Caching in Next.js

Next.js improves your application's performance and reduces costs by caching rendering work and data requests. This page provides an in-depth look at Next.js caching mechanisms, the APIs you can use to configure them, and how they interact with each other.

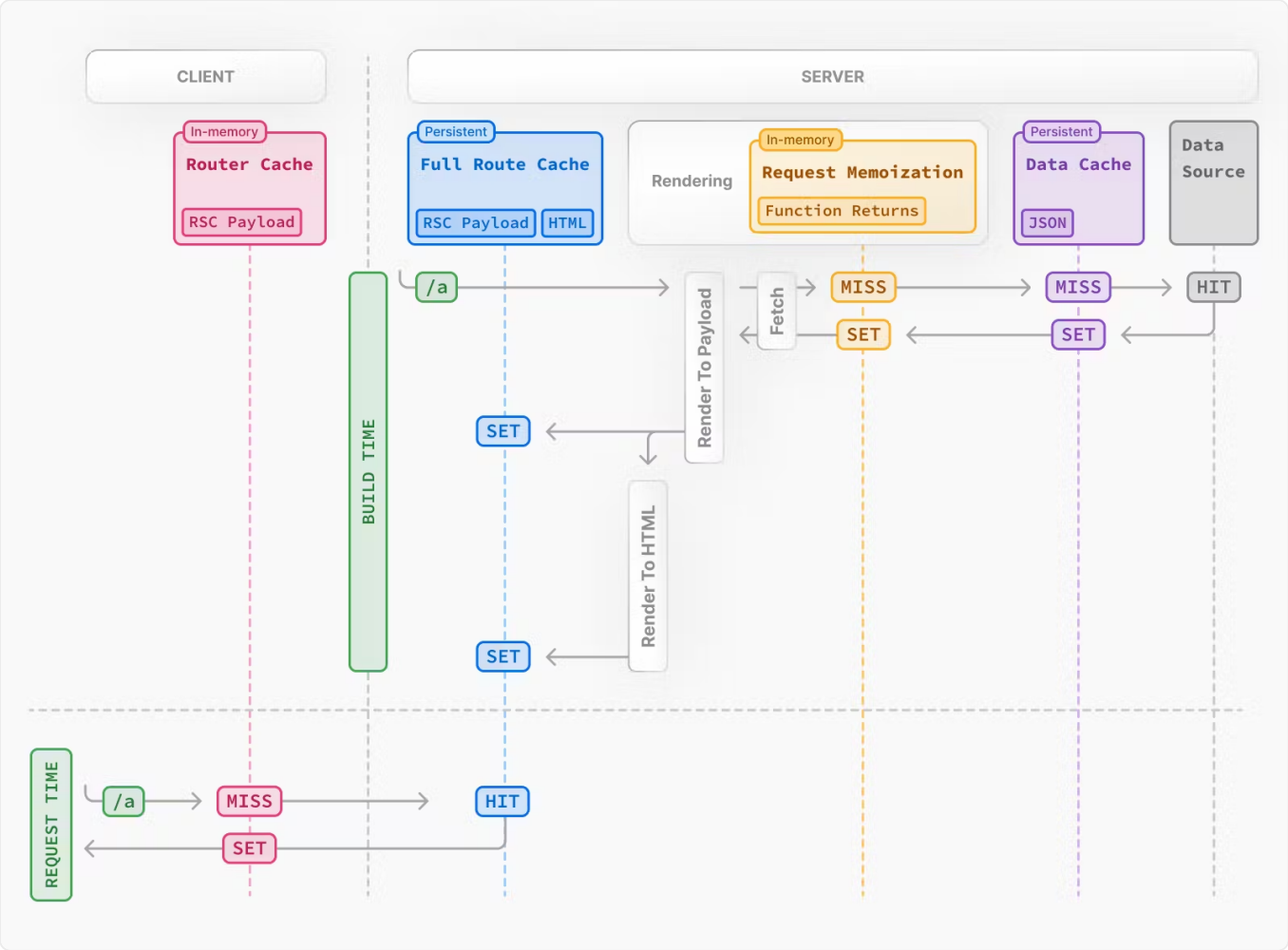
Good to know: This page helps you understand how Next.js works under the hood but is **not** essential knowledge to be productive with Next.js. Most of Next.js' caching heuristics are determined by your API usage and have defaults for the best performance with zero or minimal configuration.

Overview

Here's a high-level overview of the different caching mechanisms and their purpose:

Mechanism	What	Where	Purpose	Duration
Request Memoization	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
Data Cache	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
Router Cache	RSC Payload	Client	Reduce server requests on navigation	User session or time-based

