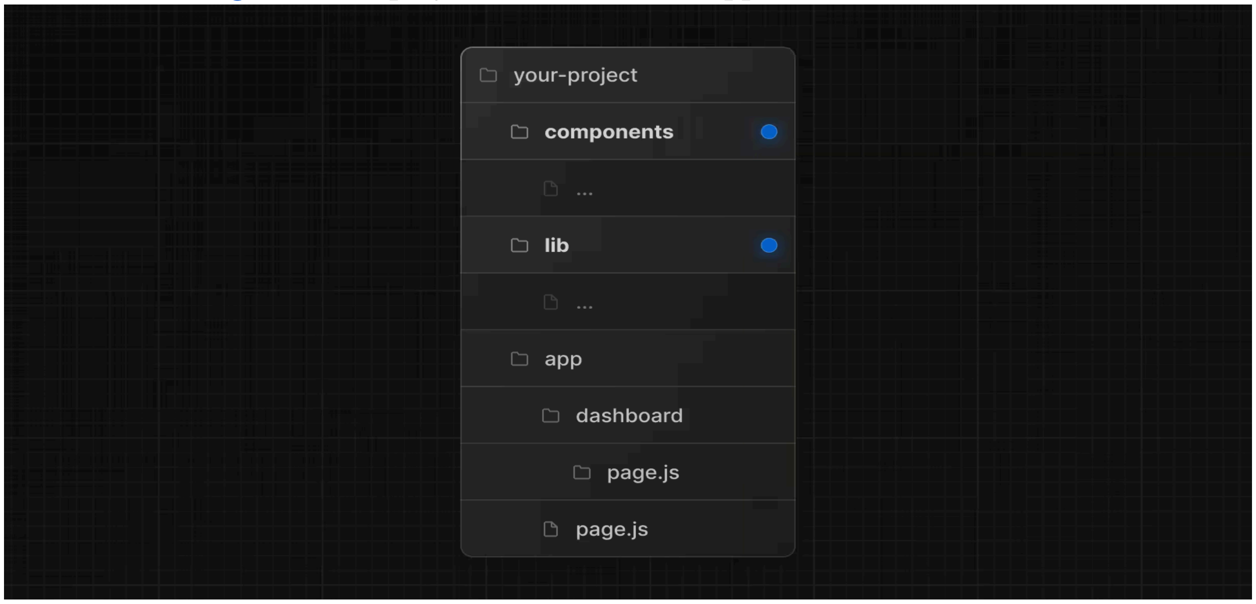
Next Js Basic

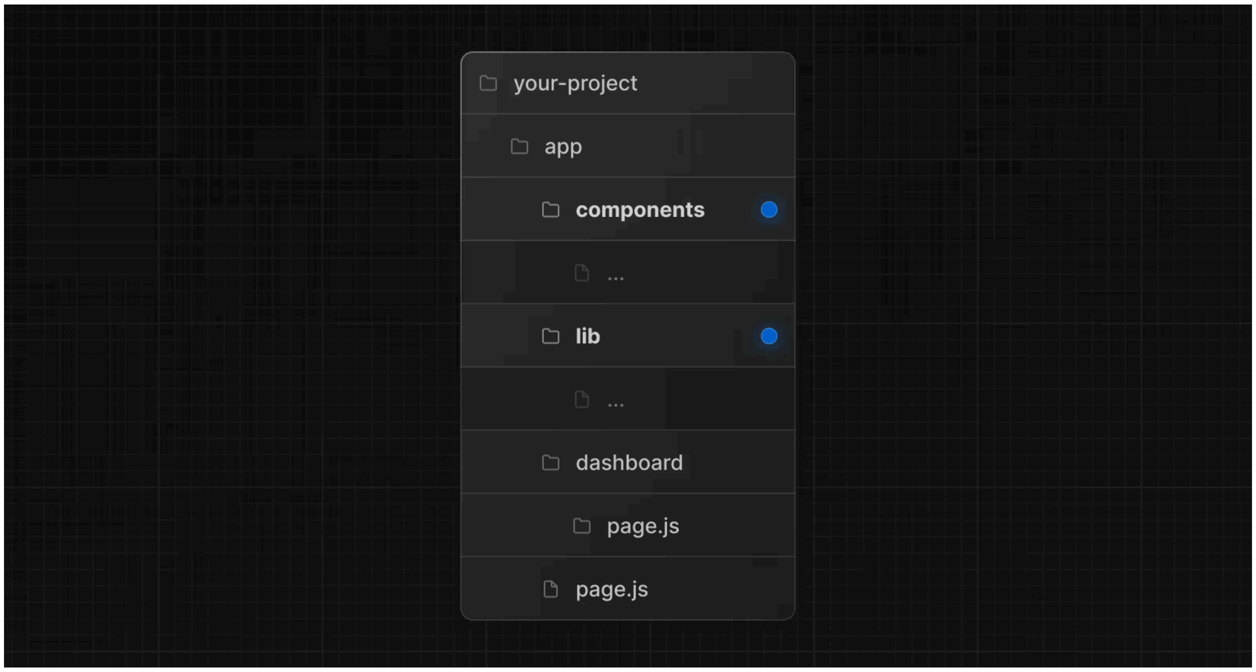
Next.js is unopinionated about how you organize and colocate your project files.

