SSR vs SSG vs CSR

منظور از Rendering در Nextjs

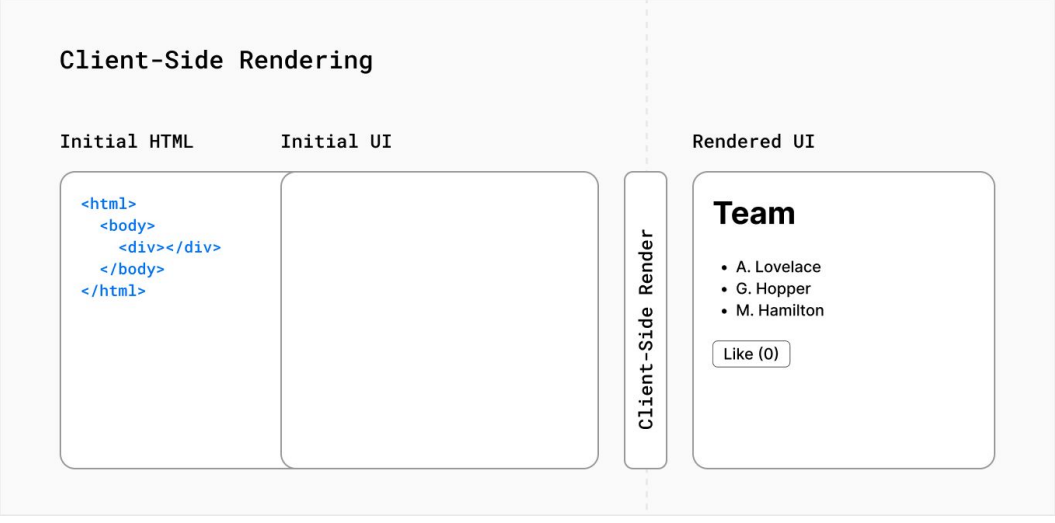
به عمل تبدیل کدهایی که (داخل پروژه React تون) مینویسید و ارائه اون بصورت رابط کاربری HTML رو Rendering میگن.

عملیات Rendering هم میتونه سمت Server مدیریت بشه هم سمت Client و همینطور میتونه فقط زمان build صفحه، عملیات Rendering انجام بشه یا به ازای هر بار Request یی که در زمان اجرا زده میشه . که بعدا فرق هاشون رو میگیم...

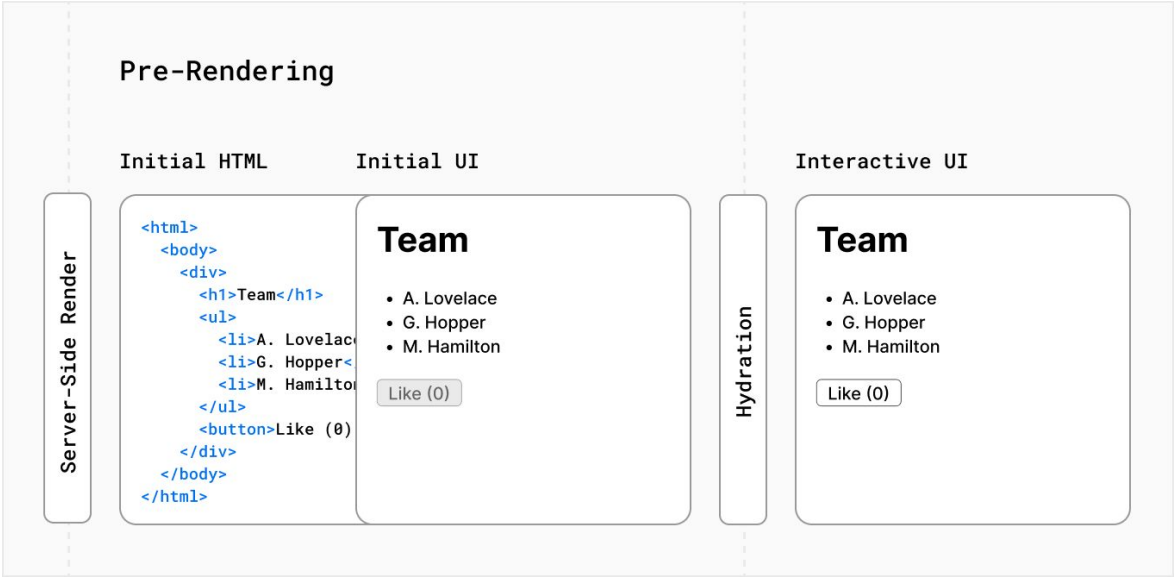
تفاوت Client-Side Rendering با Pre-Rendering