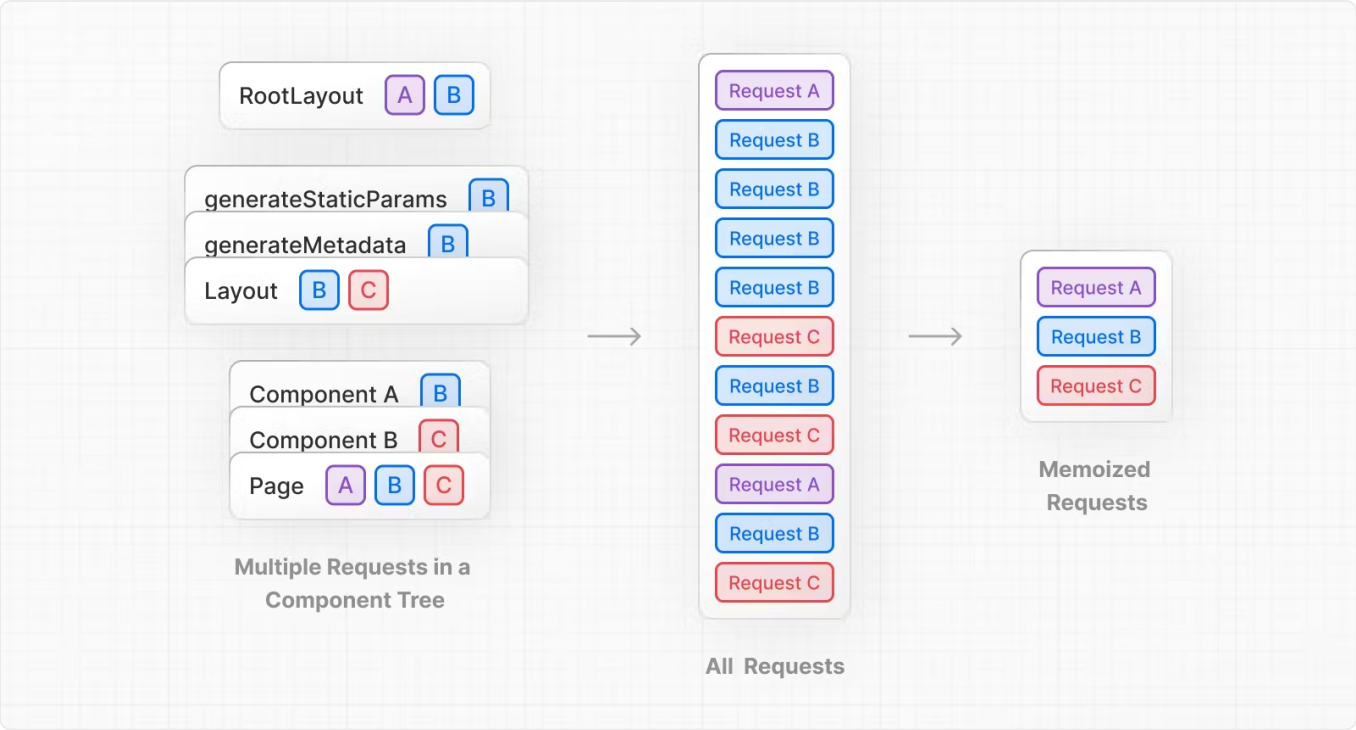
By default, Next.js will cache as much as possible to improve performance and reduce cost. This means routes are **statically rendered** and data requests are **cached** unless you opt out. The diagram below shows the default caching behavior: when a route is statically rendered at build time and when a static route is first visited.



Caching behavior changes depending on whether the route is statically or dynamically rendered, data is cached or uncached, and whether a request is part of an initial visit or a subsequent navigation. Depending on your use case, you can configure the caching behavior for individual routes and data requests.

Request Memoization

React extends the `fetch` API to automatically **memoize** requests that have the same URL and options. This means you can call a fetch function for the same data in multiple places in a React component tree while only executing it once.



For example, if you need to use the same data across a route (e.g. in a Layout, Page, and multiple components), you do not have to fetch data at the top of the tree then forward props between components. Instead, you can fetch data in the components that need it without worrying about the performance implications of making multiple requests across the network for the same data.

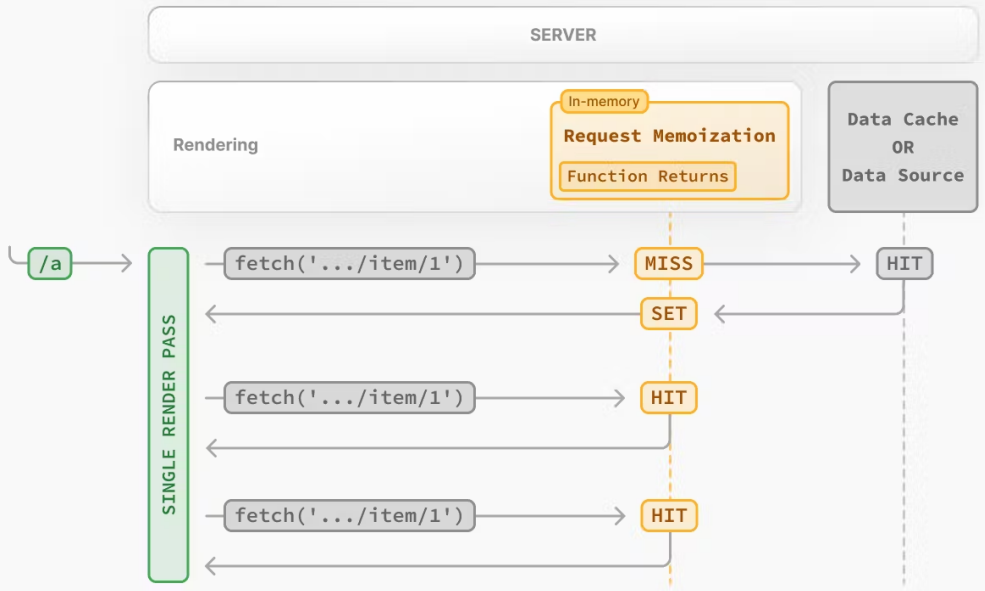
app/example.tsx TypeScript

```tsx
async function getItem() {
  // The `fetch` function is automatically memoized and the result
  // is cached
  const res = await fetch('https://.../item/1')
  return res.json()
}

// This function is called twice, but only executed the first time
const item = await getItem() // cache MISS

// The second call could be anywhere in your route
const item = await getItem() // cache HIT
```

How Request Memoization Works



- While rendering a route, the first time a particular request is called, its result will not be in memory and it'll be a cache `MISS`.
- Therefore, the function will be executed, and the data will be fetched from the external source, and the result will be stored in memory.
- Subsequent function calls of the request in the same render pass will be a cache `HIT`, and the data will be returned from memory without executing the function.
- Once the route has been rendered and the rendering pass is complete, memory is "reset" and all request memoization entries are cleared.

Good to know:

- Request memoization is a React feature, not a Next.js feature. It's included here to show how it interacts with the other caching mechanisms.
- Memoization only applies to the `GET` method in `fetch` requests.
- Memoization only applies to the React Component tree, this means:
  - It applies to `fetch` requests in `generateMetadata`, `generateStaticParams`, Layouts, Pages, and other Server Components.
  - It doesn't apply to `fetch` requests in Route Handlers as they are not a part of the React component tree.
- For cases where `fetch` is not suitable (e.g. some database clients, CMS clients, or GraphQL clients), you can use the React `cache` function to memoize functions.

Duration

The cache lasts the lifetime of a server request until the React component tree has finished rendering.

Revalidating

Since the memoization is not shared across server requests and only applies during rendering, there is no need to revalidate it.