Folder Structure

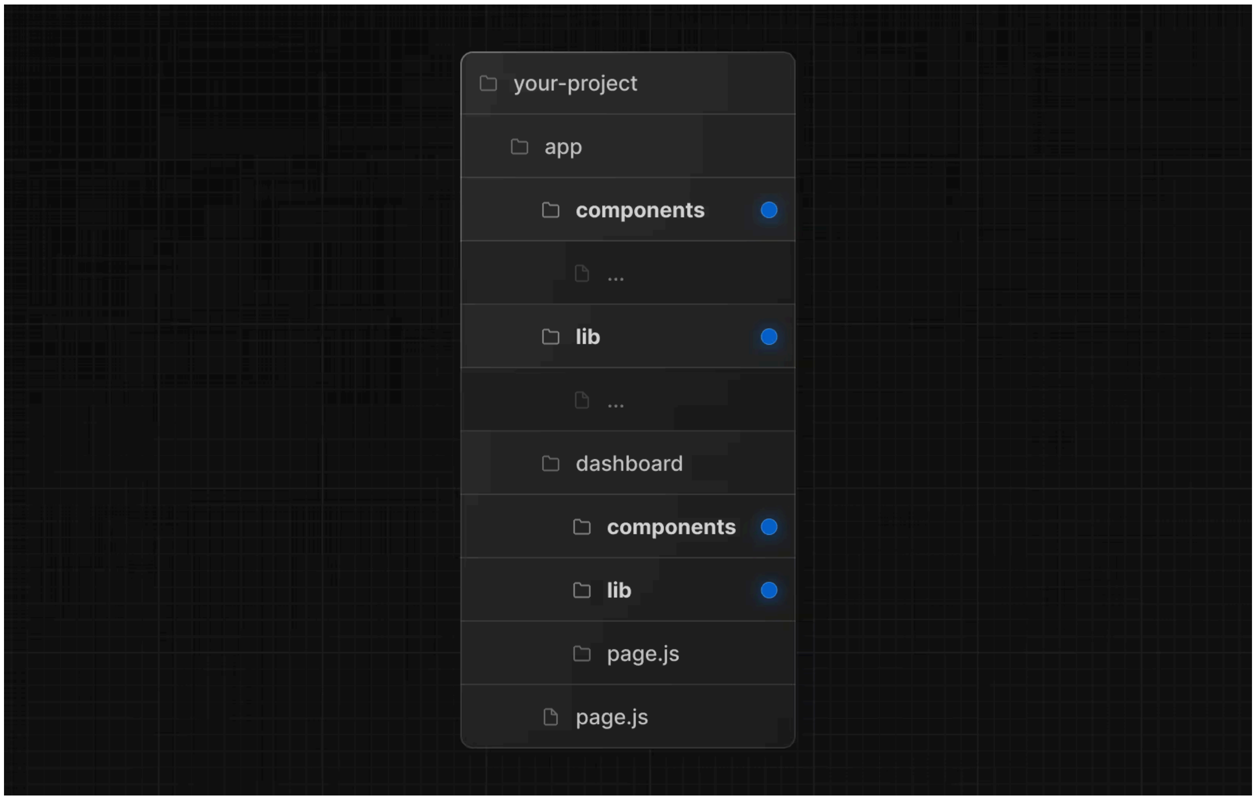
Common strategies **(**Store project files outside of app**)**



Common strategies (Store project files in top-level folders inside of app)



Common strategies **(**Split project files by feature or route**)**



Server-Side Rendering (SSR)

One of Next.js' standout features is SSR, which allows rendering React components on the server instead of the client. This results in faster initial page loads and better SEO since search engines can crawl the fully rendered HTML.

