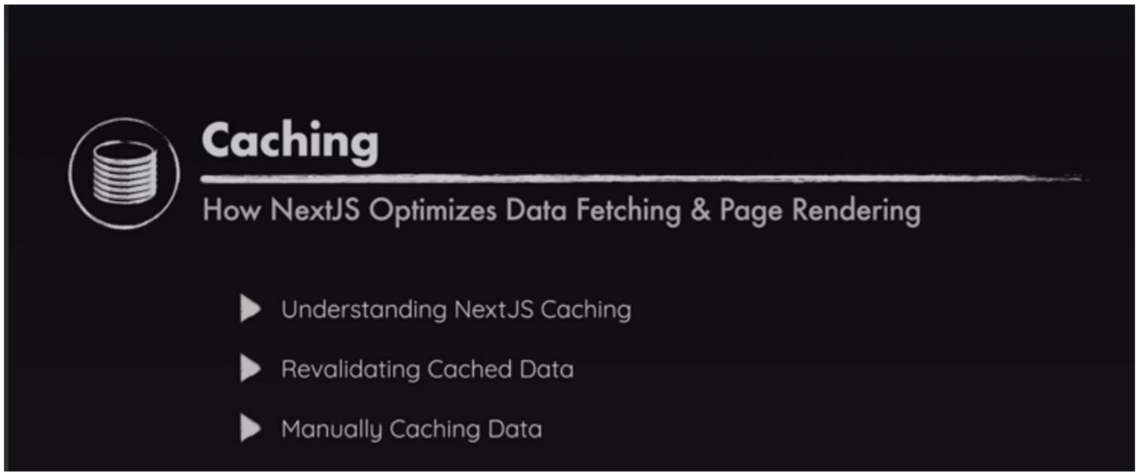
Section 7: Understanding & Configuring Catching

14 September 2024



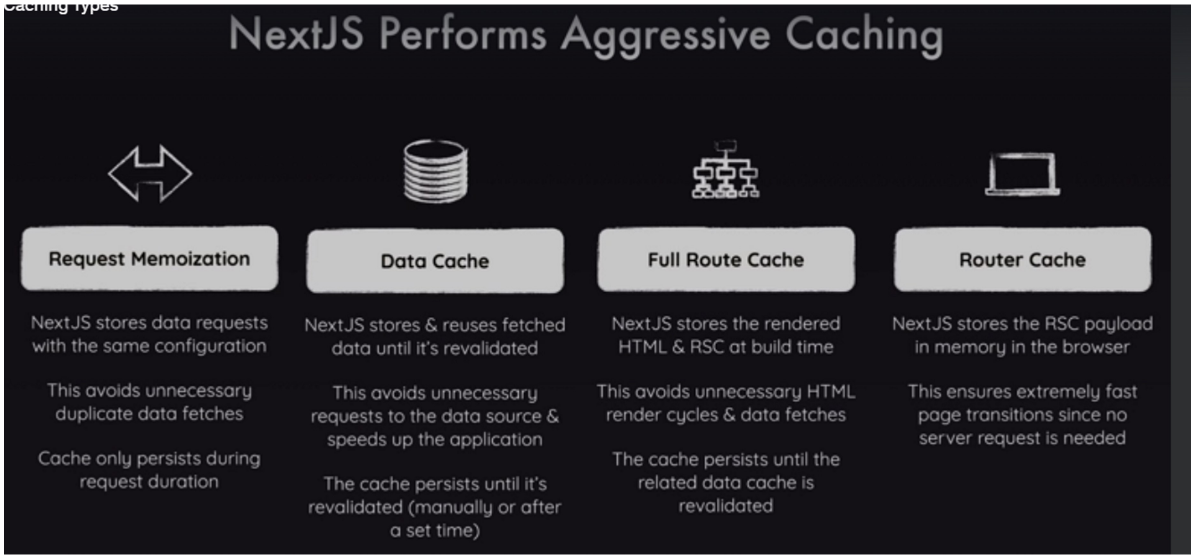
Caching

How NextJS Optimizes Data Fetching & Page Rendering

- Understanding NextJS Caching
- Revalidating Cached Data
- Manually Caching Data

NextJS performs some pretty aggressive caching. And to be precise, there are four different areas where NextJS caches data or pages, and that starts with request memorization, which is a fancy term which at the end simply means that NextJS stores data requests with the same configuration so that duplicate requests are avoided.
That's the main idea. And this request memorization only happens during one request that's being handled on the NextJS server. This still might not be 100% clear, but I'll show you how request memorization works and what it means for you in detail at a concrete example.

