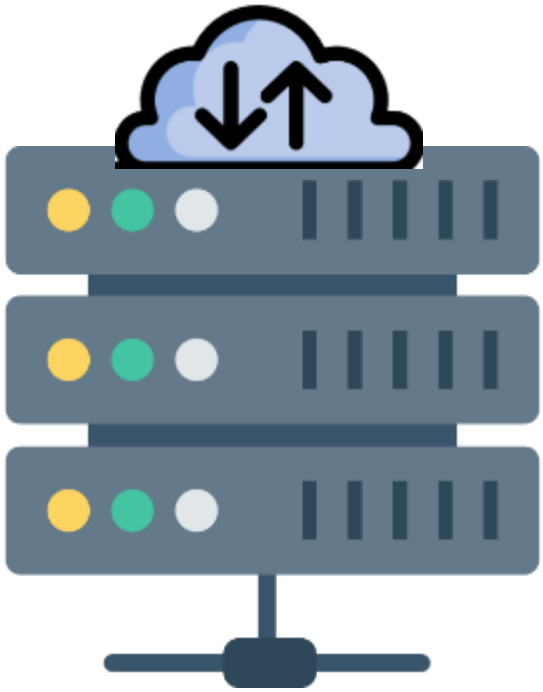
NEXT.JS

Server Actions
Data Fetching & Caching

Server Actions



Server Actions

- Server Actions allow us to create functions that run on the server and can be called directly from pages/components **without needing to create an in-between Web API layer**
  - Simpler alternative to using client-side `fetch` and API routes for data mutations
  - Reduce client-side JavaScript
- Server Actions are not fully-stable yet, so you must opt-in via the **`next.config.js`** file

```
const nextConfig = {
  experimental: {
    serverActions: true,
  },
};
```

Server Actions

- Create a Server Action in a server-side component/page by defining an asynchronous function with the **`"use server"`** directive at the top of the function body

```
async function myAction() {
  "use server";
  ...
}
```

- To invoke a Server Action either:
  - Assign it to a form **`action`** attribute to handle the form submission
  - Pass it to a child client-side component to directly invoke it to handle an event such as button click

Example - Handle Form Submission

```
async function onSubmit(formData) {
  "use server";
  const cat = {
    name: formData.get("title"),
    imageUrl: formData.get("imageUrl"),
    breed: formData.get("breed"),
  };
  await updateCat(catId, cat);
  redirect("/cats");
}

return (
  <div className="center">
    <form action={onSubmit}>
      <input name="id" type="hidden" defaultValue={cat?.id} />
```

When the form is submitted, the **onSubmit** server-side function will be invoked (without using fetch and Web API)

After the update, the user is **redirected** to /cats

Calling Server Action function from a client-side component

```tsx
import { getCats } from "./cat-repo";
import DeleteButton from "./delete-button";
import { onDeleteCat } from "./actions";

export default async function CatsPage() {
  const cats = await getCats();
  return (<div>
      <ul>
        {cats.map((cat) => (
          <li key={cat.id}>
            <a href={`/cats/${cat.id}`}>{cat.name}</a> ({cat.breed})
            <DeleteButton id={cat.id} onDeleteClicked={onDeleteCat} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```tsx
"use client";
export default function DeleteButton({ id, onDeleteClicked }) {
  return (
    <button onClick={async () => {
        if (confirm("Confirm delete?")) onDeleteClicked(id);
      }}
    >
      ❌
    </button>
  );
}
```

Server action function (**onDeleteCat**) is passed from CatsPage to the DeleteButton client-side component. It is called when the delete button is clicked.

Server Actions in actions.js file

- Server Action asynchronous functions could be defined in a separate js file (such as `actions.js`) with the `"use server"` directive at the top of the file

```
"use server";
import { revalidatePath } from "next/cache";
import { likeCat, deleteCat } from "./cat-repo";

export async function onLikeCat(catId) {
  return await likeCat(catId);
}

export async function onDeleteCat(catId) {
  deleteCat(catId);
  revalidatePath("/cats");
}
```

Components can import and call server action functions

- Components (including client-side ones) can **import** and **call** server action functions

```
"use client";
import { onLikeCat } from "./actions";

export default function LikeButton({ catId }) {
  return (
    <button  onClick={async () => {
        await onLikeCat(catId);
      }}
    > Like 👍</button>
  );
}
```

