Purwadhika
Digital Technology School

Full Stack Web Development

# Server side rendering & static site generation - Part 2

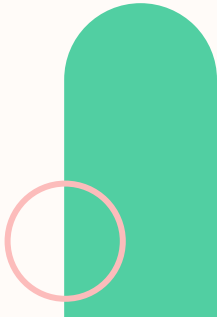
# Outline

- Data Fetching, Caching, and Revalidating
- Optimization
- Metadata
- Deployment

# Data Fetching

There are four ways you can fetch data:

- On the server, with ***fetch***
- On the server, with third-party libraries
- On the client, via a ***Route Handler***
- On the client, with third-party libraries.



# Fetching Data on the Server with *fetch*

Next.js extends the native ***fetch*** Web API to allow you to configure the **caching** and **revalidating** behavior for each fetch request on the server.

React extends ***fetch*** to automatically memoize ***fetch*** requests while rendering a React component tree.

Alternatively, you can use Axios to fetching data from server.

```
async function getData() {
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.

  if (!res.ok) {
    // This will activate the closest `error.js` Error Boundary
    throw new Error('Failed to fetch data')
  }

  return res.json()
}

export default async function Page() {
  const data = await getData()

  return <main></main>
}
```

# Fetching data on the Server with third-party libraries

In cases where you're using a **third-party library that doesn't support or expose *fetch*** (for example, a database, CMS, or ORM client), you can configure the caching and revalidating behavior of those requests using the **Route Segment Config Option** and React's cache function.

utils/get-item.js

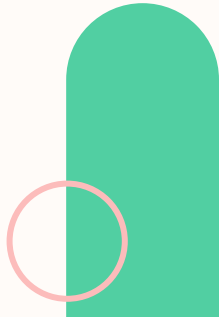
```
import { cache } from 'react'

export const revalidate = 3600 // revalidate the data at most every hour

export const getItem = cache(async (id) => {
  const item = await db.item.findUnique({ id })
  return item
})
```

app/item/[id]/page.js

```
import { getItem } from '@/utils/get-item'

export default async function Page({ params: { id } }) {
  const item = await getItem(id)
  // ...
}
```

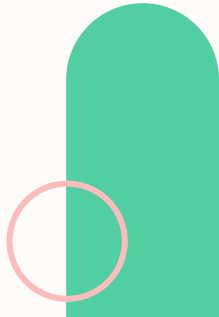
# Fetching Data on the Client with Route Handlers

If you need to fetch data in a client component, you can call a **Route Handler** from the client. **Route Handlers** execute on the server and return the data to the client. This is useful when you don't want to expose sensitive information to the client, such as API tokens.



# Fetching Data on the Client with third-party libraries

You can also fetch data on the client using a third-party library such as **SWR** or **React Query**. These libraries provide their own APIs for memoizing requests, caching, revalidating, and mutating data.



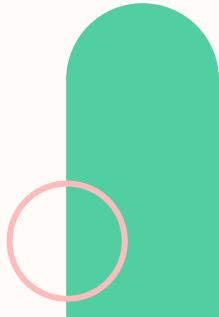
# Data Caching

**Caching** stores data so it doesn't need to be re-fetched from your data source on every request. By default, Next.js automatically caches the returned values of ***fetch*** in the **Data Cache** on the server. This means that the data can be fetched at build time or request time, cached, and reused on each data request.

# Revalidation

**Revalidation** is the process of purging the Data Cache and re-fetching the latest data. This is useful when your data changes and you want to ensure you show the latest information. Cached data can be revalidated in two ways:

- **Time-based revalidation**: Automatically revalidate data after a certain amount of time has passed. This is useful for data that changes infrequently and freshness is not as critical.
- **On-demand revalidation**: Manually revalidate data based on an event (e.g. form submission). On-demand revalidation can use a tag-based or path-based approach to revalidate groups of data at once. This is useful when you want to ensure the latest data is shown as soon as possible (e.g. when content from your headless CMS is updated).



