

Contents

Next.js	1
Folder Structure	1
Root	1
User Page	2
Advantages	2
Structure	2
Layout	2
Page	3
Loading	4
use client vs. use server	4
Syntax	5
useState Hook	5
Suspense	6
Styling	7
Usage	7
Conditional Styling	7
Environment	7

Next.js

Folder Structure

To create a page, add a page file inside the app directory and default export a React component. Folders are used to define the route segments that map to URL segments. Files (like page and layout) are used to create UI that is shown for a segment.

Root

- lib and ui have **no framework meaning** for Next (no page.tsx)

path	features
./app	root dir for react
./app/page.tsx	starting point (/ route)
./app/layout.tsx	root layout
./app/not-found.tsx	404 page
./app/lib/	type definitions, REST server client code, server side code
./app/ui/	React components for the user interface
./app/public/	static resources, here: images (png files)
./app/user/	/user page

path	features
<code>./app/global-error.tsx</code>	error handling

User Page

path	features
<code>./app/user/layout.tsx</code>	user page layout
<code>./app/user/loading.tsx</code>	fallback UI (loading screen) upon navigation
<code>./app/user/not-found.tsx</code>	custom 404 page for user
<code>./app/user/error.tsx</code>	error handling
<code>./app/user/[id]</code>	dynamic subpages
<code>./app/user/(overview)/loading.tsx</code>	decouple loading page from subdirectories
<code>./app/user/(overview)/page.tsx</code>	decouple loading page from subdirectories

Advantages

- SEO improvements through **SSR** (Server Side Rendering) and **SSG** (Static Site Generation)
- Automatic **Code Splitting** (Chunking)
- Simplified *file-based* routing
- Easy full-stack development through API routes
- **Scoped CSS** and **SASS** support
- **TypeScript** support
- Image optimization
- **HMR** (Hot Module Replacement)
- Page pre-fetch
- Minimal configuration on Vercel

Structure

Each route is reflected in the directories and files under “app” and must have at least a “page.tsx” file which defines a Page component creating the content of the (sub)page.

Layout

`layout.tsx` defines an optional **Layout component** creating the layout of page. The result of the Page component is passed to the Layout component through a `children` prop.

- **Partial rendering:** when the user navigates to a certain path, only the Page components are rerendered, not the layouts
- A layout is UI that is shared between multiple pages.
- On navigation, layouts preserve state, remain interactive, and do not rerender
- You can define a layout by default exporting a React component from a layout file.
- The component should accept a children prop which can be a page or another layout
- Nesting layouts is possible

```
import '@app/global.css'; // : import global css file to top level component

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body className={`_${inter.className} antialiased`} >{children}</body>
    </html>
    // The layout above is called a root layout because it's defined at the root of the
    // app directory.
    // The root layout is required and must contain html and body tags.
  );
}
```

Page

- A nested route is a route composed of multiple URL segments.
- For example, the `/details/[id]/edit` route is composed of four segments:
 - `/` - the root segment
 - `/details` - the details segment
 - `/[id]` - the dynamic id segment (slug)
 - `/edit` - the edit segment

```
// 'use server';

interface EditListProps {
  params: {
    id: string;
  };
}

// async component
export default async function EditDetails({ params }: { params: { id: string } }) {
```

```

    console.log('invoice id = ' + params.id);
    const id = params.id;
    const friend = JSON.parse(await getFriend(Number(id)));
    console.log('friend = ' + JSON.stringify(friend));

    return (
      <></>
    );
  };

  // alternative props with interface
  function EditDetailsWithInterface({ params }: EditListProps) {
};

```

Loading

Loading generates code to display while component Page ist not finished generating output.

This approach is called **streaming**. We display a **skeleton** of the final layout while data is being loaded. See files `loading.tsx` and `ui/skeletons.ts`

`loading.tsx` defines component Loading:

```

import DashboardSkeleton from "../ui/skeletons";

export default function Loading() {
  // simple version:
  //return <div>Loading...</div>;

  // more complex version: show dashboard skeleton
  return <DashboardSkeleton />;
}

```

- Has to be in the same folder as the `page.tsx` it is masking

use client vs. use server

use client Forced to run **client side**

For: - Hooks: `useState`, `useEffect`, `useContext` - Event listeners - API: `localStorage`, `window`, `document`

```

'use client';

import { useState } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

```

```

    return (
      <button onClick={() => setCount(count + 1)}>
        Count: {count}
      </button>
    );
  }
}

```

use server Explicitly a **server side function**

For: - Fetching data from **database** - Handling **server side actions** (form submission...) - **Secure operations** (authentication...) - **Node APIs** (fs, process.env, ...)

```

'use server';

import { revalidatePath } from 'next/cache';

export async function saveToDatabase(data) {
  await db.collection('users').insertOne(data);
  revalidatePath('/dashboard'); // Refresh data on the page
}

```

Syntax

useActionState Hook

- `const [state, formAction, isPending] = useActionState(fn, initialState, permalink?);`
- **fn**: **callback** for form submission or button press
- **initialState**: any **serializable** value, ignored after first invocation
- **useActionState** returns an **array** with the following values:
 1. **current state** (**initialState** for first render) -> after invocation *value returned by action*
 2. **new action** (can be passed as **action** prop to **form** component or **formAction** prop to any **button** component within the form, can also be *called manually* within **startTransition**)
 3. **isPending** flag for *pending Transition*

lib/auth.ts

```

export async function authenticate(
  prevState: string | undefined,
  formData: FormData,
) {
  try {
    await signIn('credentials', formData);
  }
}

```

```

    } catch (error) {
      if (error instanceof AuthError) {
        switch (error.type) {
          case 'CredentialsSignin':
            return 'Invalid credentials.';
          default:
            return 'Something went wrong.';
        }
      }
      throw error;
    }
  }
}

```

component.tsx

```

const [errorMessage, formAction, isPending] = useActionState(
  authenticate,
  undefined,
);

return (
  <form action={formAction}>
    <input name="email" type="email" placeholder="Email" required/>
    <input name="password" type="password" placeholder="Password" required/>

    {errorMessage && <p className="text-red-500">{errorMessage}</p>}

    <button type="submit" disabled={isPending}>
      {isPending ? 'Logging in...' : 'Login'}
    </button>
  </form>
);

```

Suspense

<Suspense> wraps async component (e.g. **fetch** data) -> fallback UI (e.g. skeleton, spinner) while waiting

Benefits: - **Streaming Server Rendering** - Progressively rendering HTML from the server to the client. - **Selective Hydration** - React prioritizes what components to make interactive first based on user interaction.

```

import { Suspense } from 'react';

return(
  <Suspense fallback={<p>Loading feed...</p>}>
    <PostFeed />
  </Suspense>
);

```

```
    </Suspense>
  )
```

Styling

- Global CSS files (imported in `app/layout.tsx`)
- Site specific CSS files

Usage

```
import styles from '@app/ui/home.module.css';

export default function Page() {
  return (
    <div className={styles.shape}/> // <- style.className
  )
}
```

Conditional Styling

Conditional className Strings with **CLSX**

Sometimes the CSS classes for an HTML element are dynamic depending on some state. With `clsx()` you can construct a dynamic list of strings in an easy to read way.

```
<div
  key={invoice.id}
  className={clsx('flex flex-row items-center justify-between py-4',
    { 'border-t': i !== 0 })}
/>
```

The DIV element above has fixed classes (`flex ... py-4`) and class “border-t” only if `i !== 0` holds.

Environment

`.env`:

```
DATABASE_URL=postgres://user:password@localhost:5432/mydatabase
JWT_SECRET=my-super-secret-key
...
```