Re-rendering after Data Mutation

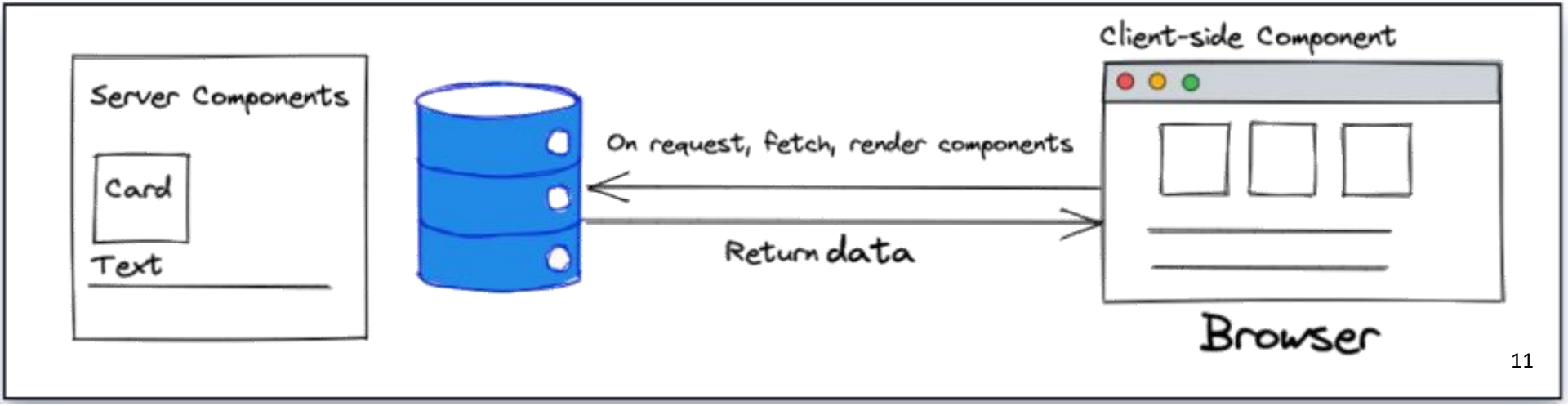
- After data mutation (e.g., handling the form submission to update a cat), you can re-render the UI to `ensure` the correct data is displayed on the client using:
  - **`revalidatePath`** function (from `"next/cache"` library) allows revalidating a Url to refresh the data
    - e.g., after deleting a cat `revalidatePath("/cats")` is called to refresh the list of cats
  - **`redirect`** function (from `"next/navigation"` library) allows redirecting to another page
    - e.g., after adding a cat `redirect("/cats")` is called to redirect to the cats page

Data Fetching & Caching

Back

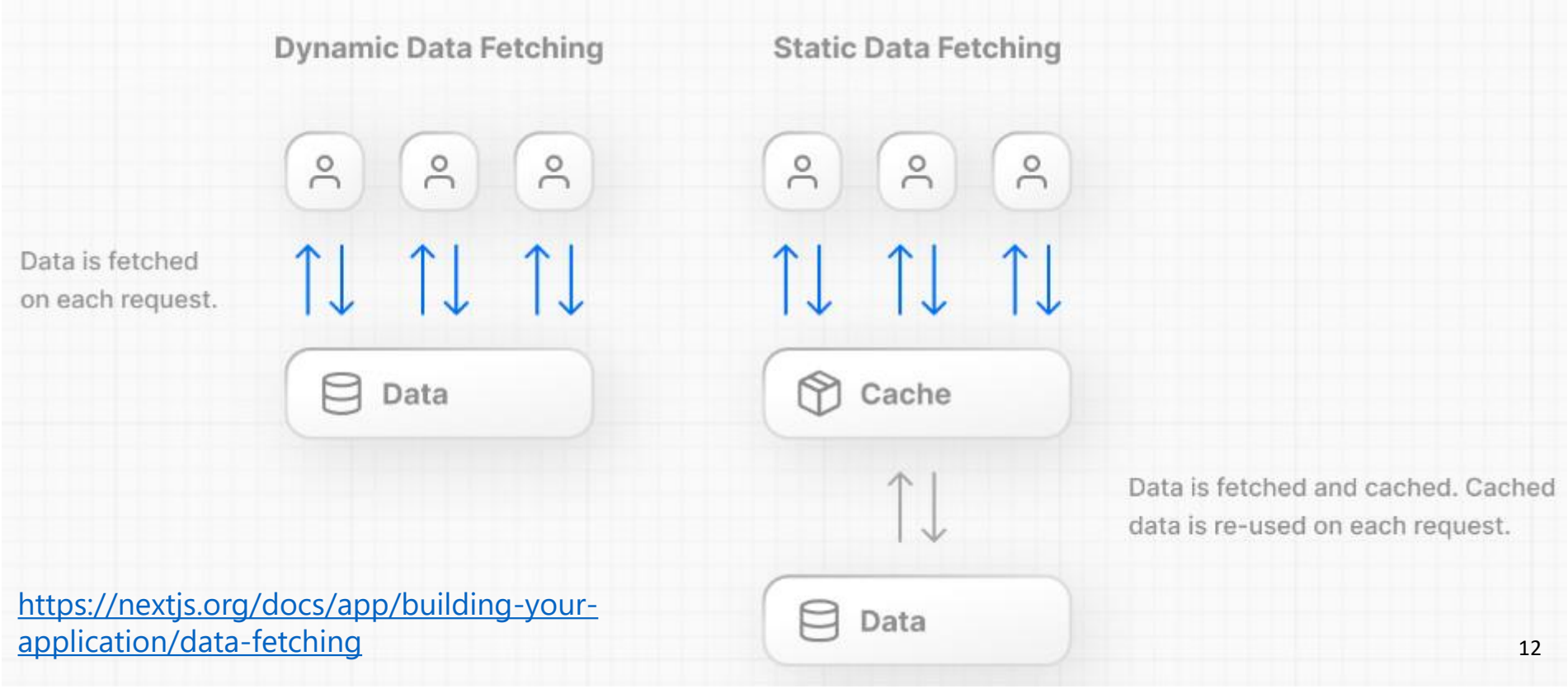
Component-level Client and Server Rendering