Caching Types



NextJS Performs Aggressive Caching

Request Memoization	Data Cache	Full Route Cache	Router Cache
NextJS stores data requests with the same configuration	NextJS stores & reuses fetched data until it's revalidated	NextJS stores the rendered HTML & RSC at build time	NextJS stores the RSC payload in memory in the browser
This avoids unnecessary duplicate data fetches	This avoids unnecessary requests to the data source & speeds up the application	This avoids unnecessary HTML render cycles & data fetches	This ensures extremely fast page transitions since no server request is needed
Cache only persists during request duration	The cache persists until it's revalidated (manually or after a set time)	The cache persists until the related data cache is revalidated	

There also is a separate data cache managed by NextJS, which might sound similar, but which actually has a different purpose.
Where request memorization aims to avoid duplicate requests to a data source for a single request that's handled by the NextJS server. The data cache is all about storing and reusing data that has been fetched
from a data source if it hasn't changed. The idea behind this data cache, therefore, is not to deduplicate requests or avoid duplicate requests, but instead to avoid requests altogether unless the data has changed,

which, of course, means that your application may be a bit faster because those extra round trips to the data source are avoided. And this data cache then persists until it's revalidated manually by you.

NextJS also manages a full route cache. And the idea here is that NextJS does not just cache and store and reuse data
that may be used by pages, but instead the entire page, the entire HTML code, and the entire React Server Component payload,
which is managed under the hood and which is used to render those pages.

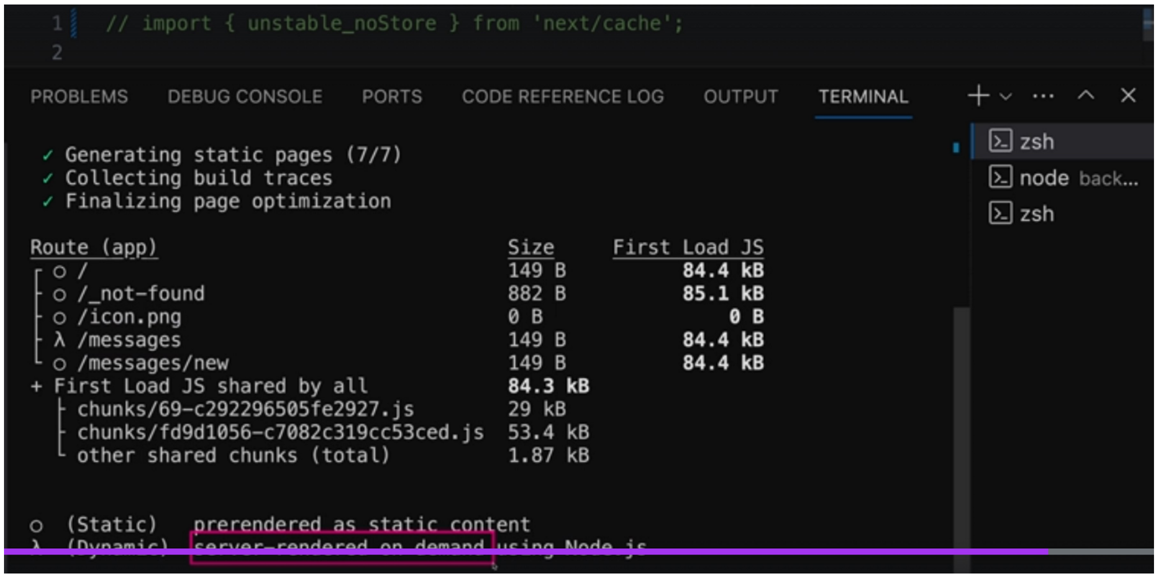
This, therefore, does not just avoid extra round trips to some data source, but it instead entirely avoids the re-rendering
of an entire HTML page. And hence, it speeds up pages even further because existing pages can be reused.
That's the idea here. And this cache also persists until the related data cache is revalidated.
there is a router cache and one key difference here is that all three other caches:
request memorization, data cache, and full route cache, all these caches are managed on the server side.

