**Infinite Scroll with Next.js Server Actions: A Simple Guide**

[[Simon Ferlat](https://medium.com/@ferlat.simon?source=post_page-----76a894824cfd--------------------------------)](https://medium.com/@ferlat.simon?source=post_page-----76a894824cfd--------------------------------)

[Simon Ferlat](https://medium.com/@ferlat.simon?source=post_page-----76a894824cfd--------------------------------)

·

Follow

5 min read

·

Jan 1

33

When handling large data volumes, effective front-end pagination is crucial for maintaining performance and readability, especially on slower devices. To add a modern touch to pagination, we’ll explore infinite scroll, and how to leverage the power of Next.js [server actions](https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations).

**Step 1 — Initialize the Project**

You can find the completed demo project [here](https://github.com/RockyStrongo/infinite-scroll-example). To create a new Next.js project, run the following command and follow the instructions.

npx create-next-app@latest

For this example, I used TypeScript, Tailwind, and Next.js App Router.

Our requirements are :

* Display a list of users from the free fake API [sling academy](https://www.slingacademy.com/article/sample-users-free-fake-api-for-practicing-prototyping/)
* Load statically at build time the first 10 users
* Load 10 more users as the users scrolls to the bottom of the page

**Step 2 — Display the Initial Users**

**Fetch Users Function**

Start by creating a function to fetch users. Create a file called actions/getUsers.ts

//actions/getUsers.ts  
  
'use server'  
import { UserAPIResponse } from '@/types/User'  
  
export const getUsers = async (offset: number, limit: number) => {  
 try {  
 const url = `https://api.slingacademy.com/v1/sample-data/users?offset=${offset}&limit=${limit}`  
 const response = await fetch(url)  
 const data = (await response.json()) as UserAPIResponse  
 return data.users  
 } catch (error: unknown) {  
 console.log(error)  
 throw new Error(`An error happened: ${error}`)  
 }  
}

**User UI Component**

Create a simple card component to display user data in src/components/UserCard.tsx

import { User } from '@/types/User'  
  
type UserProps = {  
 user: User  
}  
  
export default function UserCard({ user }: UserProps) {  
 return (  
 <div className='bg-indigo-400 text-white rounded w-96 p-3'>  
 <div>{user.id}</div>  
 <div>{user.first\_name}</div>  
 <div>{user.last\_name}</div>  
 <div>{user.phone}</div>  
 <div>{user.email}</div>  
 </div>  
 )  
}

**User Page**

Replace the default Next.js homepage in src/app/page.tsx

import UserCard from '@/components/UserCard'  
import { getUsers } from '@/actions/getUsers'  
  
const INITIAL\_NUMBER\_OF\_USERS = 10  
  
export default async function Home() {  
 const initialUsers = await getUsers(0, INITIAL\_NUMBER\_OF\_USERS)  
  
 return (  
 <div className='flex flex-col gap-3'>  
 {initialUsers.map((user) => (  
 <UserCard key={user.id} user={user} />  
 ))}  
 </div>  
 )  
}

**Step 3 — Load More Users with Infinite Scroll**

**Move user list to a child client component**

Move the contents of our user page into a child client component. Create src/components/UserList.tsx and include it in src/app/page.tsx:

// components/UserList.tsx  
import { User } from '@/types/User'  
import UserCard from './UserCard'  
  
type UserListProps = {  
 initialUsers: User[]  
}  
  
export default function UserList({ initialUsers }: UserListProps) {  
 return (  
 <div className='flex flex-col gap-3'>  
 {initialUsers.map((user) => (  
 <UserCard key={user.id} user={user} />  
 ))}  
 </div>  
 )  
}

//app/page.tsx  
import UserList from '@/components/UserList'  
import { getUsers } from '@/actions/getUsers'  
  
const INITIAL\_NUMBER\_OF\_USERS = 10  
  
export default async function Home() {  
 const initialUsers = await getUsers(0, INITIAL\_NUMBER\_OF\_USERS)  
  
 return <UserList initialUsers={initialUsers} />  
}

**Move Users to a State**

In the user list component, the list of users is now a React state, the initial value is set to the 10 users statically loaded from in the parent page.