- You can choose the rendering environment at the component level
  - **Client-side rendering**: the browser runs the JavaScript to render the interface the user can interact with. It can fetch data from the server using Web API
  - **Server-side rendering**: by default, the app router uses Server Components and renders components on the server but ..
    - **Cannot include interactivity** such as onClick handlers
    - Reduces the amount of JavaScript sent to the client



Static vs. Dynamic Data Fetching

- **Static data fetching** is suitable for static / infrequently changed data e.g., a blog post, CS study plan
- **Dynamic data fetching** is suitable for dynamic / fast changing data (e.g., stock price, breaking news) or data specific to users e.g., a shopping cart list



https://nextjs.org/docs/app/building-your-application/data-fetching

Caching Fetched Data

- **cache** and **next** parameters of the **`fetch`** function can be used to set the caching strategy

```
// This request should be cached until manually invalidated.
// `force-cache` is the default and can be omitted.
fetch(URL, { cache: 'force-cache' });

// This request should be refetched on every request.
fetch(URL, { cache: 'no-store' });

// This request should be cached with a lifetime of 10 seconds.
fetch(URL, { next: { revalidate: 10 } });
```

To see these in action you need to:

- Build the app using: **`npm run build`**
- Start the built app using: **`npm run start`**

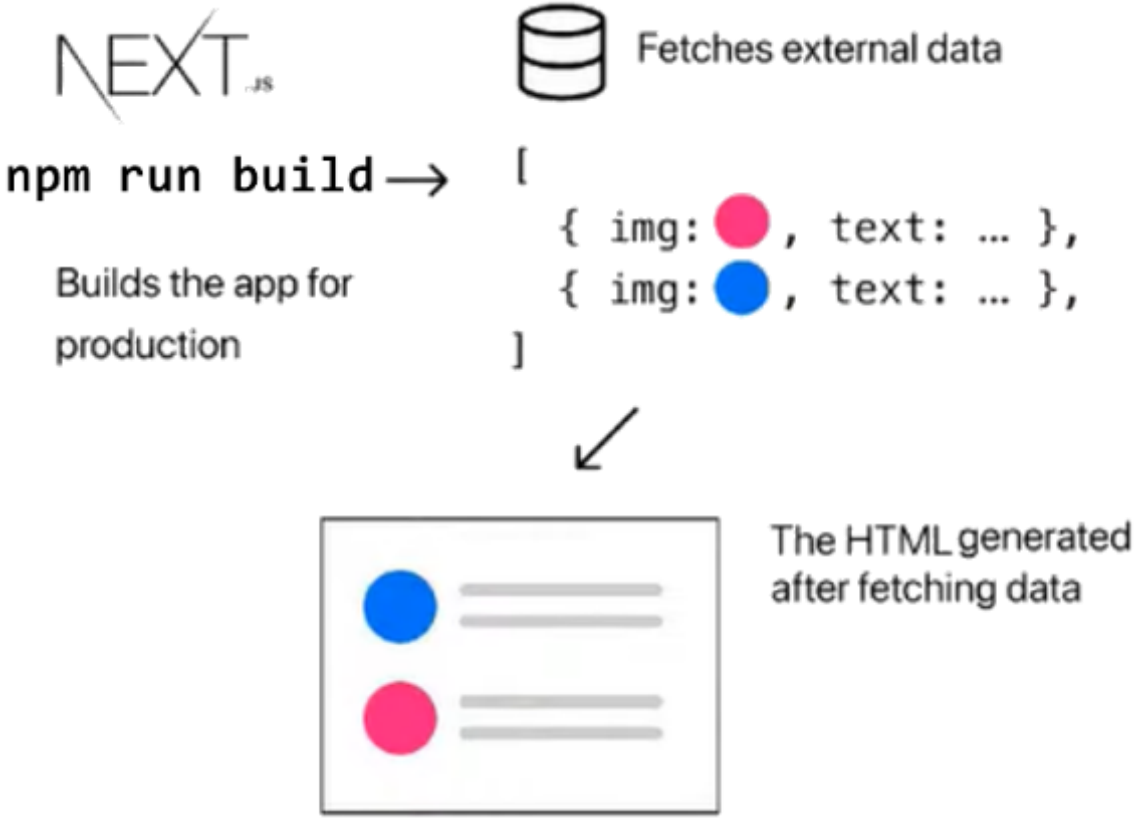
Static Data Fetching

- By default, Next.js automatically does static data fetching. This means that the data will be fetched at build time, cached, and reused on each request
- There are two benefits to using static data fetching:
  - The data is automatically cached for improved loading performance
  - It reduces the load on the database by minimizing the number of requests made and reduces the computation load on the server

```
fetch('https://...'); // cache: 'force-cache' is the default
```

Static Rendering