Opting out

To opt out of memoization in `fetch` requests, you can pass an `AbortController` `signal` to the request.

app/example.js

```
const { signal } = new AbortController()
fetch(url, { signal })
```

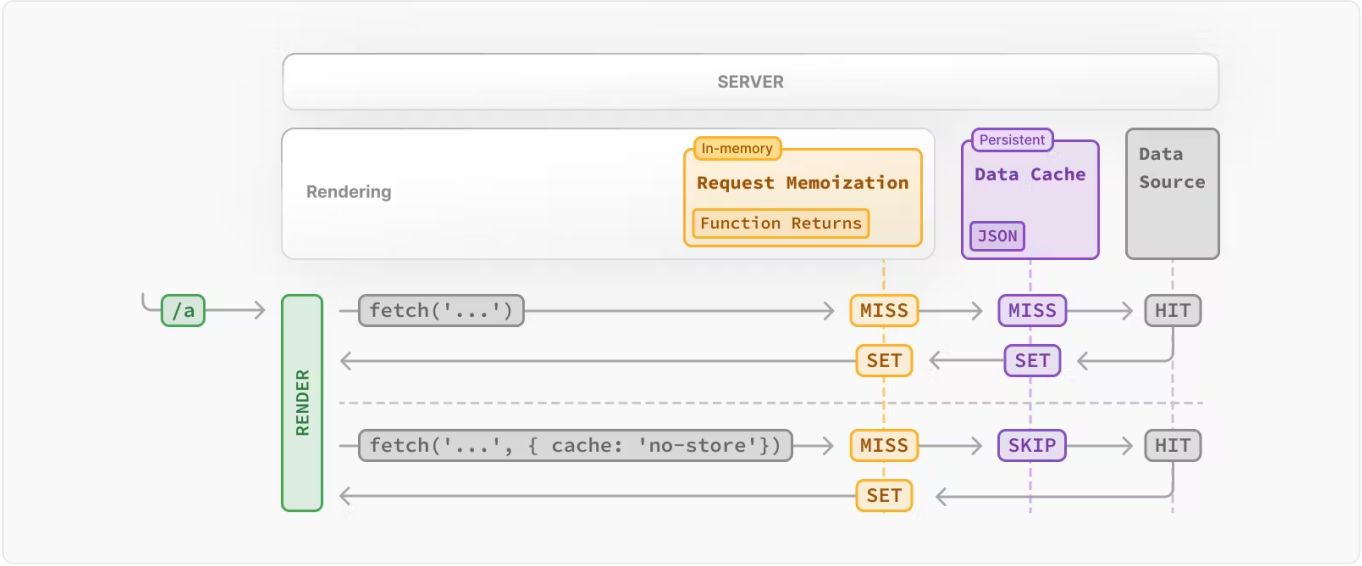
Data Cache

Next.js has a built-in Data Cache that **persists** the result of data fetches across incoming **server requests** and **deployments**. This is possible because Next.js extends the native `fetch` API to allow each request on the server to set its own persistent caching semantics.

Good to know: In the browser, the `cache` option of `fetch` indicates how a request will interact with the browser's HTTP cache, in Next.js, the `cache` option indicates how a server-side request will interact with the server's Data Cache.

By default, data requests that use `fetch` are **cached**. You can use the `cache` and `next.revalidate` options of `fetch` to configure the caching behavior.

How the Data Cache Works



- The first time a `fetch` request is called during rendering, Next.js checks the Data Cache for a cached response.
- If a cached response is found, it's returned immediately and memoized.
- If a cached response is not found, the request is made to the data source, the result is stored in the Data Cache, and memoized.
- For uncached data (e.g. `{ cache: 'no-store' }`), the result is always fetched from the data source, and memoized.
- Whether the data is cached or uncached, the requests are always memoized to avoid making duplicate requests for the same data during a React render pass.

Differences between the Data Cache and Request Memoization

While both caching mechanisms help improve performance by re-using cached data, the Data Cache is persistent across incoming requests and deployments, whereas memoization only lasts the lifetime of a request.

With memoization, we reduce the number of **duplicate** requests in the same render pass that have to cross the network boundary from the rendering server to the Data Cache server (e.g. a CDN or Edge Network) or data source (e.g. a database or CMS). With the Data Cache, we reduce the number of requests made to our origin data source.

Duration

The Data Cache is persistent across incoming requests and deployments unless you revalidate or opt-out.

Revalidating

Cached data can be revalidated in two ways, with:

- **Time-based Revalidation**: Revalidate data after a certain amount of time has passed and a new request is made. This is useful for data that changes infrequently and freshness is not as critical.
- **On-demand Revalidation:** 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).

