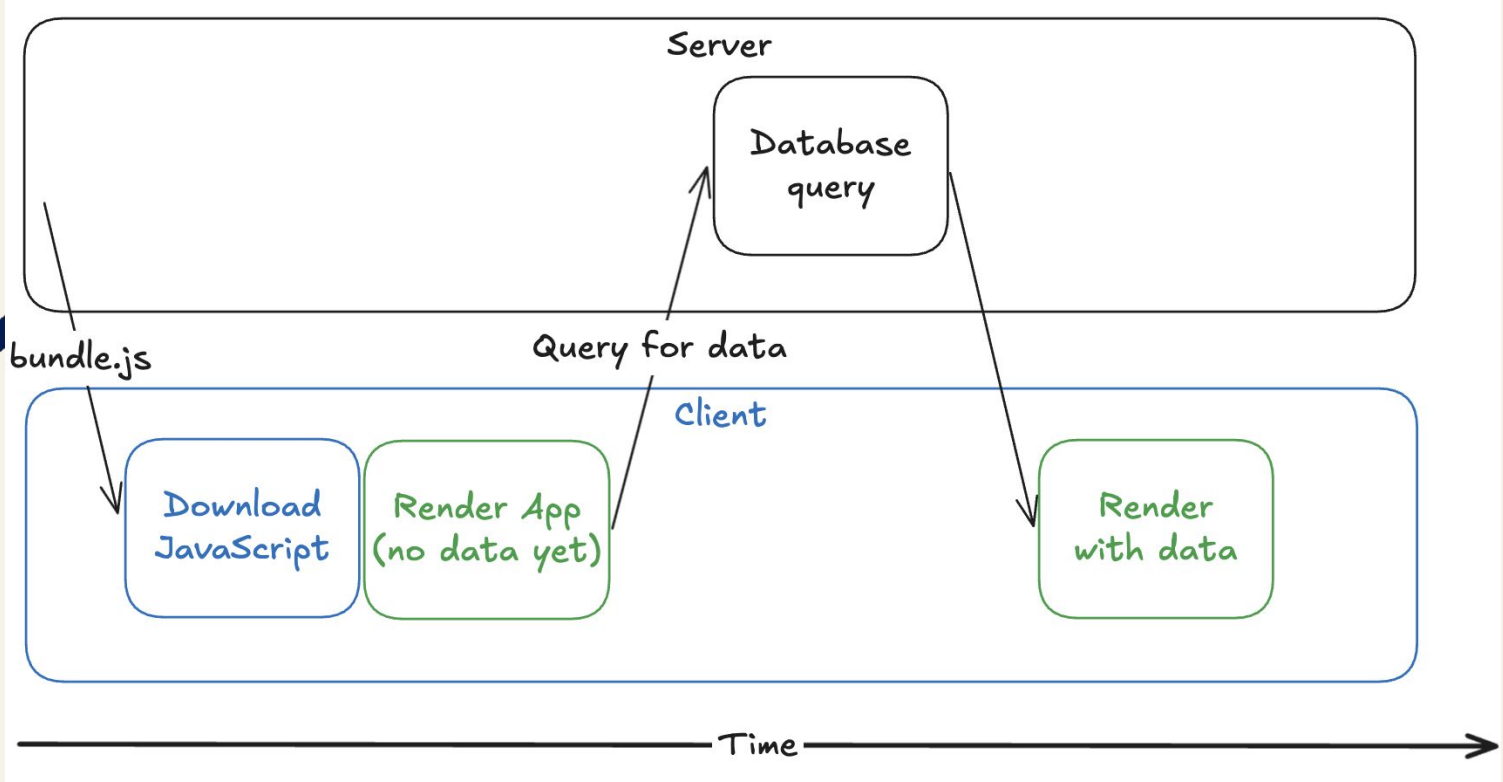
Largest Contentful Paint (LCP) : La donnée a été chargée et apparaît dans l'interface. L'application contient la donnée nécessaire à l'utilisateur.

