

Wir sind eine zertifizierte B Corporation.



HOME ► WISSENS-HUB ► BLOG

DE | EN

# // Server Actions in Next.js 14

Webdevelopment

React

JavaScript



Lukas Lehmann

10.6.2024 | 9 minutes of reading time

Server Actions were introduced in Next.js 14 as a new method to send data to the server (see the [documentation](#)). They are asynchronous functions that can be used in server components, within server-side forms, as well as in client-side components. While the invocation of a Server Action appears as a normal function call in the code, it is interpreted as POST request to the server by Next.js.

In this blog post, I demonstrate in simple examples how Server Actions can be used and what we have to consider when using them.

## Server Actions: an example

As an example, we implement a simple web application that can be used to save users and also search them. The search should use a search text and react to client-side changes in the input by loading the data from the server.

In the following, I will only show the important code sections. You can find the entire source code of the example application here:

share post



Likes



0

## Adding and editing users

We define a `UserPage` to add and edit users. The page is rendered server-side and loads the existing user directly from the database if an ID was given via the URL path. This user is then passed to the `UserForm` component, which can be used to edit the user, respectively add a new user if we did not pass any ID.

```
1 // app/user/[[...id]]/page.tsx
2
3 import { getUserFromDb } from '@app/db';
4 import { UserForm } from '@app/components/UserForm'
5
6 export default async function UserPage({ params }: {
7   const id = params.id?.at(0);
8   const user = id ? await getUserFromDb(id) : undefi
9
10   return <>
11     <h1>User</h1>
12     <UserForm user={user}/>
13   </>;
14 }
```

The `UserForm` component is implemented as a client component.\* It contains the `firstName` and `lastName` as states, which are set when we write into the respective inputs. On clicking the save button, the Server Action `saveUser` is called, passing it the `firstName`, `lastName`, and possibly the `id`, if it exists. After that, we route back to the root URL.

```
1 // app/components/UserForm.tsx
2
3 'use client';
4
5 import { useRouter } from 'next/navigation';
6 import { saveUser } from '@app/actions/saveUser';
7 import { useState } from 'react';
8 import { User } from '@app/user';
9
10 export function UserForm({ user }: { user: User | un
11   const id = user?.id;
```

```

12
13     const [firstName, setFirstName] = useState(user?.f
14     const [lastName, setLastName] = useState(user?.las
15
16     const router = useRouter();
17
18     return <>
19         <div>
20             <label>First Name</label>
21             <input
22                 type="text"
23                 value={firstName}
24                 onChange={(e) => setFirstName(e.target.value
25             />
26         </div>
27         <div>
28             <label>Last Name</label>
29             <input
30                 type="text"
31                 value={lastName}
32                 onChange={(e) => setLastName(e.target.value)
33             />
34         </div>
35         <button onClick={async () => {
36             await saveUser({ id, firstName, lastName });
37             router.push('/');
38         }}>
39             Save
40         </button>
41     </>;
42 }

```

Q

The Server Action `saveUser` is a simple asynchronous function. It has to be noted that we have to declare it with `'use server'`, in order to ensure that Next.js is able to identify it as a Server Action. The Server Action writes the given user into the database. If it does not yet have an ID (i.e., if it is a newly added user), a random ID is generated first. Finally, the cache is invalidated using `revalidatePath('/')`. Thereby, we make sure that the root page will be freshly rendered on the next load, containing the updated users.

```

1 // app/actions/saveUser.ts
2
3 'use server';
4
5 import { User } from '@app/user';

```

```
6 import { randomUUID } from 'crypto';
7 import { putUserIntoDb } from '@app/db';
8 import { revalidatePath } from 'next/cache';
9
10 export async function saveUser(user: User) {
11   if (!user.id) {
12     user.id = randomUUID();
13   }
14   await putUserIntoDb(user);
15   revalidatePath('/');
16 }
```


A circular button with a magnifying glass icon inside, used for searching.

The `saveUser` example shows us how we can use Server Actions to send data to the server. Compared to Route Handlers, Server Actions have the advantage that they provide type safety during compile time. When using Route Handlers, it can easily happen that, for instance, our request body on the client side does not match the expected request body on the server side. The static type checking on Server Actions prevents this error.

However, it is important to note that Server Actions, just as Route Handlers, are implemented as HTTP endpoints under the hood. This means that if a user has access to the frontend of the application, they will also have access to the POST request, which is sent to the server when the Server Action is invoked, as well as the response that is received from the server. A user could use this information to send an invalid request body to the server. Hence, we still require an input validation in practice, especially for publicly available applications, as TypeScript's static typing will not prevent such scenarios. Furthermore, the Server Action needs to check for authentication and authorization as well.