در حالت معمول در React ، مرورگر شما برای اینکه بیاد UI رو بسازه، میاد یه پوسته HTML خام به همراه جاوا اسکریپت های مورد نظر رو از سمت سرور میگیره بعد از اینکه جاوااسکریپت ها load شدن تازه اون موقع صفحه با محتواش بهتون نمایش داده میشه

همه چی سمت Client و مرورگر کاربر مدیریت میشه به این نوع Rendering میگن Client-side Rendering

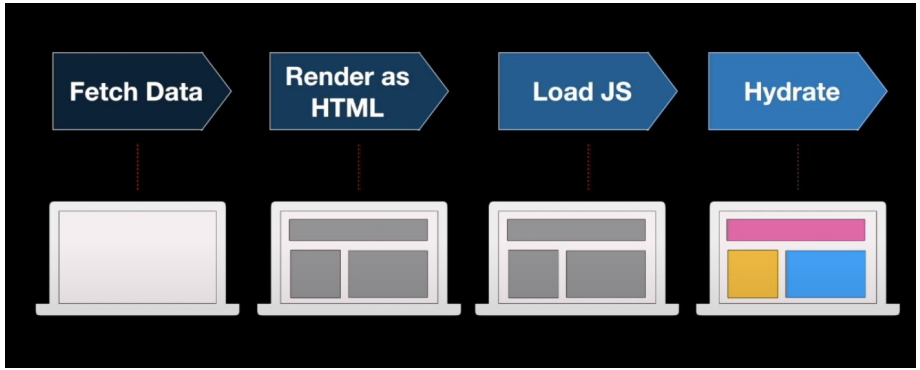


ولی در نقطه ی مقابل اون یعنی Pre-rendering ، به ازای هر Request ی که میزنیم HTML صفحه مورد نظر به همراه فایل های Json و کدهای جاوااسکریپت توی اون صفحه لازم داریم، داخل Server ساخته میشه و در نهایت نتیجه سمت Client ارسال میشه و شما توی مرورگرتون صفحه مورد نظرتون رو میبینین



Hydration

شما وقتی Request میزنید تا صفحه مورد نظرتون رو ببینید، UI اولیه صفحه سمت Server ساخته میشه و به شما نمایش داده میشه (بدون جاوااسکریپت) که در اصطلاح میگن non-interactive ، و در نهایت بعد از اینکه جاوااسکریپت های موردنظر دانلود شدن نتیجه کامل تر رو مشاهده می کنید.



در اصل فرقش با Client-side Rendering توی همینه که شما در حالت قبلی تا زمانی که کل فایل ها چه HTML و جاوا اسکریپت و... کامل load نشن ، شما با صفحه سفید مواجه هستید و وقتی تمام فایل ها load شدن اون موقع می توانید صفحه کامل رو ببینین .

حالا یه حرکت جالبی که Pre-Rendering داره اینه که ، میاد سریع یه UI اولیه از صفحه موردنظرتون رو بهتون نمایش میده تا شما نمایش اولیه صفحه رو داشته باشین در این حین میاد همزمان اون فایل های جاوااسکریپت و json و... که توی صفحه نیاز دارید رو بارگذاری میکنه به این حرکتش میگن Hydration

درباره CSR

خب Client-side Rendering یا به اختصار CSR به عملیات دریافت داده (Fetching data) در سمت Client گفته میشه

حالا آیا اصلا این متد بدرد بخور هست؟

باید بگم برای زمانی که صفحه تون نیازی به SEO نداره که گوگل بخواد اون رو index کنه و یا صفحه تون نیاز به Pre-Render نداشته باشه و data هاش هر لحظه بروز نمیشن ، این متد میتونه گزینه مناسبی برای این شرایطی که خدمتتون عرض کردم باشه

فقط یه نکته ای که خیلی حواستون باشه اینه که این روش تاثیر زیادی میتونه روی Performance داشته باشه بخصوص روی سرعت بارگزاری صفحه تون هست چون همه چی (Fetching data) سمت Client مدیریت میشه.