Client Side Rendering

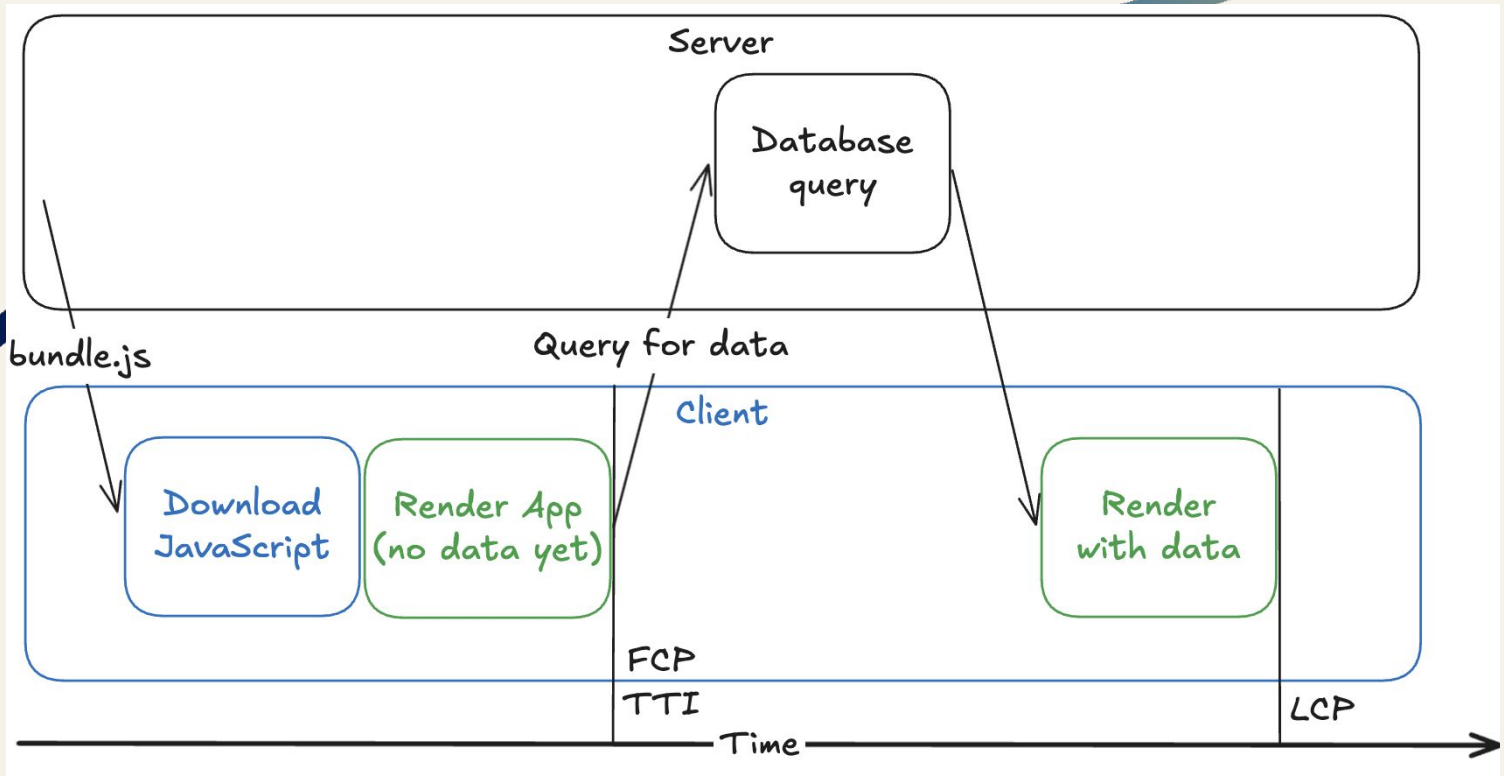
La stratégie classique des applications React, l'utilisateur se connecte sur une URL, reçoit un payload de ce type