## User search

We have seen that we can use Server Actions to send data to the server. This raises the question if we can use Server Actions to read data from the server as well. In fact, this is technically possible, as Server Actions are able to return a response to the client. I would first like to demonstrate how we can implement a live search using a Server Action and then explain why this is not that great of an idea in practice, and provide a better alternative.

A simple, empty rectangular box with a thin border, likely a placeholder for a diagram or additional content.

The `SearchPage` is implemented as a server-side component that initially loads all users directly from the database. As we make use of server-side rendering here, we require neither Server Actions nor Route Handlers to fetch the users from the server at this point.

Q

```
1 // app/page.tsx
2
3 import { UserSearch } from '@app/components/UserSea
4 import { getAllUsersFromDb } from '@app/db';
5
6 export default async function SearchPage() {
7   const users = await getAllUsersFromDb();
8
9   return <>
10     <h1>User Search</h1>
11     <UserSearch users={users}/>
12   </>;
13 }
```

The `SearchPage` passes the initially loaded users to the `UserSearch` component, which is rendered on the client side. The `UserSearch` holds the list of users as state. Via an input field, one can search for users. As soon as the input is changed, the Server Action `getUsers` is invoked and the result is written to the state via `setUsers`. Below the search field, the current list of users is printed.

```
1 // app/components/UserSearch.tsx
2
3 'use client';
4
5 import { User } from '@app/user';
6 import { useState } from 'react';
7 import { getUsers } from '@app/actions/getUsers';
8 import Link from 'next/link';
9
10 export function UserSearch(props: { users: User[] })
11   const [users, setUsers] = useState(props.users);
12
13   return <>
14     <input
15       type="text"
16       placeholder="Search user..."
17       onChange={async (e) => {
```

```

18         const users = await getUsers(e.target.value)
19         setUsers(users);
20     }}
21 />
22 <ul>
23     {users.map((user, index) => (
24         <li key={index}>
25             <Link href={`/user/${user.id}`}>{user.firs
26         </li>
27     ))}
28 </ul>
29 <Link href="/user">
30     <button>New User</button>
31 </Link>
32 </>;
33 }

```

Q

The Server Action `getUsers` is an asynchronous function, declared as Server Action via `'use server'`, returning a `Promise<User[]>` as result. First, `getUsers` loads all users from the database. Then, it filters the users using the given `searchTerm`, returning only the users whose `firstName` or `lastName` contain the `searchTerm`.

```

1 // app/actions/getUsers.ts
2
3 'use server';
4
5 import { User } from '@app/user';
6 import { getAllUsersFromDb } from '@app/db';
7
8 export async function getUsers(searchTerm: string):
9     const users = await getAllUsersFromDb();
10    return users.filter(user => user.firstName.include
11 }

```

## A better alternative without Server Actions

We have seen that we can load the list of users from the server using the Server Action `getUsers` whenever a client-side change on the search input occurs. However, this approach has a decisive drawback: the Server Action is implemented as POST request. As we are reading data from the server, we would actually expect a GET request instead. In practice, this limitation to POST

requests has the disadvantage that requests are not cached. This means that on each call of `getUsers` with the same search term the server is actually invoked, instead of loading existing data from the cache on recurring invocations.

Q

Fortunately, there is a simple way to implement live search without a Server Action (and without a Route Handler as well), namely using a search parameter that we pass to the `SearchPage`. The `SearchPage` uses server-side rendering and loads the users directly from the database. Then, the list of users is filtered using the `searchTerm`, which is given via the search parameter `q` in the URL.

```
1 // app/page.tsx
2
3 import { UserSearch } from '@app/components/UserSea
4 import { getAllUsersFromDb } from '@app/db';
5
6 export default async function SearchPage({ searchPar
7   searchParams: { [key: string]: string | string[] |
8 }) {
9   const searchTerm = getSearchTerm(searchParams);
10
11   const users = (await getAllUsersFromDb())
12     .filter(user => user.firstName.includes(searchTe
13
14   return <>
15     <h1>User Search</h1>
16     <UserSearch users={users}/>
17   </>;
18 }
19
20 function getSearchTerm(searchParams: { [key: string]
21   const searchTerm = searchParams?.q;
22   if (typeof searchTerm !== 'string') return '';
23   return searchTerm;
24 }
```