If a page uses Server-side Rendering, the page HTML is generated on each request. To use Server-side Rendering for a page, you need to export an async function called getServerSideProps. This function will be called by the server on every request.

For example, suppose that your page needs to pre-render frequently updated data (fetched from an external API). You can write getServerSideProps which fetches this data and passes it to Page like below:

```
export default 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 } }
}
```

As you can see, getServerSideProps is similar to getStaticProps, but the difference is that getServerSideProps is run on every request instead of on build time.

Definition

- SSR means generating the page on the server for every request.
- The server fetches data, renders the page, and sends the fully built HTML to the client.
- Improves SEO and ensures fresh data for each request.

How It Works

1. A request comes from the client.
2. The server fetches data and generates HTML dynamically.
3. The HTML is sent to the browser and displayed immediately.
4. React hydrates the page for interactivity.

When to Use SSR?

- When you need fresh data on every request (e.g., news sites, real-time dashboards).
- When SEO is crucial and the data changes frequently.
- User-specific content (e.g., profile pages)
- News websites

Avoid SSR if:

- You don’t need real-time data (SSG is faster).
- Your site relies on a CDN for performance.

Static Site Generation (SSG)

Next.js offers SSG, which pre-renders pages at build time. This approach generates static HTML files for each page, resulting in incredibly fast page loads and enabling easy deployment to content delivery networks (CDNs).

Definition

- SSG means generating the page at build time, not on every request.
- The page is pre-rendered and stored as static HTML.
- It’s the fastest rendering method because no backend processing is needed at runtime.

How It Works

1. At build time, Next.js fetches data and generates static HTML.
2. The HTML is stored on a CDN.
3. When a user requests the page, the static file is served instantly.

When to Use SSG?

- For pages with data that doesn’t change often (e.g., blogs, documentation, marketing sites).

- When performance is a priority, as static files load super fast.
- Best for pages that can be pre-rendered ahead of time, such as:
    - ★ Marketing pages
    - ★ Blog posts & portfolios
    - ★ E-commerce product listings
    - ★ Help & documentation pages

```
export async function getStaticProps() {

  const res = await fetch('https://api.example.com/data');

  const data = await res.json();

  return { props: { data } };

}

export default function Page({ data }) {

  return <div>{data.title}</div>;

}
```

The page is generated once at build time and served as static content.

Incremental Static Regeneration (ISR) - Extending SSG

- ISR allows updating static pages **without rebuilding the entire site**.
- Pages are regenerated in the background after a specific time.
- Example:

```
export async function getStaticProps() {


  const res = await fetch('https://api.example.com/data');
```

```
 const data = await res.json();



 return { props: { data }, revalidate: 60 }; // Regenerate every 60 seconds


}
```

Client-Side Rendering (CSR) - For Dynamic UI Components

Next.js provides a built-in client-side routing system. You can create dynamic, single-page applications (SPAs) without the need for complex routing configurations.

Definition

- CSR means the page loads with minimal HTML, and React fetches data **in the browser** after the page loads.
- Good for **interactive, user-driven applications** but **not SEO-friendly**.

When to Use CSR?

✅ **Best for pages that require heavy user interaction**, such as:

- Admin dashboards
- Search results pages
- Real-time applications (chat apps, stock price tracking)

🚫 **Avoid CSR if**:

- SEO is critical (SSR or SSG is better).
- You need a fast initial load time.

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

export default function Page() {

 const [data, setData] = useState(null);

 useEffect(() => {

  fetch('https://api.example.com/data')

   .then((res) => res.json())

   .then((data) => setData(data));

 }, []);

 return <div>{data ? data.title : 'Loading...'}</div>;