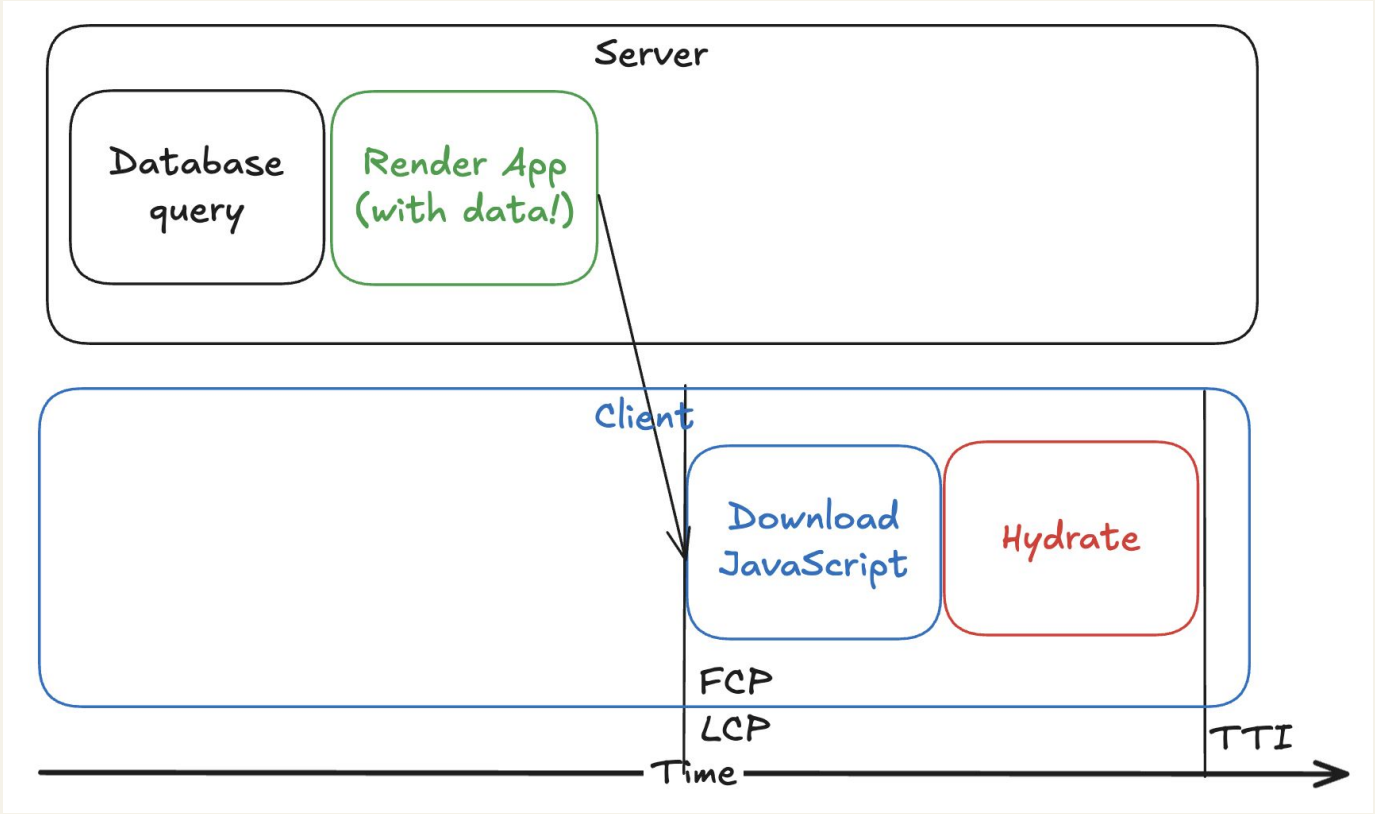
Client Side Rendering



Client Side Rendering



Server Side Rendering



Comparons le code

Client Component

```tsx
import { useQuery } from '@tanstack/react-query';

export default function ProductCard({ id }) {
  // 1. Needs an API layer
  const { data, isPending } = useQuery({
    queryKey: ['product', id],
    queryFn: () => fetch(`/api/products/${id}`),
  });

  // 2. Must handle loading state explicitly
  if (isPending) return <Skeleton />;

  return <div>{data.name}</div>;
}
```