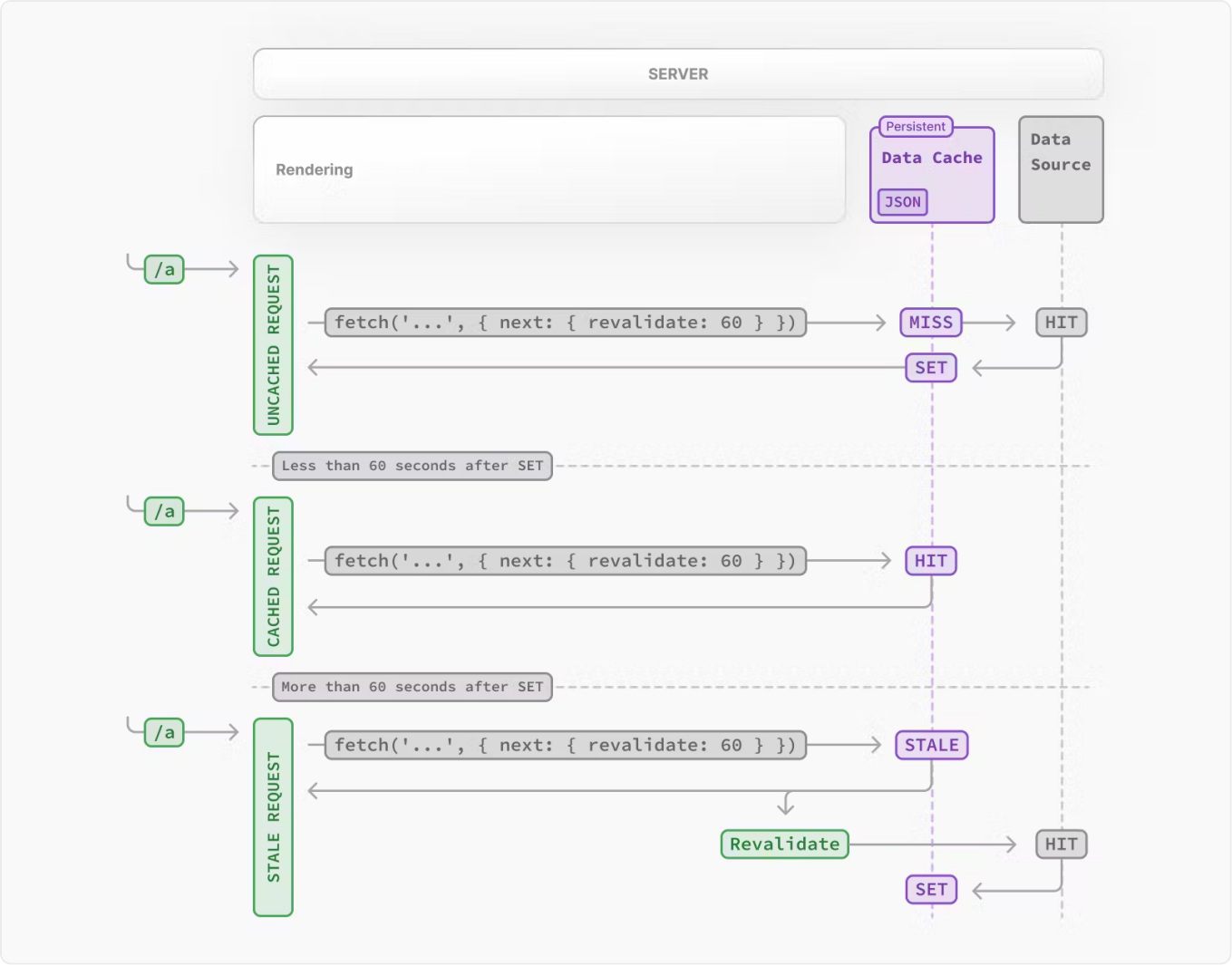
Time-based Revalidation

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).

```
1  // Revalidate at most every hour
2  fetch('https://...', { next: { revalidate: 3600 } })
```

Alternatively, you can use Route Segment Config options to configure all `fetch` requests in a segment or for cases where you're not able to use `fetch`.

How Time-based Revalidation Works



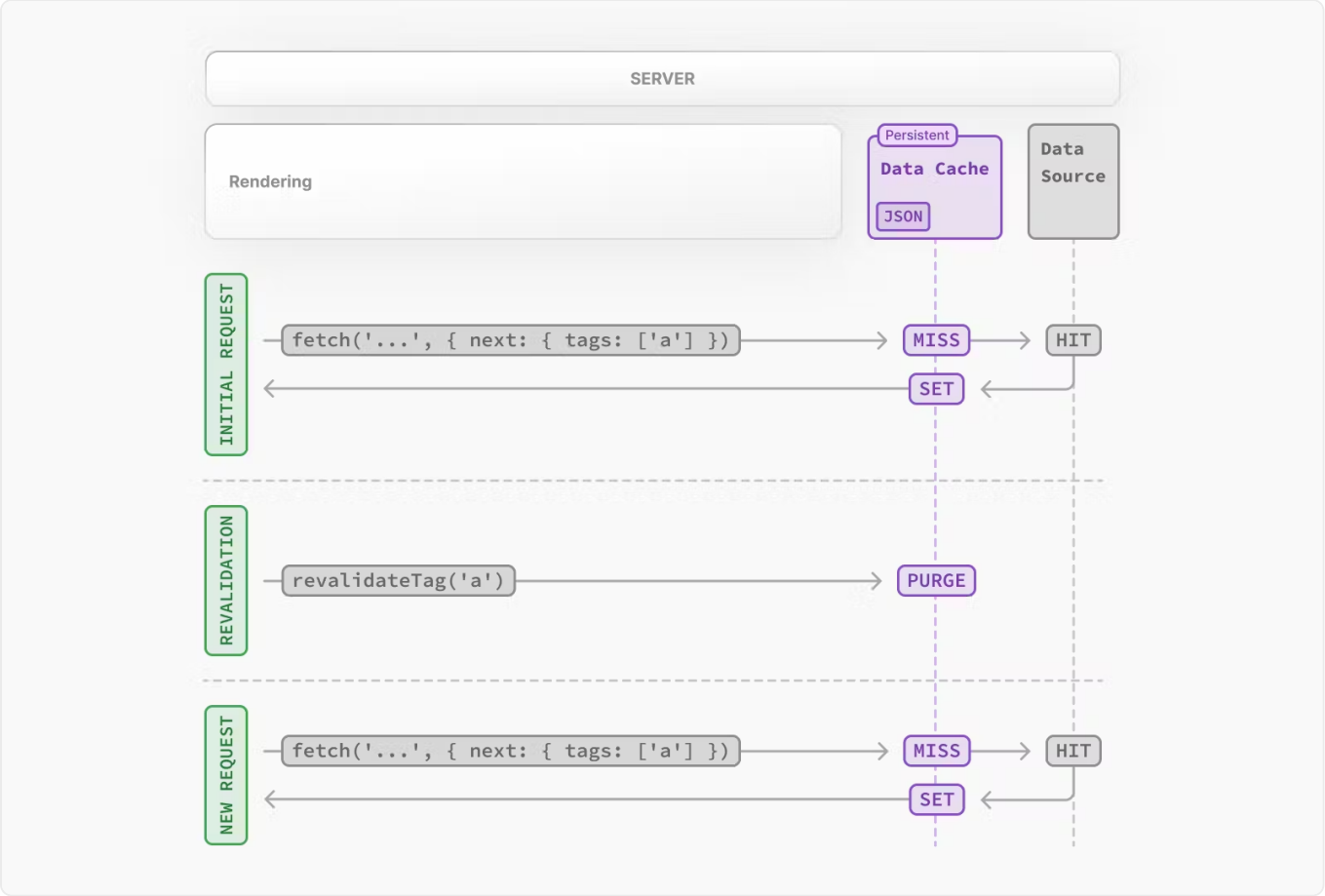
- The first time a fetch request with `revalidate` is called, the data will be fetched from the external data source and stored in the Data Cache.
- Any requests that are called within the specified timeframe (e.g. 60-seconds) will return the cached data.
- After the timeframe, the next request will still return the cached (now stale) data.
  - Next.js will trigger a revalidation of the data in the background.
  - Once the data is fetched successfully, Next.js will update the Data Cache with the fresh data.
  - If the background revalidation fails, the previous data will be kept unaltered.