The router cache, on the other hand, is a cache that's managed on the client side.

And here, NextJS also stores some React Server Component payload in memory in the browser.
And it does that so that navigations between pages can happen faster

because even with the full route cache, NextJS, of course, would need to send a request to the NextJS server to get that cached page and that request to the server can be sped up
or avoided if the page data is already managed on the client side, and that's the idea behind the router cache.

Making Sense of The Full Route Cache



```
// import { unstable_noStore } from 'next/cache';
```

```
 ✓ Generating static pages (7/7)
 ✓ Collecting build traces
 ✓ Finalizing page optimization

Route (app)                              Size     First Load JS
┌ ○ /                                    149 B          84.4 kB
├ ○ /_not-found                          882 B          85.1 kB
├ ○ /icon.png                            0 B                0 B
├ λ /messages                            149 B          84.4 kB
└ ○ /messages/new                        149 B          84.4 kB
+ First Load JS shared by all            84.3 kB
  ├ chunks/69-c292296505fe2927.js        29 kB
  ├ chunks/fd9d1056-c7082c319cc53ced.js  53.4 kB
  └ other shared chunks (total)          1.87 kB


○  (Static)   prerendered as static content
λ  (Dynamic)  server-rendered on demand using Node.js
```

So we did now learn about a lot of different cache settings, and we learned how those settings can be used
to control the data cache. Now, besides that data cache, NextJS also has a full route cache,
and that's actually the kind of cache we saw in action in this application before.

Because that full route cache is actually already created and initialized at build time, which simply means that if I stop that development server and I build this NextJS application for production by running

npm run build,

NextJS goes ahead and pre-renders all the pages of this application, unless they're dynamic pages with a dynamic parameter,
because there, by default, it doesn't know all the possible dynamic parameter values, but it pre-renders all the pages it's able to pre-render.
You also see that here, that all those pages marked with a dot were pre-rendered,
so the messages page, for example, was pre-rendered, as you can tell, and the implication of that is
that when we start the production server with

npm start,

it will actually be those pre-rendered pages that we see out of the box, and those pages
are actually cached by NextJS, you could say.

Therefore, now with that production server up and running, if I clear and restart my backend,

```
cd backend
npm start
```

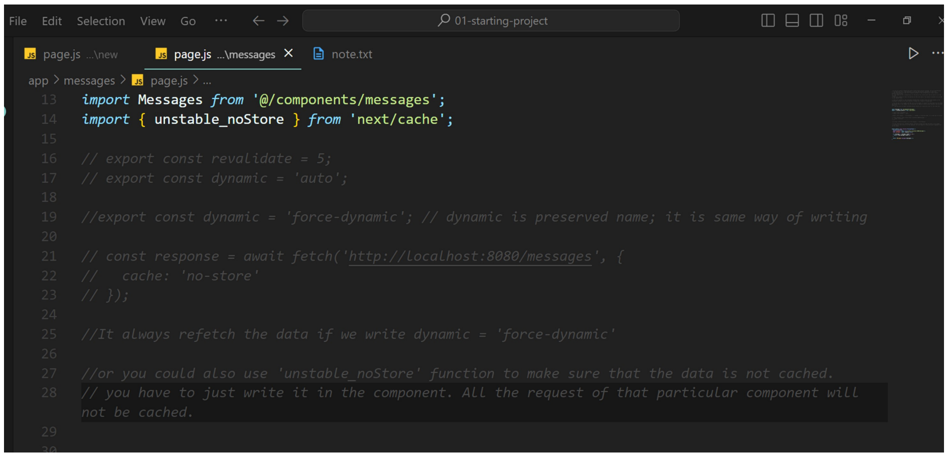
we will actually see no log message at all here when I reload that messages page,
and we see no log message here at all because that page was pre-rendered at build time
and it's cached, and it's now always that pre-rendered cached page that's being served here,

no matter how often I reload or navigate away and come back. It's always that pre-rendered page.
Hence, we got no logs here on that backend server. Now, that's, of course, fine, as long as the data here doesn't change,
and in this dummy application with that dummy backend, that's the case because the data served
from that dummy backend does never change, but we saw before in the previous section where we learned about data mutations with NextJS that this can be a problem if we have data that does change, because due to that pre-rendered cached page,