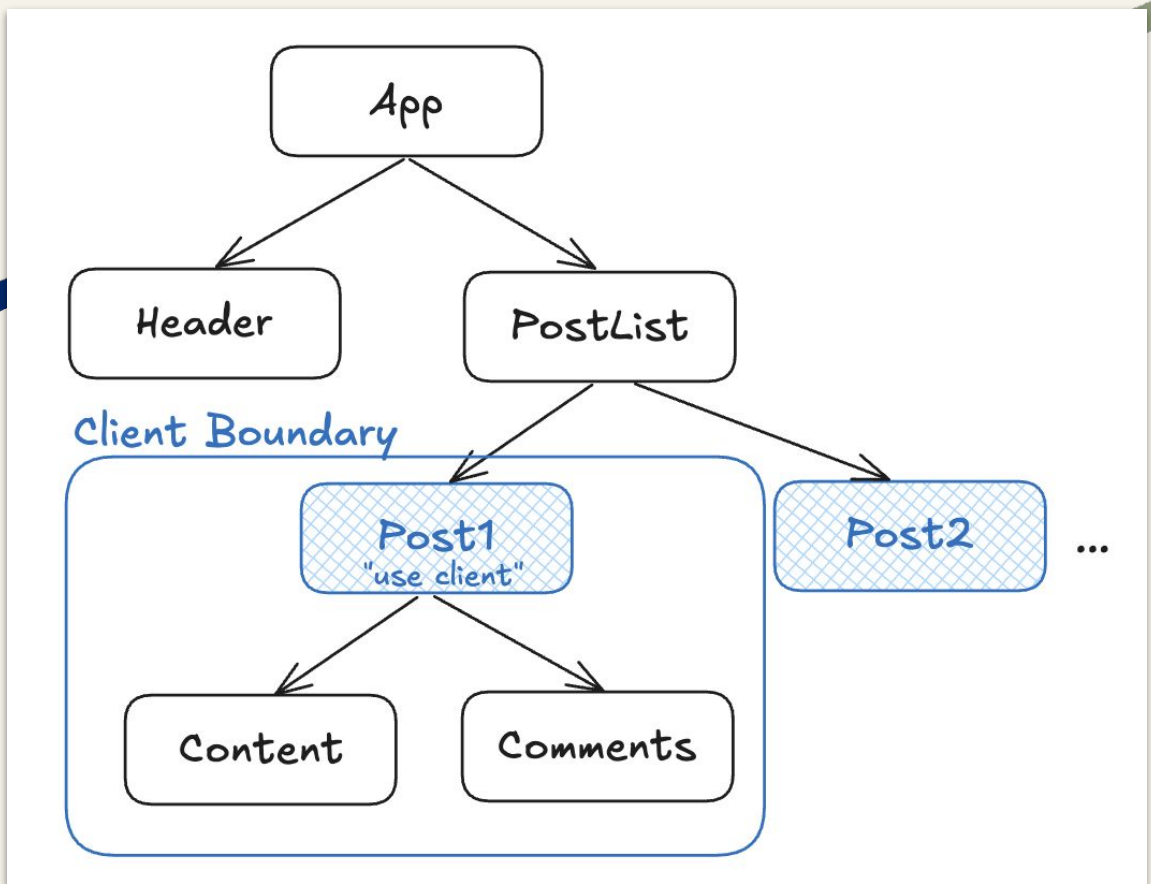
Server Component

```tsx
import { db } from "@/lib/db";

// 1. Component is async
export default async function ProductCard({ id }) {
  // 2. Direct Backend Access
  const product = await db.product.findUnique({
    where: { id },
  });

  // 3. Render immediately
  return <div>{product.name}</div>;
}
```

Différences de Rendering

	Server Component	Client Component
Rendu côté serveur	✅	✅
Rendu côté client	❌	✅
Embarqué dans le bundle JS	❌	✅

Frontières client



Conclusion

Amélioration des métriques de FCP et LCP grâce au modèle hybride client serveur.

L’envoi d’une page partiellement *rendered* améliore le SEO.