This is similar to **stale-while-revalidate** ↗ behavior.

On-demand Revalidation

Data can be revalidated on-demand by path (`revalidatePath`) or by cache tag (`revalidateTag`).

How On-Demand Revalidation Works



- The first time a `fetch` request is called, the data will be fetched from the external data source and stored in the Data Cache.
- When an on-demand revalidation is triggered, the appropriate cache entries will be purged from the cache.
  - This is different from time-based revalidation, which keeps the stale data in the cache until the fresh data is fetched.
- The next time a request is made, it will be a cache `MISS` again, and the data will be fetched from the external data source and stored in the Data Cache.

Opting out

For individual data fetches, you can opt out of caching by setting the `cache` option to `no-store`. This means data will be fetched whenever `fetch` is called.

```
1  // Opt out of caching for an individual `fetch` request
2  fetch(`https://...`, { cache: 'no-store' })
```

Alternatively, you can also use the Route Segment Config options to opt out of caching for a specific route segment. This will affect all data requests in the route segment, including third-party libraries.

```
// Opt out of caching for all data requests in the route segment
export const dynamic = 'force-dynamic'
```

Vercel Data Cache

If your Next.js application is deployed to Vercel, we recommend reading the Vercel Data Cache ↗ documentation for a better understanding of Vercel specific features.

Full Route Cache

Related terms:

You may see the terms **Automatic Static Optimization**, **Static Site Generation**, or **Static Rendering** being used interchangeably to refer to the process of rendering and caching routes of your application at build time.

Next.js automatically renders and caches routes at build time. This is an optimization that allows you to serve the cached route instead of rendering on the server for every request, resulting in faster page loads.

To understand how the Full Route Cache works, it's helpful to look at how React handles rendering, and how Next.js caches the result:

1. React Rendering on the Server

On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks: by individual routes segments and Suspense boundaries.

Each chunk is rendered in two steps:

1. React renders Server Components into a special data format, optimized for streaming, called the **React Server Component Payload**.

2. Next.js uses the React Server Component Payload and Client Component JavaScript instructions to render **HTML** on the server.

This means we don't have to wait for everything to render before caching the work or sending a response. Instead, we can stream a response as work is completed.

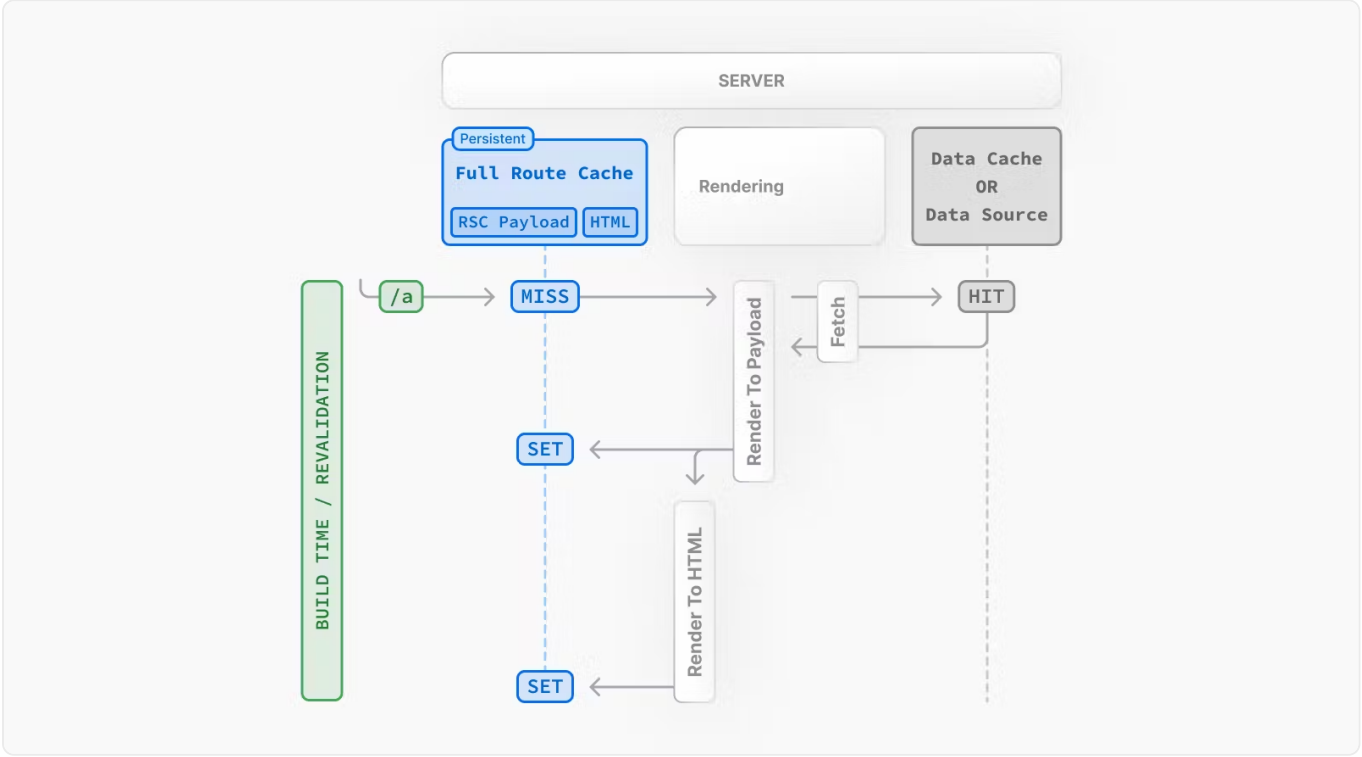
What is the React Server Component Payload?

The React Server Component Payload is a compact binary representation of the rendered React Server Components tree. It's used by React on the client to update the browser's DOM. The React Server Component Payload contains:

- The rendered result of Server Components
- Placeholders for where Client Components should be rendered and references to their JavaScript files
- Any props passed from a Server Component to a Client Component

To learn more, see the Server Components documentation.

2. Next.js Caching on the Server (Full Route Cache)



The default behavior of Next.js is to cache the rendered result (React Server Component Payload and HTML) of a route on the server. This applies to statically rendered routes at build time, or during revalidation.