روش های استفاده از CSR

```jsx
import { useState, useEffect } from 'react'

function Profile() {
  const [data, setData] = useState(null)
  const [isLoading, setLoading] = useState(false)

  useEffect(() => {
    setLoading(true)
    fetch('/api/profile-data')
      .then((res) => res.json())
      .then((data) => {
        setData(data)
        setLoading(false)
      })
  }, [])

  if (isLoading) return <p>Loading...</p>
  if (!data) return <p>No profile data</p>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

```jsx
import useSWR from 'swr'

const fetcher = (...args) => fetch(...args).then((res)
=> res.json())

function Profile() {
  const { data, error } = useSWR('/api/profile-data',
fetcher)

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

درباره SSR

وقتی میخواید صفحه مورد نظرتون SEO داشته باشه و گوگل اون رو index کنه باید از این متد استفاده کنید

شما وقتی میخواهید صفحه تون SSR بشه نیاز به یک تابع دارید به نام getServerSideProps وقتی که از این تابع توی صفحه مون ما export بگیریم ، Nextjs اون صفحه رو SSR در نظر میگیره به ازای هر Request ی که زده میشه این صفحه رو برامون Pre-Render میکنه.

به طور کلی این تابع در سمت server اجرا میشه و هرگز روی مرورگر کاربر اجرا نخواهد شد و وقتی که شما صفحه SSR رو از طریق next/link و یا next/router از سمت Client درخواست (Request) میدید، Nextjs هم میاد درخواست شما رو در قالب یک Api Request میفرسته سمت Server که شامل getServerSideProps هست

```
function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Pass data to the page via props
  return { props: { data } }
}

export default Page
```

یه نکته جالب دیگه ای که لازمه بدونید اینه که Nextjs عملیات Client-bundle رو هم انجام میده به این معنی که میاد اون کدهایی که لازم نیست در سمت client باشن و bundle بشن رو حذف میکنه مثل همین تابع getServerSideProps.

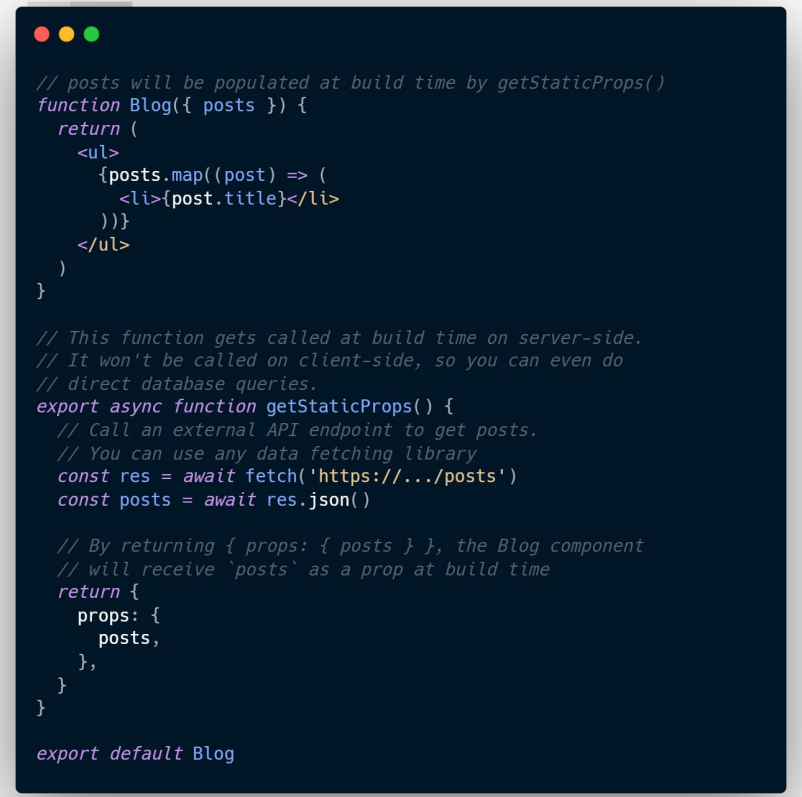
مبحث جالب دیگه از SSR هست که در واقع خاصیت Caching اون هست که میتونیم با (Cache-Control) توی header مون تعریفش کنیم و حتی با استفاده از stale-while-revalidate. میتونیم برای Revalidate اون، زمان بندی تعریف کنیم.

```
// This value is considered fresh for ten seconds (s-maxage=10).
// If a request is repeated within the next 10 seconds, the previously
// cached value will still be fresh. If the request is repeated before 59 seconds,
// the cached value will be stale but still render (stale-while-revalidate=59).
//
// In the background, a revalidation request will be made to populate the cache
// with a fresh value. If you refresh the page, you will see the new value.
export async function getServerSideProps({ req, res }) {
  res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalidate=59'
  )

  return {
    props: {},
  }
}
```