Coûts cachés

Framework obligatoire

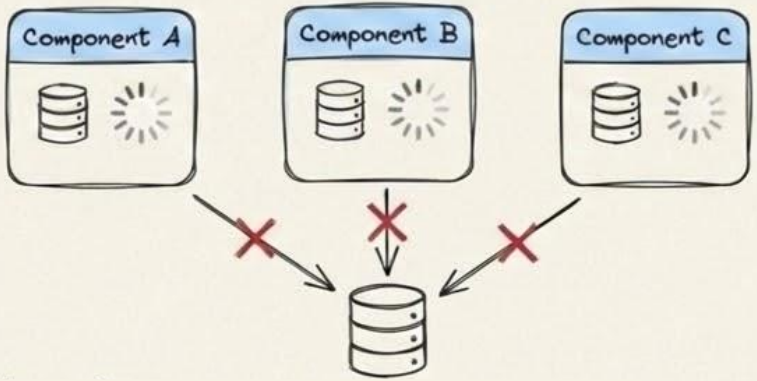
Contrairement à la plupart des features de React, il ne suffit pas de mettre React à jour pour en profiter.

Les server components changent la façon dont le code est bundled et exécuté, React met à disposition cette feature, mais il faut utiliser un router / framework les supportant !

- Next.js
- React Router

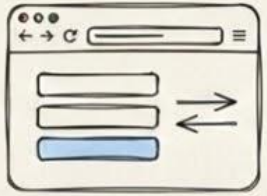
Un state fragmenté

La donnée de chaque Server Component est isolée



- Requêtes redondantes
- Pas de coordination entre composants
- Pas de loading global

Et on a toujours besoin de state client (formulaires, websockets, optimistic UI) !



Une requête à chaque navigation

Pas de cache côté client

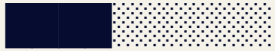
-> Chaque navigation mène à une requête réseau

Les visites suivantes

On gagne lors de la première visite, mais pas forcément par la suite, car le Client Side Rendering possède :

- Navigation instantanée
- Cache du navigateur pour les futures visites

Première visite:

RSC: 

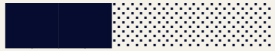
CSR: 

Durant une session:

RSC: 

CSR: 

Retour le lendemain:

RSC: 

CSR: 

Interactivité

Les RSC vivent dans un serveur et non un navigateur

- Pas de hook
- Pas d’api de navigateur
- Pas de réactivité

```
// UserProfile.tsx (Server Component)
async function UserProfile({ id }) {
  const user = await fetchUser(id)

  // Can't have interactivity, so just pass data down
  return <UserProfileClient user={user} />
}

// UserProfileClient.tsx (Client Component)
'use client'
function UserProfileClient({ user }) {
  const [isEditing, setIsEditing] = useState(false)
  // All the actual logic lives here
}
```

Le déploiement

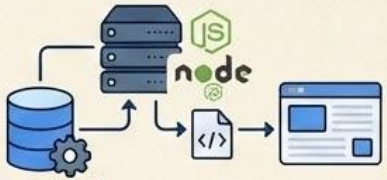
En CSR, on a simplement besoin d'héberger de assets statiques: le bundle de l'application

Quand on utilise les RSC, on a besoin d'un serveur node.js capable de générer les composants, le coût et la complexité augmentent

Déploiement CSR

- Hébergement simple:
- Assets statiques (bundle de l'application).
- Infrastructure minimale.

Déploiement RSC



- Serveur Node.js:
- Génération dynamique des composants.
- Coût et complexité accrus.