The `UserSearch` component is changed with regard to the `onChange` handler of the input field, which no longer calls the Server Action `getUsers`. Instead, the router is used to add the input search term as search parameter `q` to the URL.

```

1  'use client';
2
3  import { User } from '@app/user';
4  import Link from 'next/link';
5  import { useRouter } from 'next/navigation';
6
7  export function UserSearch(props: { users: User[] })
8    const router = useRouter();
9
10   return <>
11     <input
12       type="text"
13       placeholder="Search user..."
14       onChange={async (e) => {
15         router.push(`?q=${e.target.value}`);
16       }}
17     />
18     <ul>
19       {props.users.map((user, index) => (
20         <li key={index}>
21           <Link href={`/user/${user.id}`}>{user.firs
22         </li>
23       ))}
24     </ul>
25     <Link href="/user">
26       <button>New User</button>
27     </Link>
28   </>;
29 }

```

Q

On every character that the user enters into the search field, a fetch request is executed, using the GET method and the new search term as parameter. A fetch request has the advantage that it does not lead to a full page reload, but instead only loads the data on the page, which leads to a better user experience. Furthermore, it uses the client-side Router Cache of Next.js. The Router Cache prevents the same fetch request from being executed multiple times when we enter the same search term several times in a row. In this way, we are able to reduce the number of requests actually sent to the server.

## Conclusion

Server Actions in Next.js 14 offer an interesting option to process data on the



server while ensuring static type safety. Based on an example, we have seen how we can implement a web application with database access in Next.js completely without using Route Handlers. In the code, Server Actions appear as normal function invocations, which may improve readability of the code.

However, we also learned that Server Actions are not a one-size-fits-all solution to implement the entire communication between client and server, even though we can technically use them in that way. For reading data, server-side rendering is a better option, as it allows us to make optimal use of caching mechanisms. Furthermore, when using Server Actions, we must also take measures such as authorization and validation to ensure that our applications run securely and correctly.

---

*\* In this concrete example, it would indeed be possible to implement **UserForm** as a server component, which uses **saveUser** as a submit action. In practice however, we often would like to make use of client-side validation, which shows potential errors directly on a user input and therefore can only work with client-side rendering.*

Was this post helpful?

Ja

<❤>  
0



## Blog author



**Lukas Lehmann**

*Backend Developer*



Do you still have questions? Just send me a message.

**Get in contact**

## More articles

from Lukas Lehmann





## Answer questions about your documents with OpenAI and Pinecone

In recent years, large language models (LLMs) have made remarkable progress in interacting with humans, showcasing their ability to answer a wide array of questions. Trained on publicly accessible internet content, these models have broad knowledge across...

AI

📅 13.11.2023 | 12 Minuten Lesezeit



Lukas Lehmann

## Macro annotations in

In a previous blog post v annotations in Scala 2, v present for a while. Only added to Scala 3 as well release version 3.3.0-RC Same as...

Scala

📅 4.4.2023 | 9 Minuten Lesezeit



Lukas Lehmann

// Your job at  
codecentric?



Jobs

Agile Developer und Consultant



/r

📍 Alle Standorte

Backend

Frontend

Fullstack

View Job ▶

// More articles in this subject area





## Angular 17 – Eine echte Renaissance?

Gefühlt war es lange still rund um das Frontend-Framework Angular. Echte Innovationen blieben aus und man konnte das Gefühl nicht loswerden, dass Vue.js und React mit all ihren Derivaten den Vorsprung zu Angular uneinholbar weit ausbauen. Doch mit Version...

Angular

Webdevelopment

Frontend

 15.12.2023 | 8 Minuten Lesezeit




Stephan Köninger

## Immersive Web statt

Nachdem der Hype der I mit der Umbenennung von Spitze erreicht und viele Brainstorms zum Thema ist nicht mehr viel übrig Die ambitionierten ...

AR/VR

React

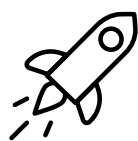
 23.6.2023 | 5 Minuten Lesezeit



Alexander Bruckmann

// Gemeinsam bessere Projekte umsetzen.





**Wir helfen deinem  
Unternehmen.**

Du stehst vor einer großen IT-Herausforderung? Wir sorgen für eine maßgeschneiderte Unterstützung. Informiere dich jetzt.

**Unsere Leistungen** ▶



**Hilf uns, noch besser zu  
werden.**

Wir sind immer auf der Suche nach neuen Talenten. Auch für dich ist die passende Stelle dabei.

**Zu den Jobangeboten** ▶

@codecentric



[Unternehmen ▼](#)

[codecentric für dich ▼](#)

[Sitemap ▼](#)

[Forschung & Entwicklung](#)