میاد داده ها رو در حافظه پنهان نگه داری میکنه و طبق اون تنظیماتی که ما براش مشخص میکنیم داده ها رو از cache گرفته و دوباره میاد و داده ها رو یا revalidate اعتبار سنجی مجدد میکنه

درباره SSG



```
// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
// It won't be called on client-side, so you can even do
// direct database queries.
export async function getStaticProps() {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}

export default Blog
```

برای زمانی که میخواهید از Static Site Generation یا همون SSG استفاده کنید، کافیه تابع getStaticProps رو توی صفحه تون export کنید و وقتی Nextjs بلافاصله ببینه تو صفحه تون از این تابع استفاده کردین فقط یکبار موقع build پروژه اون رو Pre-render میکنه برامون

(برعکس SSR که به ازای هر Request میومد برامون Pre-render رو انجام میداد.)

فرق SSG با SSR

ما صفحه ای رو SSR میکردیم که اطلاعاتش همش در حال بروز شدن بود و می خواستیم کاربر هر بار Request میزنه ، صفحه مون هم Pre-Render بشه و هم SEO گوگل اون هم حفظ بشه

ولی توی SSG میگیم ما نمیخوایم صفحه مون به ازای هر بار Request کاربر بخواد Pre-render انجام بده چون data مون هربار تغییری نمیکنه و یکبار هم Pre-render انجام بشه برامون کافیه و همین هم باعث میشه صفحه مون با سرعت بالاتری نمایش داده بشه که اگه گوگل ببینه سرعت صفحه بالاست ، index بهتری بهش میده.

حالا اون یکبار Pre-render کجاست؟ اون زمانی که پروژه رو build میگیری .

یه موضوع دیگه ای که Nextjs در خصوص SSG بهش اشاره کرده اینه که میگه getStaticProps چون HTML که درست میکنه به صورت static هست به Request های ورودی مثل query parameters و HTTP headers دسترسی نداره و برای اینکه دسترسی داشته باشیم باید از Middleware استفاده کنیم

```jsx
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// the path has not been generated.
export async function getStaticPaths() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at build time.
  // { fallback: blocking } will server-render pages
  // on-demand if the path doesn't exist.
  return { paths, fallback: 'blocking' }
}

export default Blog
```

درباره ISR
(Incremental Static Regeneration)

شبیه SSG هست با این تفاوت که برای build کردن کافیه بهش بگی مثلا بعد از ده ثانیه دوباره از این صفحه build بگیر

ISR در واقع ترکیبی از SSR و SSG هست

نگاهی عمیق تر به Hydration

Hydration کاری هست که توی مرورگر انجام میشه تا صفحه ای که سمت سرور رندر شده رو به وضعیتی برسونه که انگار توی کلاینت رندر شده. این شامل دانلود و اجرا جاوا اسکریپت مربوط به React و همه کامپوننت ها، پیدا کردن المنت های DOM ای که باید event handler بهشون وصل بشه، وصل کردن event handler، مقدار دهی اولیه state و ... میشه.

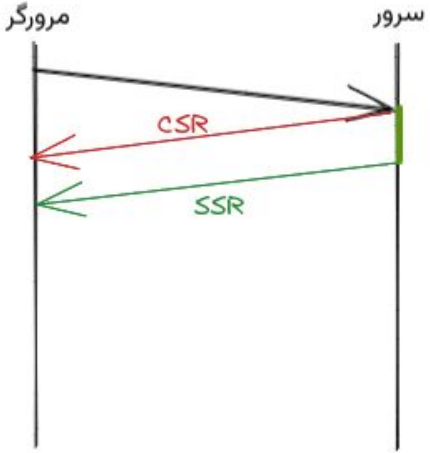
و این **یعنی دوباره کاری!!**

این کار ها قبلا توی سرور انجام شده بودند ولی فریم ورک عزیز اونا رو فراموش کرده و فقط HTML رو بهمون داده. البته تا حدی هم منطقیه چون وصل کردن event handler ها توی سرور حتی اگر ممکن باشد بعید میدونم به درد کسی بخوره.