By default, Next.js automatically does static data fetching then uses the fetched data to statically render the pages at **build time**



```
async function getNavItems() {
  const navItems = await fetch('https://api.example.com/...');
  return navItems.json();
}

export default async function Layout({ children }) {
  const navItems = await getNavItems();

  return (
    <>
      <nav>
        <ul>
          {navItems.map((item) => (
            <li key={item.id}>
              <Link href={item.href}>{item.name}</Link>
            </li>
          ))}
        </ul>
      </nav>
      {children}
    </>
  );
}
```

Static Rendering Example

Statically Generate Pages with Dynamic Routes

`generateStaticParams` function can be used in combination with dynamic route segments to define the list of route segment parameters that will be used for statically generating dynamic route pages at **build time**

```
export default function Page({ params }) {
  const { slug } = params;

  return ...
}

export async function generateStaticParams() {
  const posts = await getPosts();

  return posts.map((post) => ({
    slug: post.slug,
  }));
}
```

Dynamic Data Fetching

- Re-fetch data on every fetch() request using the `cache: 'no-store'` option

```
fetch('https://...', { cache: 'no-store' });
```

- Components are rendered on the server at request time. The result of the work is **not cached**

Revalidating Data

- Revalidation is the process of purging the cache and re-fetching the latest data
  - This ensures that the app shows the latest **data changes** without having to rebuild the entire app

Next.js provides two types of revalidation:

- **Background**: Revalidates the data at a specific time interval (**but ONLY if someone visits the page after the configured time interval has passed**)
- **On-demand**: Revalidates the data whenever there is an update

Background Revalidation

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

```
fetch('https://...',
    { next: { revalidate: 60 } })
```