La charge mentale

Projet RSC

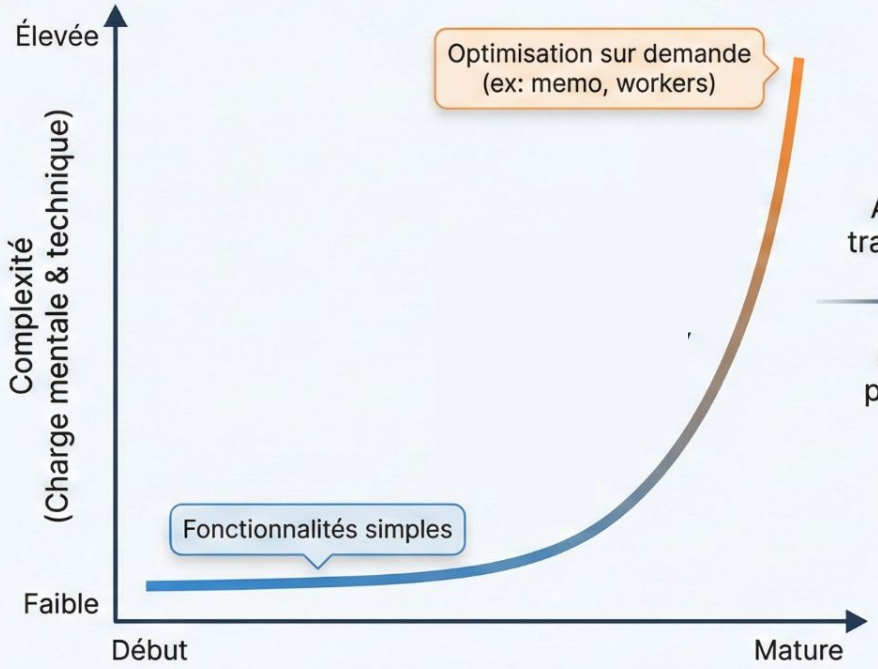
- Je suis sur le serveur ou le client ?
- Est-ce que j'ai accès à cette API / cette librairie ici ?
- Ca va casser si j'en fais un composant serveur ?
- Est-ce que j'envoie trop de données à travers la barrière client ?
- Comment je synchronise mes composants entre eux ?

Projet CSR

- Je suis dans le navigateur
- Est-ce que mon composant a trop de responsabilités ?
- Quelle est la queryKey pour cette donnée ?
- Est-ce que je veux prefetch la donnée sur cette route ?

La Courbe d'Optimisation Inversée

React Standard
(Optimisation sur demande)

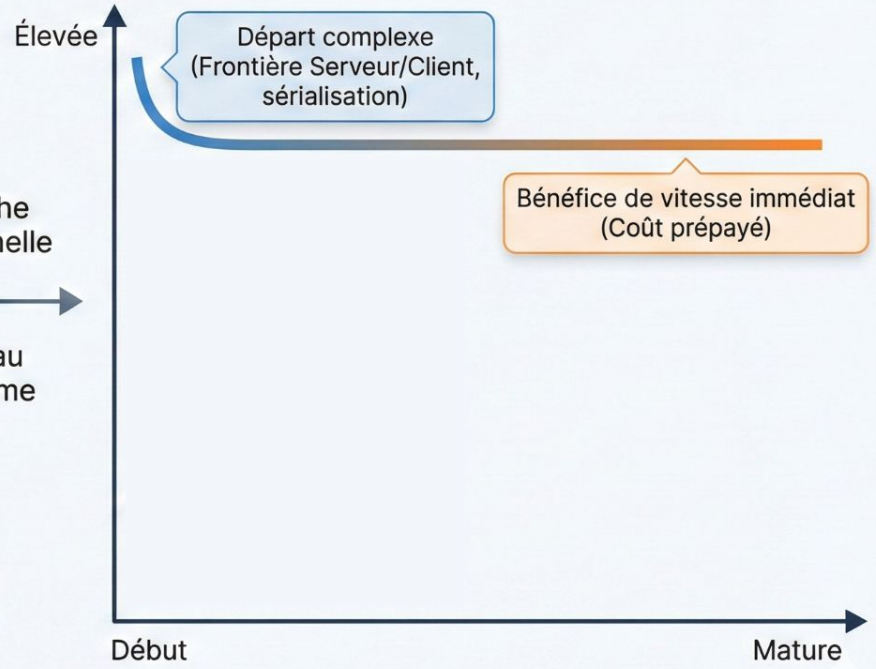


Approche traditionnelle

vs

Nouveau paradigme

Next.js RSC
(Performance par défaut)