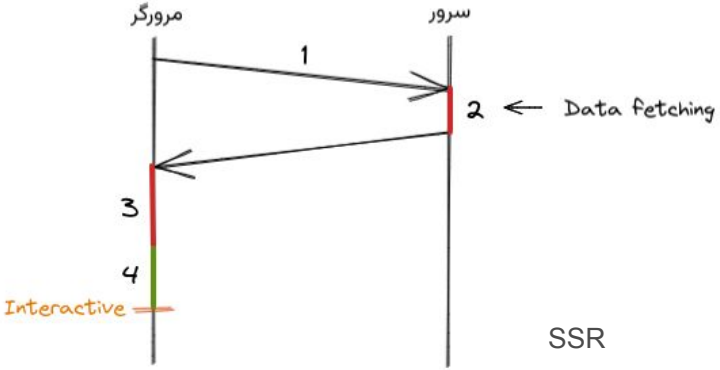
دانلود و اجرا اون همه جاوااسکریپت زمان، پهنای باند و cpu میگیره که چیز خوبی نیست. وقتی یه صفحه سمت سرور رندر میشه ما HTML کامل رو تحویل مرورگر میدیم و این یعنی کاربر در کمترین زمان ممکن به کل محتوا صفحه دسترسی داره ولی تا وقتی hydration کامل نشه نمیتونه کار مفیدی انجام بشه. پس اگر hydration بیش از حد طور بکشه تجربه خوبی رو به همراه نداره.

شاید بگید خب توی CSR هم همون جاوااسکریپت ها دانلود و اجرا میشن و این قضیه اونقدر هم بد نیست. توی CSR حداقل این کار دو بار انجام نمیشه. وقتی ریکوئست میفرستیم، سرور بلافاصله یه HTML خالی بهمون تحویل میده و فورا دانلود و اجرا جاوا اسکریپت ها انجام میشه.

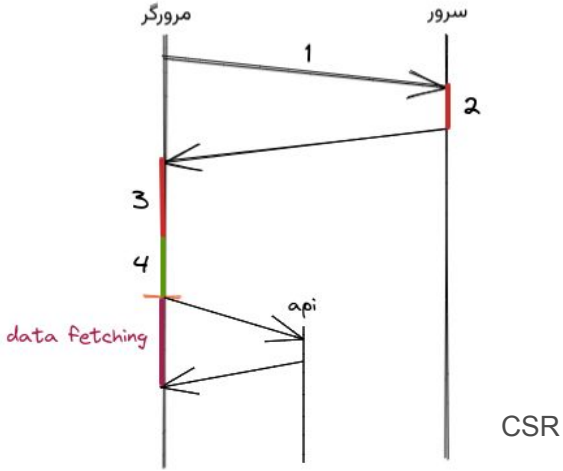
ولی توی SSR سرور همین الان هم سر data fetching و رندر کردن HTML یکم وقت تلف کرده و اینکه بخواد اون جاوااسکریپت هارو هم دانلود و اجرا کنه زیاد جالب نیست.



تفاوت CSR و SSR در زمان ارسال HTML



SSR



CSR

پس چرا هنوز از SSR استفاده میکنیم؟

خیلی وقت ها **SSR** بهتر از **CSR** جواب میده چون یک سری کار هارو زودتر شروع میکنه. مثل **data fetching**.

وقتی از **SSR** استفاده میکنیم، به محض اینکه ریکوئست رسید کار **data fetching** رو انجام میدیم. ولی وقتی از **Client Side Rendering** استفاده میکنیم، اول کل جاوااسکریپت باید دانلود و اجرا بشه، بعد **HTML** رندر بشه، بعد **asset** هایی که نیاز داره مثل عکس و **CSS** رو دانلود کنه (عکس، **css** و **inline javascript** ها همه **blocking** هستند و تا وقتی دانلود و اجرا نشدن بخش **async** جاوااسکریپت که احتمالا **data fetching** داره اجرا نخواهد شد) و وقتی اینها کامل شد تازه اون موقع میتونیم اطلاعات مورد نیاز رو از یه **api** بگیریم.

سرور وقتی با درخواست csr و ssr و ssg مواجه میشه، چیکار می‌کنه؟

توی csr

1. درخواست فرستاده میشه به سرور
2. سرور فقط اطلاعات مورد نیاز صفحه رو میگیره از دیتابیس و می‌فرسته به کلاینت
3. کلاینت صفحه رو می‌سازه.
4. کاربر صفحه رو می‌بینه

توی ssg

1. کاربر یه درخواست میزنه به سرور
2. سرور صفحه ای کاربر میخواد رو زمان آپلود پروژه روی هاست ساخته و داره. همونو میفرسته واسه کاربر
3. کاربر صفحه رو می‌بینه

توی ssr

1. کاربر درخواست می‌زنه به سرور
2. سرور اطلاعات جدید رو از دیتابیس میگیره و صفحه جدید رو همون لحظه میسازه
3. صفحه فرستاده میشه به کلاینت.
4. کاربر صفحه رو میبینه.