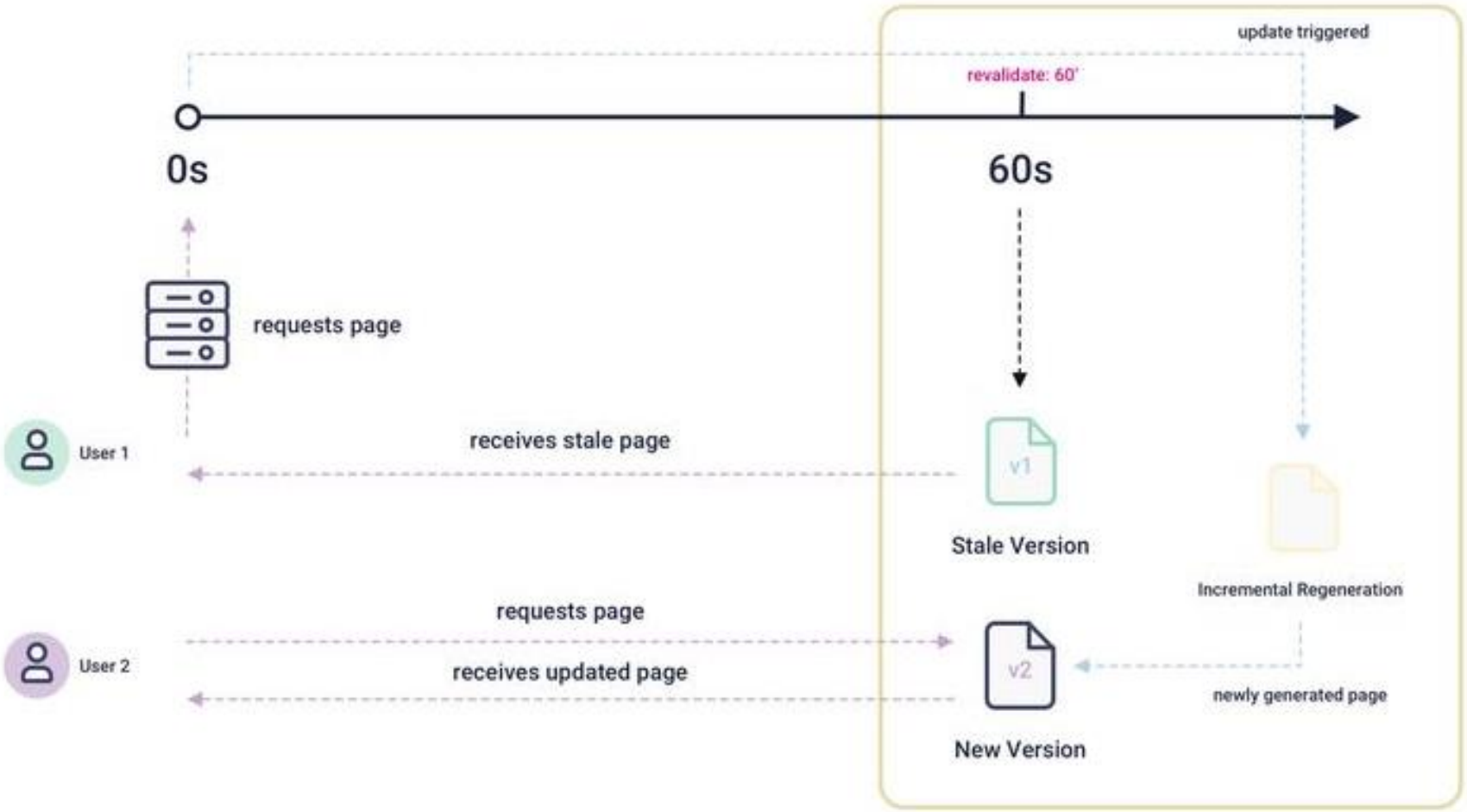
- To revalidate data that does not use fetch (e.g., getting data using a repo function that uses Prisma) simply export a **revalidate** value

```
// revalidate this page every 60 seconds
export const revalidate = 60
```

Background Revalidation - How it works?

- When a request is made to the route that was statically rendered at build time, it will initially show the cached data
- Any subsequent requests made to the route within the specified time frame, prior to the expiration of the revalidation interval (e.g., 60 seconds), will be fulfilled using the cached data
- After the revalidation interval (e.g., 60 seconds), the next request will still show the cached (stale) data
  - Next.js will trigger a regeneration of the data in the background
  - Once the data generates successfully, Next.js will invalidate the cache and use the updated data

Background Revalidation



On-Demand Revalidation

- **`revalidatePath`** allows you to revalidate data associated with a specific path. It allows updating the cached data without waiting for a revalidation period to expire

JS app/api/revalidate/route.js

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

export async function GET(request) {
  const path = request.nextUrl.searchParams.get('path') || '/';
  revalidatePath(path);
  return Response.json({ revalidated: true, now: Date.now() });
}
```

Summary

- Server Actions allow us to create functions that run on the server and can be called directly from pages/components **without needing to create an in-between Web API layer**

- Next.js supports different data fetching and caching strategies: **Static Data Fetching**, **Dynamic Data Fetching**, and configurable **Data Revalidation** (Background or On-demand)

Resources

- Server Actions

  https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions

- Data Fetching & Caching

  https://nextjs.org/docs/app/building-your-application/data-fetching