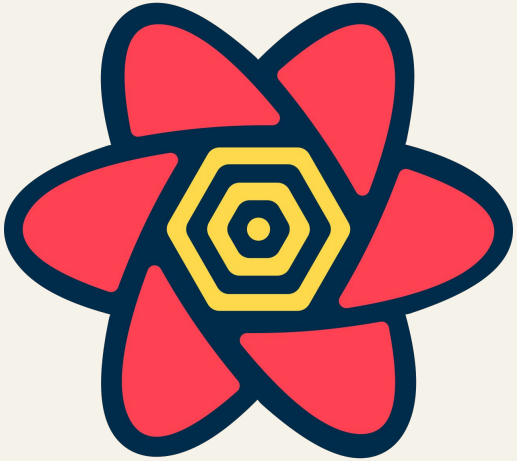


L’écosystème Tanstack

L’alternative: une architecture client-side moderne

Un client intelligent

Plutôt que de le vider, on va responsabiliser le client, car c’est lui qui a le plus de contexte sur une session utilisateur



Un cache unifié dans le navigateur

SWR et staleTime

- Navigation immédiate
- UI avec donnée stale
- Refetch si le staleTime est dépassé

Invalidation chirurgicale

- Pas d’overfetching
- Transitions douces

Seeding

- Initial data
- Placeholder data
- Récupération de données depuis d’autres queries

Routing type safe

```
<main>
  {posts.map((post) => (
    <Link key={post.id} to={""}>
```

```
.
..
/
/profile
/your-feed
/your-feed/$id
```

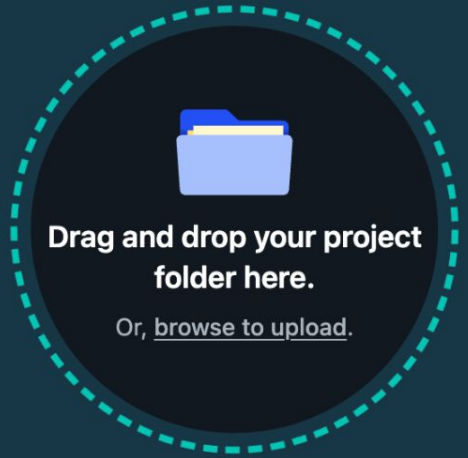
```
const ZodSchema = z.object({
  category: z.enum(['ALL', 'CATEG
});

export const Route = createFileRo
  validateSearch: ZodSchema,
  loaderDeps: ({ search }) => ({ search }),
  loader: ({ context: { queryClient }, deps: { search } }) => {
    queryClient.ensureQueryData(archivedCategoriesQueryOptions(search));
  },
  component: RouteComponent,
});
```

```
(property) search: {
    category: "ALL" | "CATEGORY_A" | "CATEGORY_B";
    fromDate?: string | undefined;
    toDate?: string | undefined;
    offerId?: string | undefined;
}
```

Prefetching

- Détection intelligente d’intention
- Prefetching chirurgical
- Navigations instantanées

Simplicité de déploiement



Drag & drop. It’s online.

Drop a folder with your project’s HTML, CSS, and JS files.
We’ll give you a link to share it.

Choisir son architecture

Opt-in vs opt-out

Les RSC échangent de la **performance** contre de la **complexité** !

C’est parfait en opt-in, à activer quand ça devient nécessaire.

L’App Router de next.js prend l’approche opposée et cette feature est en opt-out.

Matrice de décision

Application Type	SEO Priority	Interactivity	Auth Required?	Recommended Strategy
Marketing / Blog	Critique	Basse	Non	SSR
Dashboard / SaaS	Inutile	Haute	Oui	CSR
E-commerce	Critique	Moyenne	Mixte	SSR
Outil interne / Admin	Inutile	Haute	Oui	CSR

Verdict

Par défaut, je vous conseille Tanstack Router avec Tanstack Query

L’App router de Next.js est supérieur dans les cas suivants:

- Blog
- E-commerce
- Site marketing
- Landing page

Contact

https://www.linkedin.com/in/nassim-yagoub/