// components/UserList.tsx  
  
import { User } from '@/types/User'  
import UserCard from './UserCard'  
import { useState } from 'react'  
  
type UserListProps = {  
 initialUsers: User[]  
}  
  
export default function UserList({ initialUsers }: UserListProps) {  
 const [users, setUsers] = useState<User[]>(initialUsers)  
  
 return (  
 <div className='flex flex-col gap-3'>  
 {users.map((user) => (  
 <UserCard key={user.id} user={user} />  
 ))}  
 </div>  
 )  
}

**Create Offset State and Function to Load More Users**

In the code snippet below, I set up a system to keep track of the data offset, initially set to 10. This offset increases by 10 each time the user requests more data. Additionally, I’ve defined a function that retrieves additional users based on the current offset and adds them to the user state.

// components/UserList.tsx  
  
'use client'  
import { User } from '@/types/User'  
import UserCard from './UserCard'  
import { useState } from 'react'  
import { getUsers } from '@/actions/getUsers'  
  
type UserListProps = {  
 initialUsers: User[]  
}  
  
const NUMBER\_OF\_USERS\_TO\_FETCH = 10  
  
export default function UserList({ initialUsers }: UserListProps) {  
 const [offset, setOffset] = useState(NUMBER\_OF\_USERS\_TO\_FETCH)  
 const [users, setUsers] = useState<User[]>(initialUsers)  
  
 const loadMoreUsers = async () => {  
 const apiUsers = await getUsers(offset, NUMBER\_OF\_USERS\_TO\_FETCH)  
 setUsers([...users, ...apiUsers])  
 setOffset(offset + NUMBER\_OF\_USERS\_TO\_FETCH)  
 }  
  
 return (  
 <div className='flex flex-col gap-3'>  
 {users.map((user) => (  
 <UserCard key={user.id} user={user} />  
 ))}  
 <button onClick={loadMoreUsers}>Load more</button>  
 </div>  
 )  
}

🥳 We now have a fully functional pagination feature activated by a button to load more users.

Since the “getUsers” function is a [NextJS server action](https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations), it runs on the server rather than the client side. When a user loads more users for the first time, the information is fetched from the API and cached. Subsequent requests for more data by other users won’t require additional API calls, as the data will already be stored in the NextJS cache! This significantly enhances page performance and user experience. Moreover, this can reduce your hosting costs, as less back-end calls are needed to achieve the same result.

**Load More Users at Scroll**

The finishing touch for our project involves detecting user scrolling and activating the loadMoreUsers function accordingly.

To achieve this, we’ll install the react-intersection-observer library. This library simplifies the process of monitoring whether a component is within the viewport.

npm install react-intersection-observer

// components/UserList.tsx  
  
'use client'  
import { User } from '@/types/User'  
import UserCard from './UserCard'  
import { useEffect, useState } from 'react'  
import { getUsers } from '@/actions/getUsers'  
import { useInView } from 'react-intersection-observer'  
  
type UserListProps = {  
 initialUsers: User[]  
}  
  
const NUMBER\_OF\_USERS\_TO\_FETCH = 10  
  
export default function UserList({ initialUsers }: UserListProps) {  
 const [offset, setOffset] = useState(NUMBER\_OF\_USERS\_TO\_FETCH)  
 const [users, setUsers] = useState<User[]>(initialUsers)  
 const { ref, inView } = useInView()  
  
 const loadMoreUsers = async () => {  
 const apiUsers = await getUsers(offset, NUMBER\_OF\_USERS\_TO\_FETCH)  
 setUsers([...users, ...apiUsers])  
 setOffset(offset + NUMBER\_OF\_USERS\_TO\_FETCH)  
 }  
  
 useEffect(() => {  
 if (inView) {  
 loadMoreUsers()  
 }  
 }, [inView])  
  
 return (  
 <div className='flex flex-col gap-3'>  
 {users.map((user) => (  
 <UserCard key={user.id} user={user} />  
 ))}  
 <div ref={ref}>  
 Loading...  
 </div>  
 {/\* <button onClick={loadMoreUsers}>Load more</button> \*/}  
 </div>  
 )  
}

Congratulations! We have built a super user-friendly infinite scroll feature that not only makes our application more modern but also boosts its performance. By using Next.js server actions and caching, we’ve fine-tuned data retrieval to make sure everything runs smoothly, creating an all-around positive experience for your users! 🚀