Next.js 14 Advanced and Next 15 and React 19

Next 14

- using Next.js we can write the react code for the UI and the API that can be called by the react code.
- npm run dev for development server and npm run build, npm run start production server

Dynamic Routes

- Dynamic segments are passed as params prop to **layout**, **page**, **route**, or **generateMetadata**
- We can use generateStaticParamas() function to generate all the possible dynamic slugs and return them when we build our application, so the dynamic pages will be prerendered during **build time** instead of on Demand at request time.

for **app/[slug]/page.tsx**

```
export function generateStaticParams() {
return [{ id: '1' }, { id: '2' }, { id: '3' }]
}
// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/1
// - /product/2
// - /product/3
export default function Page({ params }: { params: { id: string } }) {
const { id } = params
// ...
}
```

for catch all segment: **app/[...slug]/page.**

```
export function generateStaticParams() {
return [{ slug: ['a', '1'] }, { slug: ['b', '2'] }, { slug: ['c', '3'] }]
}
// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default function Page({ params }: { params: { slug: string[] } }) {
const { slug } = params
// ...
}
```

for multiple Dynamic segment: **app/[category]/[product]/page.**

```
export function generateStaticParams() {
return [
{ category: 'a', product: '1' },
{ category: 'b', product: '2' },
{ category: 'c', product: '3' },
]
}
// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /products/a/1
// - /products/b/2
// - /products/c/3
export default function Page({
params,
}: {
params: { category: string; product: string }
}) {
const { category, product } = params
// ...
}
```

we can also generate params from **top down**.
for app/[category].page.

```
// Generate segments for [category]
export async function generateStaticParams() {
const products = await fetch('https://.../products').then((res) => res.json())
return products.map((product) => ({
category: product.category.slug,
}))
}
export default function Layout({ params }: { params: { category: string } }) {
// ...
}
```

then for **app/[category]/[product]/page.**

```
// Generate segments for [product] using the `params` passed from
// the parent segment's `generateStaticParams` function
export async function generateStaticParams({
params: { category },
}: {
params: { category: string }
}) {
const products = await fetch(
`https://.../products?category=${category}`
).then((res) => res.json())
return products.map((product) => ({
```

```
product: product.id,
}))
}
export default function Page({
params,
}: {
params: { category: string; product: string }
}) {
// ...
}
```

- the difference between **catch-all-segment and optional catch-all-segment** is that the optional catch-all also consider that we might not even have a dynamic route which means the **parent route** is also matched.
- if we use fetch() Api inside **generateStaticParams()** the request will be **cached**, it won't be duplicated again.

Routing Metadata

- super important for SEO ranking.
- we use metadata API to export metadata for a specific route either from **layout.tsx or page.tsx.**
- we can either export static metadata object or dynamic using **generateMetadata** function.
- if there is multiple metadata for the same route, it will be **combined(Merged)** but the **page** metadata will replace the layout.
- metadata can only be exported from a **server components.**

generateMetadata

- parameters: **props**: just an object containing the parameter of the current route. like params and searchParams.
- **params**: object containing all the dynamic params.
- **searchParams**: object containing current URL searchParams.
- it should return a **Metadata** object with on or more metadata fields.
- it can also be async if we use **fetch()** API in which the result will be **memoized** for the same data in generateStaticParams, layouts, page, server components...

Fields

- title: can be string or object with {default, template and absolute} properties which are **strings**.
- **default** will be user as fallback for child segments incase they don't have a metadata, **template** will be used to add prefix and suffix to child segments. {example value for template: '%s | suffix'}, and **absolute** will be used in child segment to overwrite the default.
- other basic fields are [description, applicationName, keywords: string[], author: {}[], creator, publisher..]

File based metadata

- File based metadata has the higher priority and override any config based metadata.
- the special files are: [favicon.ico, icon.jpg, opengraph-image.jpg, twitter-image.jpg, robot.txt, sitemap.xml] can be created inside App directory.

- opengraph-image is image that allows a website to provide a snippet of information when shared on social medias. it makes a shared link look better on social media by showing a preview of the website.
- sitemaps are machine readable formats that tells search engines about website pages, videos.....