}
```

Comparison Table

Feature	SSG	ISR (SSG + Updates)	SSR	CSR
When data is fetched	At build time	At build time (updates later)	On each request	On the client
Speed	⚡ Fastest (cached by CDN)	⚡ Fast (with updates)	🛑 Slower (server request)	🟡 Slowest (loads after page)
SEO	✅ Yes	✅ Yes	✅ Yes	❌ No

Use case	Blogs, docs, marketing pages	Product pages, news	Real-time data, user profiles	Dashboards, interactive apps

Automatic Code Splitting

Next.js automatically splits your JavaScript code into smaller chunks, ensuring that users only download the code necessary for the current page. This optimizes load times and performance.

File-Based Routing

Next.js simplifies routing by allowing developers to create pages using a file-based system. Simply create a JavaScript file in the “pages” directory, and Next.js handles the routing for you.

Caching and revalidation

In Next.js, **caching and revalidation** determine how frequently data is refreshed and whether a new request is needed. These strategies optimize **performance, load times, and freshness** of data.

1️⃣No Cache (Always Fresh Data)

Definition:

- The request always fetches fresh data from the server.
- The response is **not stored in cache**.

Where is it used?

✅ **Use No Cache for:**

- Real-time applications (e.g., stock prices, live dashboards).
- Data that must always be up-to-date (e.g., latest comments, chat messages).
- Avoiding outdated or stale data.

Example in Next.js API Route (Server Actions / Route Handlers)

```
export const fetchData = async () => {
  const res = await fetch('https://api.example.com/data', {
cache: 'no-store' });
  const data = await res.json();
  return data;
};
```

🔹 **cache: 'no-store' forces Next.js to fetch fresh data every time.**

2️⃣ Cache (Static Cache - Default in SSG)

Definition:

- The request is **cached** and reused until revalidation happens.
- Improves **performance** by reducing server requests.
- Default behavior in **SSG (getStaticProps)**.

Where is it used?

✅ **Use Cached Data for:**

- Static marketing pages, documentation, blogs.
- Data that does not change frequently.
- Reducing server load and improving page speed.

Example: Cached Data in getStaticProps (SSG)

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return { props: { data }, revalidate: 60 }; // Revalidates
every 60 sec
}
```

🔹 **Data is cached and will refresh every 60 seconds.**

3️⃣ Force Revalidate (ISR - Incremental Static Regeneration)

Definition:

- Cached data is refreshed **only when a revalidation trigger happens**.
- Ensures fresh data **without rebuilding the whole site**.

Where is it used?

✅ **Use ISR for:**

- Blog posts, product listings, e-commerce pages.
- Content that updates **occasionally** but does not require real-time fetching.
- Reducing server load while keeping data fresh.

Example: ISR (Revalidate Every 60s)

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return { props: { data }, revalidate: 60 }; // Revalidate
every 60 sec
}
```

🔹 **Users get the cached page**, but after **60 seconds**, Next.js fetches fresh data.

4️⃣On-Demand Revalidation (Manual Cache Refresh)

Definition:

- Instead of automatic refresh, **an API call manually triggers cache invalidation**.
- Best for **CMS-based websites** where admins update content manually.

Where is it used?

✅ **Use On-Demand Revalidation for:**

- CMS content (WordPress, Strapi, Sanity, etc.).
- Admin dashboards that trigger content updates.

Example: Manually Revalidating a Page

```
export default async function handler(req, res) {
  await res.revalidate('/blog'); // Clears the cache for '/blog'
  res.json({ revalidated: true });
}
```

🔹 **Call this API whenever you want to refresh a page's cache.**

🔹 **Summary Table**

Caching Strategy	**Behavior**	**When to Use?**
No Cache (cache: 'no-store')	Always fetch fresh data (no caching).	Real-time apps (stock prices, chat, live dashboards).
Cache (Default in SSG)	Data is cached and reused until manually invalidated.	Static blogs, marketing pages, documentation.
Force Revalidate (ISR)	Cached data updates after a certain time.	E-commerce product listings, blogs, slowly changing data.
On-Demand Revalidation	Cache refresh triggered manually via API.	CMS-powered sites, admin dashboards updating content.

Middleware (Custom Request Handling)

- Middleware allows **custom logic before rendering pages**.
- Can be used for **authentication, redirects, logging**.

✅ **Best for:** Protecting pages, handling security, redirecting users.

Example:

```
import { NextResponse } from 'next/server';

export function middleware(req) {

  if (!req.cookies.token) {

    return NextResponse.redirect('/login');

  }

}
```

🔹 If a user is **not logged in**, they are redirected to `/login`.

Image Optimization

- Uses `<Image>` component to **automatically optimize images**.
- Reduces **image size** while keeping quality high.

✅ **Best for:** Any website that uses images.

Example:

```
import Image from 'next/image';

export default function Home() {

  return <Image src="/example.jpg" width={500} height={300}

alt="Example" />;

}
```

🔹 **Images load faster** with built-in lazy loading.