How to use server actions in Next.js

Offloading work from your client to your server is easy with Next.js server actions. Here's everything you need to know about server actions in Next.js.

Offloading work from your client to your server is easy with **Next.js** server actions . Here's everything you need to know about **server actions in** Next.js.

What is Server Action?

Server Actions or server actions allow you to write one-off server-side functionality right alongside your server components. This is important because you no longer need to write API routes when submitting forms or making any other type of data changes, including GraphQL data changes.

You can write functions that run on the server and then call them from client or server components. This is an alpha feature in Next.js 13.4, built on top of React Actions. Using server actions results in reduced client-side JavaScript and can help you gradually create advanced forms.

Example of server action

With server action, you can make changes inside Next.js, on the server. Let's look at this new feature with an example page created with Next.js that displays a form that allows you to create posts.

These are the imports:

```
import Link from "next/link"
import FormGroup from "@/components/FormGroup"
import { revalidateTag } from "next/cache"
```

```
import { redirect } from "next/navigation"
```

Now with the post creation code. This function is a server action. It runs on the server and sends the post title and content to the API (creates the post in the database):

```
async function createPost(data) {
  "use server"
  const title = data.get("title")
  const body = data.get("body")

  await fetch("http://127.0.0.1/posts", {
    header: {"Content-Type": "application/json"},
    method: POST,
    body: JSON.stringify({title, body})
  })

  revalidateTag("posts")
  redirect("/")
}
```

This function takes the post title and content, then sends it to the **/posts** endpoint via a POST request. It then forces the cache to refresh the content associated with the "post" tag and redirects back to the home page.

Here is the form that collects new post titles & content:

```
export default NewPostForm() {
  return (
    form action={createPost}
      div
        FormGroup
          label htmlFor="title" Title label
          input type="text" name="title" id="title"
        FormGroup
      div

      div
        FormGroup
          label htmlFor="title"Bodylabel
          textarea type="text" name="body" id="body" textarea
        FormGroup
      div
      div
        button Save button
      div
    form
  )
}
```

You can see that the form contains no logic related to creating a post, just a form action: the **createPost()** function . Recall that the **createPost** function has a "use server" directive declaring that everything in that function will run on the server.

All the code is running on the server and your client knows nothing about it. This is because the form is a server component and nothing is dynamically displayed on the client.

When the Save button is clicked , Next.js will call **the createPost()** function . This sends the title and body as form data. It will be posted to the local API to save the information.

Verify the post again

At the bottom of the **createPost()** function is the following line of code:

```
revalidateTags("posts")
```

To understand what the function means, imagine that you have a page of posts. Inside the posts component, you'll call a function called **getPosts()** to get all the posts from the database and display them for the user to see:

```
export default async function Posts(){
  const posts = await getPosts()

  return (
    div
      {posts.map(post => {
        PostCard key={post.id} {.post}
      })}
    div
  )
}
```

The getPosts() function looks like this:

```
async function getPosts() {
  const res = await fetch("https://127.0.0.1/posts?sort=title", {
    next: {tags: ["posts"], revalidate: 3600 }
  })

  return res.json()
}
```

This function passes the next option to fetch, allowing you to set the revalidation interval. By default, every single fetch request you make inside the server component will be cached permanently. Monitoring the cache and refreshing it when necessary is an important part of the data fetching process in Next.js.

The above code tells Next.js to store **post** data for up to one hour (3,600 seconds). After an hour, that data becomes stale and Next.js "refreshes" it to get the latest data from the database.

Remember that the **createPost()** function calls **revalidateTag("posts")** after it has completed the remaining work. This forces Next.js to re-fetch data including the new post the app just created.

To make sure all the code is running on the server, you can add the following console log command in the **createPost()** function :

```
console.log("Running on the server")
```

Then try creating a new post by clicking the submit button. If your code is running on the server, that log message will show up on your terminal. But if it is running on the client then the log will show up in the browser

console.

Above is how to use server actions in Next.JS. Hope the article is useful to you.

You finished reading the article "**How to use server actions in Next.js**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.
