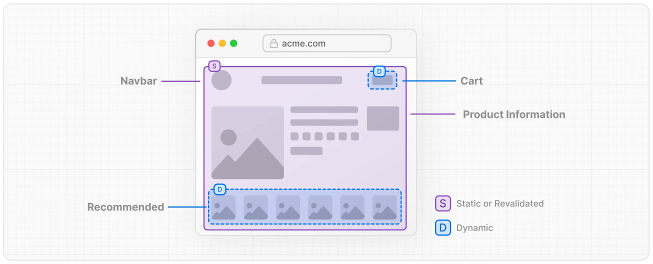
Partial Prerendering

Partial Prerendering (PPR) enables you to combine static and dynamic components together in the same route.

During the build, Next.js prerenders as much of the route as possible. If dynamic code is detected, like reading from the incoming request, you can wrap the relevant component with a React Suspense ↗ boundary. The Suspense boundary fallback will then be included in the prerendered HTML.

Note: Partial Prerendering is an **experimental** feature and subject to change. It is not ready for production use.



🎥 **Watch:** Why PPR and how it works → YouTube (10 minutes) ↗.

Background

PPR enables your Next.js server to immediately send prerendered content.

To prevent client to server waterfalls, dynamic components begin streaming from the server in parallel while serving the initial prerender. This ensures dynamic components can begin rendering before client JavaScript has been loaded in the browser.

To prevent creating many HTTP requests for each dynamic component, PPR is able to combine the static prerender and dynamic components together into a single HTTP request. This ensures there are not multiple network roundtrips needed for each dynamic component.

Using Partial Prerendering

Incremental Adoption (Version 15)

In Next.js 15, you can incrementally adopt Partial Prerendering in layouts and pages by setting the `ppr` option in `next.config.js` to `incremental`, and exporting the `experimental_ppr` route config option at the top of the file:

next.config.ts TypeScript

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    ppr: 'incremental',
  },
}

export default nextConfig
```

app/page.tsx TypeScript

```
import { Suspense } from "react"
import { StaticComponent, DynamicComponent, Fal

export const experimental_ppr = true

export default function Page() {
  return {
     <>
      <StaticComponent />
      <Suspense fallback={<Fallback />}>
        <DynamicComponent />
      </Suspense>
     </>
  };
}
```

Good to know:

- Routes that don't have `experimental_ppr` will default to `false` and will not be prerendered using PPR. You need to explicitly opt-in to PPR for each route.
- `experimental_ppr` will apply to all children of the route segment, including nested layouts and pages. You don't have to add it to every file, only the top segment of a route.
- To disable PPR for children segments, you can set `experimental_ppr` to `false` in the child segment.

Enabling PPR (Version 14)

For version 14, you can enable it by adding the `ppr` option to your `next.config.js` file. This will apply to all routes in your application:

next.config.ts TypeScript

```ts
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    ppr: true,
  },
}

export default nextConfig
```

Dynamic Components

When creating the prender for your route during `next build`, Next.js requires that dynamic functions are wrapped with React Suspense. The `fallback` is then included in the prerender.

For example, using functions like `cookies()` or `headers()`:

app/user.tsx TypeScript

```tsx
import { cookies } from 'next/headers'
```

```tsx
export function User() {
  const session = cookies().get('session')?.valu
  return '...'
}
```

This component requires looking at the incoming request to read cookies. To use this with PPR, you should wrap the component with Suspense:

app/page.tsx TypeScript

```tsx
import { Suspense } from 'react'
import { User, AvatarSkeleton } from './user'

export const experimental_ppr = true

export default function Page() {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Suspense fallback={<AvatarSkeleton />}>
        <User />
      </Suspense>
    </section>
  )
}
```

Components only opt into dynamic rendering when the value is accessed.

For example, if you are reading `searchParams` from a `page`, you can forward this value to another component as a prop:

app/page.tsx TypeScript

```tsx
import { Table } from './table'

export default function Page({
  searchParams,
}: {
  searchParams: { sort: string }
}) {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Table searchParams={searchParams} />
    </section>
  )
}
```

Inside of the table component, accessing the value from `searchParams` will make the component run dynamically:

app/table.tsx TypeScript

```tsx
export function Table({ searchParams }: { search
  const sort = searchParams.sort === 'true'
  return '...'
}
```

Previous
Composition Patterns

Next
Runtimes

Was this helpful?