NextJS never re-renders and updates that page, And that's another thing that will be controlled
by setting one of these cache settings here. For example, if I set force

dynamic to this entire file to make sure that there is no data caching,

NextJS will always re-render that page whenever a request is sent to it,
which ensures that we're always fetching fresh data,
and that's why, when I enable this dynamic constant and set it to force dynamic here, and I then save that file,
and I then rebuild my project to rebuild it for production, now, this messages route has a different symbol here,
which means that it's a dynamic page now, which is server-rendered on demand,

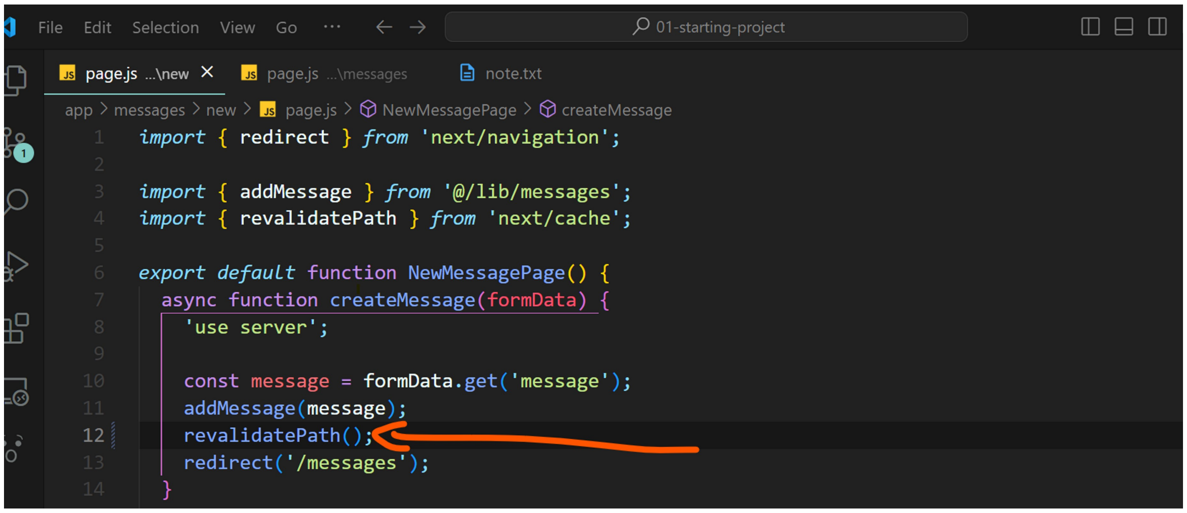


```
import Messages from '@/components/messages';
import { unstable_noStore } from 'next/cache';

// export const revalidate = 5;
// export const dynamic = 'auto';

//export const dynamic = 'force-dynamic'; // dynamic is preserved name; it is same way of writing

// const response = await fetch('http://localhost:8080/messages', {
//   cache: 'no-store'
// });

//It always refetch the data if we write dynamic = 'force-dynamic'

//or you could also use 'unstable_noStore' function to make sure that the data is not cached.
// you have to just write it in the component. All the request of that particular component will
not be cached.

```

as you can tell down here. And indeed, that's what we can see when we start the production server with npm start.

Now, if we take a look at this backend log, if I reload this page a couple of times, I see a bunch of logs here,
because now it's not just the data cache that's cleared, but also that full route cache,
and we make sure that it's not some pre-rendered page that's reused over and over again. Of course, we don't have to use this setting to make sure that we see updated data.

A different approach would be to revalidate the cache of a specific page on demand with the revalidate path function we already used in the previous section, and that's now a function which we'll revisit in the next lecture
before we then thereafter will take a look at how we can manage caching



```
import { redirect } from 'next/navigation';

import { addMessage } from '@/lib/messages';
import { revalidatePath } from 'next/cache';

export default function NewMessagePage() {
  async function createMessage(formData) {
    'use server';

    const message = formData.get('message');
    addMessage(message);
    revalidatePath();
    redirect('/messages');
  }
```