3. React Hydration and Reconciliation on the Client

At request time, on the client:

1. The HTML is used to immediately show a fast non-interactive initial preview of the Client and Server Components.
2. The React Server Components Payload is used to reconcile the Client and rendered Server Component trees, and update the DOM.
3. The JavaScript instructions are used to hydrate ↗ Client Components and make the application interactive.

4. Next.js Caching on the Client (Router Cache)

The React Server Component Payload is stored in the client-side Router Cache - a separate in-memory cache, split by individual route segment. This Router Cache is used to improve the navigation experience by storing previously visited routes and prefetching future routes.

5. Subsequent Navigations

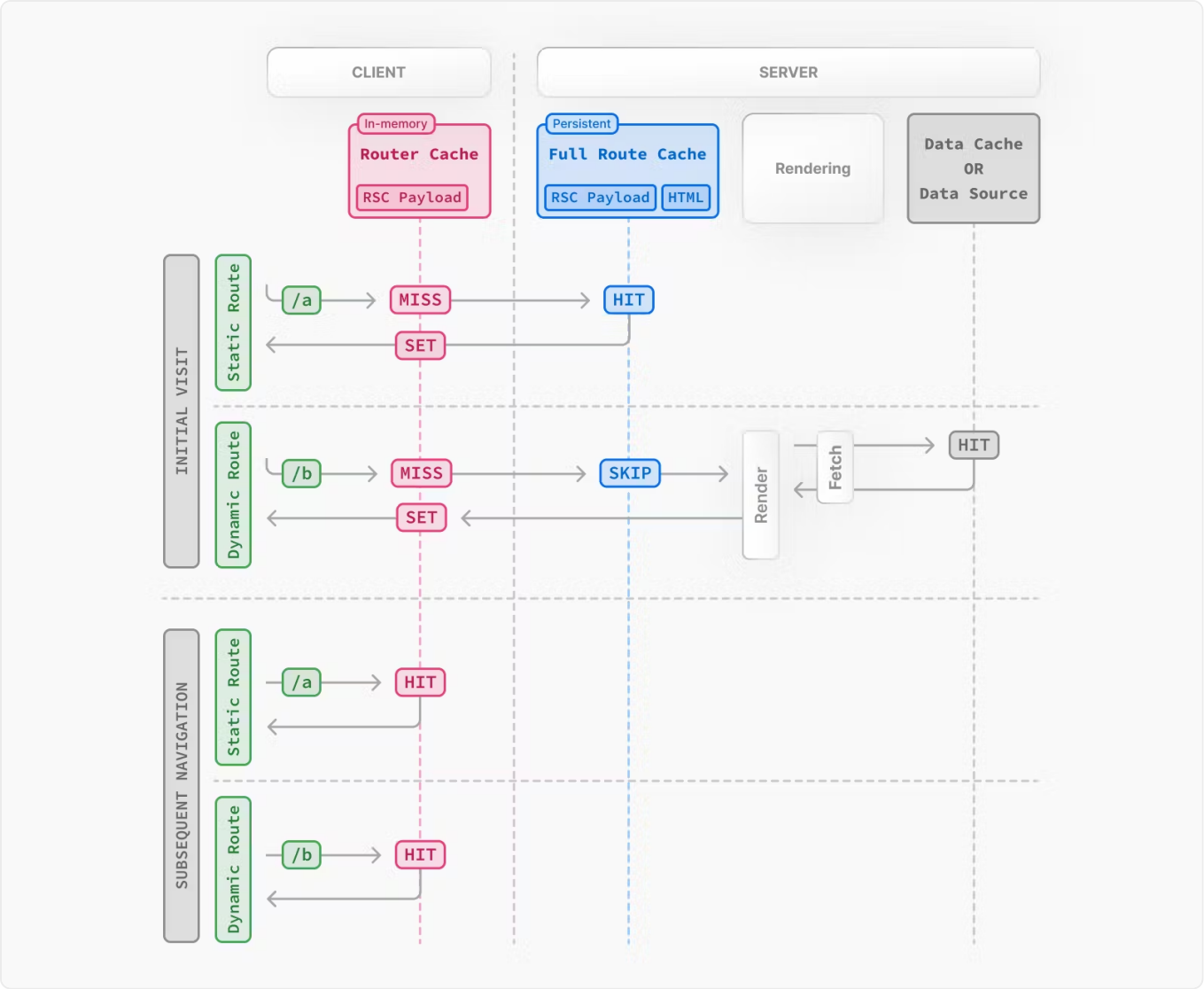
On subsequent navigations or during prefetching, Next.js will check if the React Server Components Payload is stored in the Router Cache. If so, it will skip sending a new request to the server.

If the route segments are not in the cache, Next.js will fetch the React Server Components Payload from the server, and populate the Router Cache on the client.

Static and Dynamic Rendering

Whether a route is cached or not at build time depends on whether it's statically or dynamically rendered. Static routes are cached by default, whereas dynamic routes are rendered at request time, and not cached.

This diagram shows the difference between statically and dynamically rendered routes, with cached and uncached data:



Learn more about static and dynamic rendering.

Duration

By default, the Full Route Cache is persistent. This means that the render output is cached across user requests.

Invalidation

There are two ways you can invalidate the Full Route Cache:

- **Revalidating Data**: Revalidating the Data Cache, will in turn invalidate the Router Cache by re-rendering components on the server and caching the new render output.
- **Redeploying**: Unlike the Data Cache, which persists across deployments, the Full Route Cache is cleared on new deployments.

Opting out

You can opt out of the Full Route Cache, or in other words, dynamically render components for every incoming request, by:

- **Using a Dynamic Function**: This will opt the route out from the Full Route Cache and dynamically render it at request time. The Data Cache can still be used.
- **Using the `dynamic = 'force-dynamic'` or `revalidate = 0` route segment config options**: This will skip the Full Route Cache and the Data Cache. Meaning components will be rendered and data fetched on every incoming request to the server. The Router Cache will still apply as it's a client-side cache.
- **Opting out of the Data Cache**: If a route has a `fetch` request that is not cached, this will opt the route out of the Full Route Cache. The data for the specific `fetch` request will be fetched for every incoming request. Other `fetch` requests that do not opt out of caching will still be cached in the Data Cache. This allows for a hybrid of cached and uncached data.