Parallel Route

- parallel routes are created using named slots. Slots are defined with @folder convention.
- Slots are passed to the parent shared Layout along with the children.
- the layout.tsx will accept the slots by their @folder name alongside the children prop and render them in parallel.
- Slots are not route segments and don't affect the URL structure.
- one of the benefit of using parallel route is independent route handling: which means each slots will have their own loading state and error state, so we can display them only on the specific slot, which doesn't affect the whole thing.
- the other benefit is sub-navigation: in which each slots are like mini-application with its own navigation and state management.
- including the children Prop, all the slots when being sub navigated only the active one will be a matched route, the others including children will be unmatched.
- so if there is no default.tsx file for the unmatched routes, when the page reloads the unmatched will be 404 pages including children.
- so we need to create default.tsx file for all slots including children, and mirror the UI.

Intercepting Route

- can be used like, for feed when a user click on a photo or for login modal where the URL is changed to the exact route but we are displaying the intercepting route without leaving the context of the first route.
- the concept is to display the intercepted route but when on page refresh displaying the actual route.
- so its like an additional route for a specific route but to display without leaving the context and if the page reloads the original full route will be displayed.
- the convention is [(.)folder for same level, (..)folder one level above, (..)(..)folder for 2 above and (...)folder from the root app.]

Route Handlers

- used to create custom request handlers for our routes.
- allows us to create Restful endpoint, also great for external API request.
- simple as exporting async function with one of the allowed HTTP method as its name, with response.

```
export async function GET() {
const res = await fetch('https://data.mongodb-api.com/...', {
headers: {
'Content-Type': 'application/json',
'API-Key': process.env.DATA_API_KEY,
},
})
```

```
const data = await res.json()
return Response.json({ data })
}
```

- Headers are metadata about the request and response API.
- Cookies are small piece of data a server send to a client, the client may store it or resend it back to the server with request.

Rendering

- using Suspense is necessary when using server side rendering as it will help improve the user experience by not waiting for the whole page and loading some parts and wait for other part.
- trying to stay server and when necessary using client components is effective for performance
- Static Rendering is when HTML is rendered at build time. and in Next 14 static rendering is the default rendering strategy.]
- when using dynamic functions like Cookies, Header, searchParams Next 14 will automatically convert it into dynamic rendering.
- import 'server-only' and import 'client-only' to make sure it won't work incase there is a mistake.

Next 15

- Async Request API: API's that are dependent on Request-Specific Data are Asynchronous. Like Header, Cookies, Params and SerachParams.

```
import { cookies } from 'next/headers';
export async function AdminPanel() {
const cookieStore = await cookies();
const token = cookieStore.get('token');
// ...
}
```

- Get Route handler are not by default cached, but it can be cached by changing the config into export dynamic = 'force static
- Client routes are not by default cached except shared layout, loading.ts..... meaning every time we navigate the routes will be dynamic and display a fresh data.
- Next 15 display static route indicator, to show either the route is static or dynamic.
- Form component: improve the normal HTML form element with Prefetching, client side navigation and Progressive enhancement. it is useful for forms that navigate to the result page after completion.
  when the form element is in view, the layout and loading file will be prefetched. for smoother navigation.
  even if there is no JavaScript loaded, the form will still work.

```
import Form from 'next/form';
export default function Page() {
return (
```

```
<Form action="/search">
<input name="query" />
<button type="submit">Submit</button>
</Form>
);
}
```

- Enhanced Security for server actions: either remove unused server actions or create secure ID for reference by client.
- Server components are by default re-executed when saved which means any fetch request will be re-called. but to save any billed API calls there is HMR(hot module replacement) that can reuse fetch responses from previous renders.
- next dev --turbo (turbopack) for better developer experience when building the project.

React 19

- Actions: common use case is to implement data mutation and then update a state in response.
- useActionState: which accept server action and initial state and returns {state, formAction, and isPending}
- useFormStatus: to destructure the pending state and use it display.
- we can pass ref as a props for functional component.
- useOptimistic to display optimistic data while the request is going on.