CSR

Strong Sides:

- **Fast on the server:** Because you are only rendering a blank page it’s very fast to render.

CSR

Weak Sides:

- **No initial render:** You are sending a blank page to the customer. So if your app is big, or the customer is on a slow connection, that's going to be less than ideal.
- **Security:** Compare to the traditional page ,Single Page Application is less secure due to Cross-site scripting (XSS).
- **Memory Leak:** Memory leak in JavaScript can even cause a powerful system to slow down.
- **Being empty <body>** means there will be not content to crawl the data for a search engine, so SEO is the biggest weak point here especially if you want many users to reach your web app
-

SSR

Strong Sides:

- **Content immediately available:** Because you are sending rendered HTML to the client who will start to see content almost immediately. Content is up to date because it fetches content on the go;
- **No additional client fetches:** Ideally the server rendering process will make all the required calls to get the data, so you won't be making any additional service calls from the client. At least until the user starts to play around with the page a little. your users' devices have little relevance to the load time of your page.
- **Great for SEO:** Search engines like HTML content. If you are using client-side rendering only then you are counting on search engines running your Javascript, which may or may not happen, but here crawlers of search engines will fetch your content faster and easier.

SSR

Weak Sides:

- **Slower on the server:** You are rendering the page twice, once on the server, and once on the client. You are also probably making service calls from the server to render the page. All of that takes time so the initial send of the HTML to the client could be delayed.
- **Incompatible with some UI libraries**: If the UI library uses window or document objects then you are going to need to fix that or use something else because Node doesn't have window or document objects, example when you use window object to know the size of the screen of the user to render specific components, then you should use some solutions which depend on the type of framework you use with SSR.
-

SSG

Strong Sides:

- **Fast website:** Since all of your site's pages and content have been generated at build time, the client will start to see content almost immediately. Plus, you do not have to worry about API calls to the server for content and this makes your site very fast.
- **No additional client fetches**: Ideally the server rendering process will make all the required calls to get the data, so you won't be making any additional service calls from the client. At least until the user starts to play around with the page a little.
- **Great for SEO**: Search engines like HTML content.
- **Security:** Statically generated site is solely composed of static files, the risk of being vulnerable to cyber attacks is minimal. This is because static generated sites have no database, attackers cannot inject malicious code or exploit your database.
- **No Server:** You don't have to run a server. So you don't need to monitor that server and you are going to get far less pager duty pings.

SSG

Weak Sides:

- **Can be slow to rebuild large sites:** · Build time would increase depending on the size of the application.
- **Incompatible with some UI libraries**: If the UI library you like uses window object then you are going to need to fix that or use something else because Node doesn't have `window` or `document ob`.
- **Content can become stale** if it changes too quickly, to update its content, you have to rebuild the site.

SSG • SSR • CSR • ISR

Static Site Generator	Server-side Rendering	Client-side Rendering	Incremental Static Regeneration	What
Build Time	Request Time	Request Time	Build Time & Request Time	When
Server	Server	Client	Server	Where
getStaticProps & getStaticPaths	getServerSideProps	client side fetch	getStaticProps with {revalidate}	How
SEO, cache	SEO, dynamic data	smooth transition between pages, dynamic data	SEO, cache, dynamic data	Why

جمع بندی

ممکنه یه صفحه مون اصلا به SEO گوگل احتیاجی نداشته باشه میایم از CSR استفاده میکنیم ، ممکنه صفحه داشته باشیم که هر لحظه data ش update میشه مثل لیست قیمت محصولات فروشگاه و میخوایم گوگل اون رو index کنه ، پس بهتره از SSR استفاده بشه . و یا مثلا صفحه وبلاگی داریم میخوایم گوگل اونو index کنه ولی اطلاعاتش دیر به دیر update میشه، خب میایم از SSG استفاده میکنیم و یا میخوایم ترکیبی عمل کنیم یعنی صفحه مون میخوایم SSG باشه ولی بعد از یه مدت خاص Pre-render بشه اونجا هم میایم از ISR استفاده میکنیم و تمام.