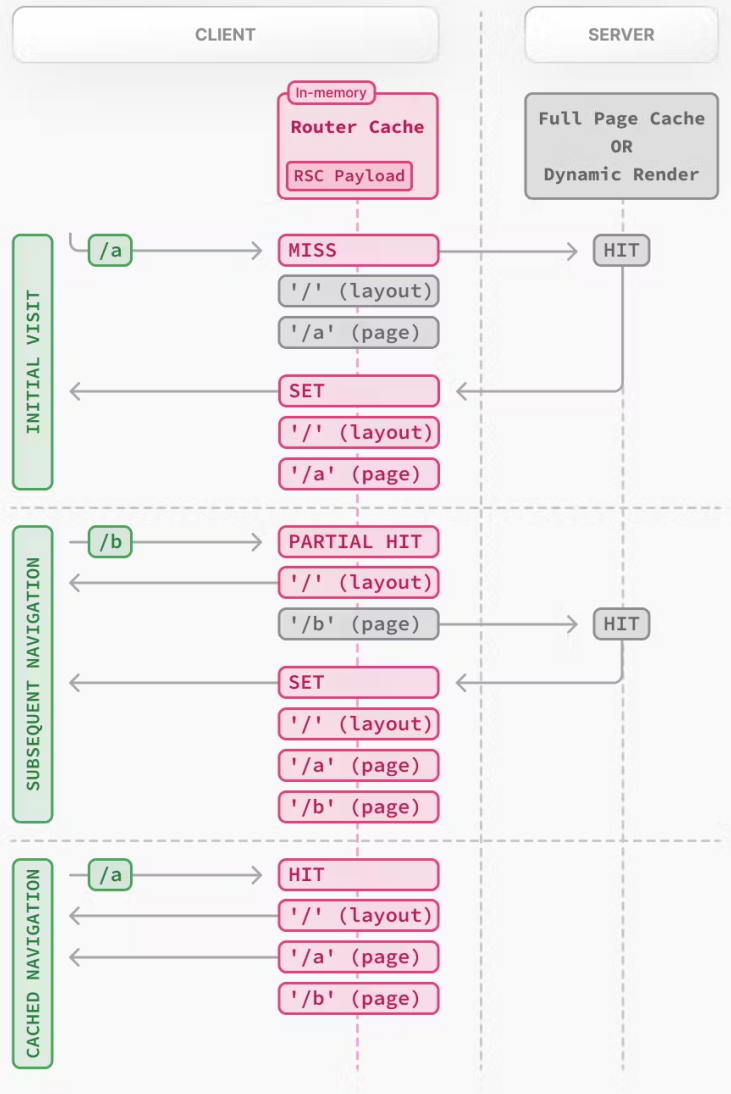
Router Cache

Related Terms:

You may see the Router Cache being referred to as **Client-side Cache** or **Prefetch Cache**. While **Prefetch Cache** refers to the prefetched route segments, **Client-side Cache** refers to the whole Router cache, which includes both visited and prefetched segments. This cache specifically applies to Next.js and Server Components, and is different to the browser's bfcache ↗, though it has a similar result.

Next.js has an in-memory client-side cache that stores the React Server Component Payload, split by individual route segments, for the duration of a user session. This is called the Router Cache.

How the Router Cache Works



As a user navigates between routes, Next.js caches visited route segments and prefetches the routes the user is likely to navigate to (based on `<Link>` components in their viewport).

This results in an improved navigation experience for the user:

- Instant backward/forward navigation because visited routes are cached and fast navigation to new routes because of prefetching and partial rendering.
- No full-page reload between navigations, and React state and browser state are preserved.

Difference between the Router Cache and Full Route Cache:

The Router Cache temporarily stores the React Server Component Payload in the browser for the duration of a user session, whereas the Full Route Cache persistently stores the React Server Component Payload and HTML on the server across multiple user requests.

While the Full Route Cache only caches statically rendered routes, the Router Cache applies to both statically and dynamically rendered routes.

Duration

The cache is stored in the browser's temporary memory. Two factors determine how long the router cache lasts:

- **Session**: The cache persists across navigation. However, it's cleared on page refresh.
- **Automatic Invalidation Period**: The cache of an individual segment is automatically invalidated after a specific time. The duration depends on whether the route is statically or dynamically rendered:
  - **Dynamically Rendered**: 30 seconds
  - **Statically Rendered**: 5 minutes

While a page refresh will clear **all** cached segments, the automatic invalidation period only affects the individual segment from the time it was last accessed or created.

By adding `prefetch={true}` or calling `router.prefetch` for a dynamically rendered route, you can opt into caching for 5 minutes.

Invalidation

There are two ways you can invalidate the Router Cache:

- In a **Server Action**:
  - Revalidating data on-demand by path with (`revalidatePath`) or by cache tag with (`revalidateTag`)
  - Using `cookies.set` or `cookies.delete` invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. authentication).
- Calling `router.refresh` will invalidate the Router Cache and make a new request to the server for the current route.

Opting out

It's not possible to opt out of the Router Cache.

You can opt out of **prefetching** by setting the `prefetch` prop of the `<Link>` component to `false`. However, this will still temporarily store the route segments for 30s to allow instant navigation between nested segments, such as tab bars, or back and forward navigation. Visited routes will still be cached.

Cache Interactions

When configuring the different caching mechanisms, it's important to understand how they interact with each other:

Data Cache and Full Route Cache

- Revalidating or opting out of the Data Cache **will** invalidate the Full Route Cache, as the render output depends on data.
- Invalidating or opting out of the Full Route Cache **does not** affect the Data Cache. You can dynamically render a route that has both cached and uncached data. This is useful when most of your page uses cached data, but you have a few components that rely on data that needs to be fetched at request time. You can dynamically render without worrying about the performance impact of re-fetching all the data.

Data Cache and Client-side Router cache

- Revalidating the Data Cache in a Route Handler **will not** immediately invalidate the Router Cache as the Route Handler isn't tied to a specific route. This means Router Cache will continue to serve the previous payload until a hard refresh, or the automatic invalidation period has elapsed.
- To immediately invalidate the Data Cache and Router cache, you can use `revalidatePath` or `revalidateTag` in a Server Action.