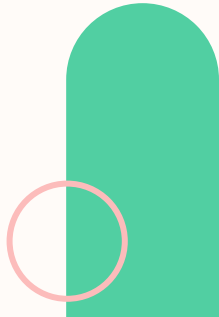
# Time-based Revalidation

To revalidate data at a timed interval, you can use the next.revalidate option of fetch to set the cache lifetime of a resource (in seconds).

```
fetch('https://...', { next: { revalidate: 3600 } })
```

Alternatively, to revalidate all fetch requests in a route segment, you can use the Segment Config Options.

```
export const revalidate = 3600 // revalidate at most every hour
```



# On-demand Revalidation

Data can be revalidated on-demand by path (**revalidatePath**) or by cache tag (**revalidateTag**) inside a **Server Action** or **Route Handler**. Next.js has a cache tagging system for invalidating fetch requests across routes.

- When using **fetch**, you have the option to tag cache entries with one or more tags.
- Then, you can call **revalidateTag** to revalidate all entries associated with that tag.

For example, the following fetch request adds the cache tag collection:

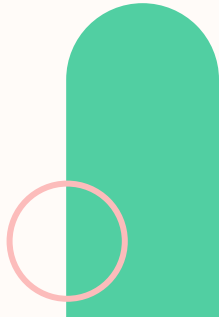
```
export default async function Page() {
  const res = await fetch('https://...', { next: { tags: ['collection'] } })
  const data = await res.json()
  // ...
}
```

```
'use server'

import { revalidateTag } from 'next/cache'

export default async function action() {
  revalidateTag('collection')
}
```

# Optimizations

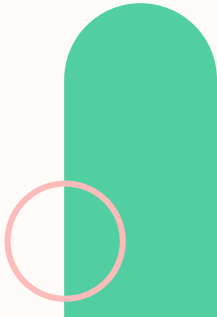
Next.js comes with a variety of built-in **optimizations** designed to improve your application's speed and **Core Web Vitals**.

- **Built-in Components** : Built-in components abstract away the complexity of implementing common UI optimizations.
- **Metadata** : Metadata helps search engines understand your content better (which can result in better SEO), and allows you to customize how your content is presented on social media, helping you create a more engaging and consistent user experience across various platforms.
- **Static Assets** : Next.js /public folder can be used to serve static assets like images, fonts, and other files. Files inside /public can also be cached by CDN providers so that they are delivered efficiently.



# Built-in Components

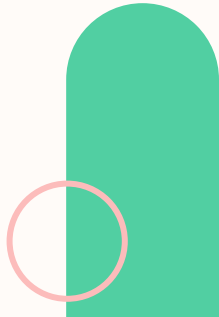
- **Images**: Built on the native <img> element. The Image Component optimizes images for performance by lazy loading and automatically resizing images based on device size.
- **Link**: Built on the native <a> tags. The Link Component prefetches pages in the background, for faster and smoother page transitions.
- **Scripts**: Built on the native <script> tags. The Script Component gives you control over loading and execution of third-party scripts.



# Font Optimization

**next/font** will automatically optimize your fonts (including custom fonts) and remove external network requests for improved privacy and performance. next/font includes built-in automatic self-hosting for any font file. This means you can optimally load web fonts with zero layout shift, thanks to the underlying CSS size-adjust property used.

This new font system also allows you to conveniently use all Google Fonts with performance and privacy in mind. CSS and font files are downloaded at build time and self-hosted with the rest of your static assets. No requests are sent to Google by the browser.

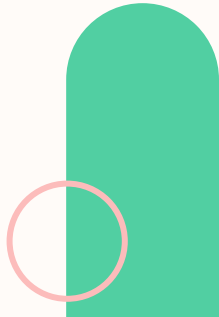


# Metadata

Next.js has a **Metadata** API that can be used to define your application metadata (e.g. meta and link tags inside your HTML head element) for **improved SEO and web shareability**. There are two ways you can add metadata to your application:

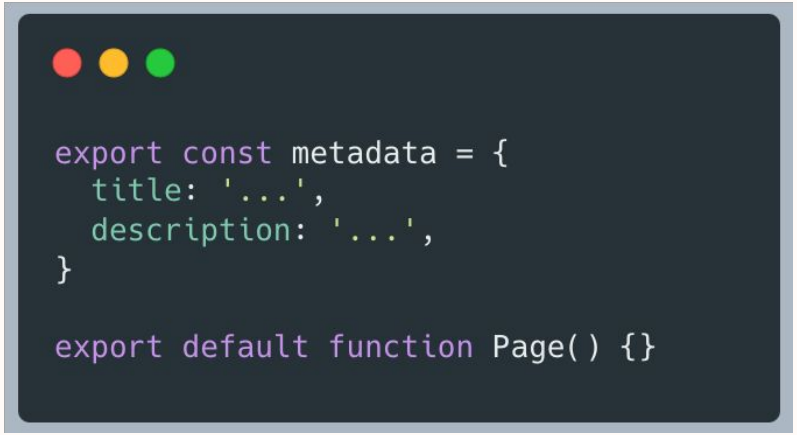
- **Config-based Metadata**: Export a static metadata object or a dynamic generateMetadata function in a layout.js or page.js file.
- **File-based Metadata**: Add static or dynamically generated special files to route segments.

With both these options, Next.js will automatically generate the relevant <head> elements for your pages. You can also create dynamic OG images using the ImageResponse constructor.



# Static Metadata

To define static metadata, export a Metadata object from a layout.js or static page.js file.



```
export const metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

# Dynamic Metadata

You can use **generateMetadata** function to fetch metadata that requires dynamic values.

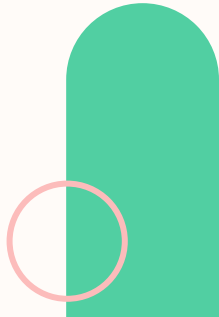
```
export async function generateMetadata({ params, searchParams }, parent) {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

export default function Page({ params, searchParams }) {}
```

# File-based Metadata

These special files are available for metadata:

- favicon.ico, apple-icon.jpg, and icon.jpg
- opengraph-image.jpg and twitter-image.jpg
- robots.txt
- sitemap.xml

You can use these for static metadata, or you can programmatically generate these files with code.



# Static Assets

Next.js can serve static files, like images, under a folder called public in the root directory. Files inside public can then be referenced by your code starting from the base URL (/). For example, if you add me.png inside public, the following code will access the image:

```
import Image from 'next/image'

export function Avatar() {
  return <Image src="/me.png" alt="me" width="64" height="64" />
}
```

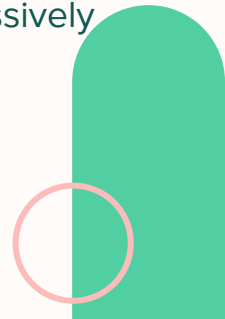
# Lazy Loading

**Lazy loading** in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route. It allows you to defer loading of Client Components and imported libraries, and only include them in the client bundle when they're needed. For example, you might want to defer loading a modal until a user clicks to open it.

There are two ways you can implement lazy loading in Next.js:

- Using **Dynamic Imports** with **next/dynamic**
- Using **React.lazy()** with **Suspense**

By default, **Server Components** are automatically **code split**, and you can use streaming to progressively send pieces of UI from the server to the client. **Lazy loading applies to Client Components**.

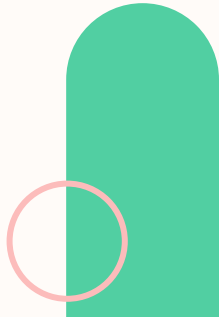


# Lazy Loading

```
'use client'

import { useState } from 'react'
import dynamic from 'next/dynamic'

// Client Components:
const ComponentA = dynamic(() => import('../components/A'))
const ComponentB = dynamic(() => import('../components/B'))
const ComponentC = dynamic(() => import('../components/C'), { ssr: false })

export default function ClientComponentExample() {
  const [showMore, setShowMore] = useState(false)

  return (
    <div>
      {/* Load immediately, but in a separate client bundle */}
      <ComponentA />

      {/* Load on demand, only when/if the condition is met */}
      {showMore && <ComponentB />}
      <button onClick={() => setShowMore(!showMore)}>Toggle</button>

      {/* Load only on the client side */}
      <ComponentC />
    </div>
  )
}
```

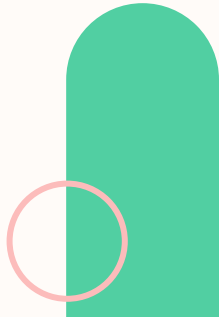
# Deployment

https://nextjs.org/learn-pages-router/basics/deploying-nextjs-app/deploy



# Exercise

Optimize and improve metadata of your portfolio website (in Next.js) and deploy!



# Thank You!