APIs

The following table provides an overview of how different Next.js APIs affect caching:

API	Router Cache	Full Route Cache	Data Cache	React Cache
`<Link prefetch>`	Cache			
`router.prefetch`	Cache			
`router.refresh`	Revalidate			
`fetch`			Cache	Cache
`fetch` `options.cache`			Cache or Opt out	
`fetch` `options.next.revalidate`		Revalidate	Revalidate	
`fetch` `options.next.tags`		Cache	Cache	
`revalidateTag`	Revalidate (Server Action)	Revalidate	Revalidate	
`revalidatePath`	Revalidate (Server Action)	Revalidate	Revalidate	
`const revalidate`		Revalidate or Opt out	Revalidate or Opt out	
`const dynamic`		Cache or Opt out	Cache or Opt out	
`cookies`	Revalidate (Server Action)	Opt out		
`headers`, `useSearchParams`, `searchParams`		Opt out		
`generateStaticParams`		Cache		
`React.cache`				Cache
`unstable_cache`				

`<Link>`

By default, the `<Link>` component automatically prefetches routes from the Full Route Cache and adds the React Server Component Payload to the Router Cache.

To disable prefetching, you can set the `prefetch` prop to `false`. But this will not skip the cache permanently, the route segment will still be cached client-side when the user visits the route.

Learn more about the `<Link>` component.

`router.prefetch`

The `prefetch` option of the `useRouter` hook can be used to manually prefetch a route. This adds the React Server Component Payload to the Router Cache.

See the `useRouter` hook API reference.

`router.refresh`

The `refresh` option of the `useRouter` hook can be used to manually refresh a route. This completely clears the Router Cache, and makes a new request to the server for the current route. `refresh` does not affect the Data or Full Route Cache.

The rendered result will be reconciled on the client while preserving React state and browser state.

See the `useRouter` hook API reference.

`fetch`

Data returned from `fetch` is automatically cached in the Data Cache.

```
1  // Cached by default. `force-cache` is the default option and can be ommitted.
2  fetch(`https://...`, { cache: 'force-cache' })
```

See the `fetch` API Reference for more options.

`fetch options.cache`

You can opt out individual `fetch` requests of data caching by setting the `cache` option to `no-store`:

```
// Opt out of caching
fetch(`https://...`, { cache: 'no-store' })
```

Since the render output depends on data, using `cache: 'no-store'` will also skip the Full Route Cache for the route where the `fetch` request is used. That is, the route will be dynamically rendered every request, but you can still have other cached data requests in the same route.

See the `fetch` API Reference for more options.

`fetch options.next.revalidate`

You can use the `next.revalidate` option of `fetch` to set the revalidation period (in seconds) of an individual `fetch` request. This will revalidate the Data Cache, which in turn will revalidate the Full Route Cache. Fresh data will be fetched, and components will be re-rendered on the server.

```
// Revalidate at most after 1 hour
fetch(`https://...`, { next: { revalidate: 3600 } })
```

See the `fetch` API reference for more options.

`fetch options.next.tags` and `revalidateTag`

Next.js has a cache tagging system for fine-grained data caching and revalidation.

1. When using `fetch` or `unstable_cache`, you have the option to tag cache entries with one or more tags.
2. Then, you can call `revalidateTag` to purge the cache entries associated with that tag.

For example, you can set a tag when fetching data:

```
// Cache data with a tag
fetch(`https://...`, { next: { tags: ['a', 'b', 'c'] } })
```

Then, call `revalidateTag` with a tag to purge the cache entry:

```
// Revalidate entries with a specific tag
revalidateTag('a')
```

There are two places you can use `revalidateTag`, depending on what you're trying to achieve:

1. Route Handlers - to revalidate data in response of a third party event (e.g. webhook). This will not invalidate the Router Cache immediately as the Router Handler isn't tied to a specific route.
2. Server Actions - to revalidate data after a user action (e.g. form submission). This will invalidate the Router Cache for the associated route.

`revalidatePath`

`revalidatePath` allows you manually revalidate data **and** re-render the route segments below a specific path in a single operation. Calling the `revalidatePath` method revalidates the Data Cache, which in turn invalidates the Full Route Cache.

```
revalidatePath('/')
```

There are two places you can use `revalidatePath`, depending on what you're trying to achieve:

1. Route Handlers - to revalidate data in response to a third party event (e.g. webhook).
2. Server Actions - to revalidate data after a user interaction (e.g. form submission, clicking a button).

See the `revalidatePath` API reference for more information.

> **`revalidatePath`** vs. **`router.refresh`**:
>
> Calling `router.refresh` will clear the Router cache, and re-render route segments on the server without invalidating the Data Cache or the Full Route Cache.
>
> The difference is that `revalidatePath` purges the Data Cache and Full Route Cache, whereas `router.refresh()` does not change the Data Cache and Full Route Cache, as it is a client-side API.

Dynamic Functions

`cookies`, `headers`, `useSearchParams`, and `searchParams` are all dynamic functions that depend on runtime incoming request information. Using them will opt a route out of the Full Route Cache, in other words, the route will be dynamically rendered.

`cookies`

Using `cookies.set` or `cookies.delete` in a Server Action invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. to reflect authentication changes).

See the `cookies` API reference.

Segment Config Options

The Route Segment Config options can be used to override the route segment defaults or when you're not able to use the `fetch` API (e.g. database client or 3rd party libraries).

The following Route Segment Config options will opt out of the Data Cache and Full Route Cache:

- `const dynamic = 'force-dynamic'`
- `const revalidate = 0`

See the Route Segment Config documentation for more options.

`generateStaticParams`

For dynamic segments (e.g. `app/blog/[slug]/page.js`), paths provided by `generateStaticParams` are cached in the Full Route Cache at build time. At request time, Next.js will also cache paths that weren't known at build time the first time they're visited.

You can disable caching at request time by using `export const dynamicParams = false` option in a route segment. When this config option is used, only paths provided by `generateStaticParams` will be served, and other routes will 404 or match (in the case of catch-all routes).

See the `generateStaticParams` API reference.

React `cache` function

The React `cache` function allows you to memoize the return value of a function, allowing you to call the same function multiple times while only executing it once.

Since `fetch` requests are automatically memoized, you do not need to wrap it in React `cache`. However, you can use `cache` to manually memoize data requests for use cases when the `fetch` API is not suitable. For example, some database clients, CMS clients, or GraphQL clients.

utils/get-item.ts TypeScript

```ts
import { cache } from 'react'
import db from '@/lib/db'

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

Previous
Edge and Node.js Runtimes

Next
Styling