Next.js v13 Cheatsheet

This cheatsheet aims to provide a quick reference guide to some of the most significant additions and updates in Next.js v13.

Creating a new Next.js 13 app

```
npx create-next-app@latest my-app
```

It may ask some questions like the name of the app, the directory, and the type of app you want to create.

Navigate to the directory and run the app.

```
cd my-app
code .
```

Start up the development server.

```
npm run dev
```

App Directory

`app/` is a directory that contains global styles, components, and other files that are used throughout the application.

```
/app
  /api
    /hello
      /route.js
  /components
    /Navbar
      /Navbar.js
      /Navbar.module.css
  /layout.js
  /page.js
  /page.module.css
  /about
    /page.js
    /page.module.css
```

page.js

`page.js` is the main page of the app, somewhat similar to `index.js` in Next.js v12.

page.module.css

Next.js v13 uses CSS Modules by default, so you can use CSS Modules like `page.module.css`.

Routing

Earlier we had a `pages` directory which had all the pages with the component name as the page route.

Now, we create pages inside the `app` directory. We do that by first creating a directory with the route name and then creating a `page.js` file inside it.

For example, if we want to create a page with the route `/about`, we create a directory named `about` inside the `app` directory and then create a `page.js` file inside it.

```
/app
  /about
    page.js
  /contact
    page.js
  page.js
```

Nested Routes

Also, if we want to create a page with the route `/about/team`, we create a directory named `team` inside the `about` directory and then create a `page.js` file inside it.

```
/app
  /about
    /team
      page.js
    page.js
  /contact
    page.js
  page.js
```

This way we also don't need to create any other components directory for the different components that we want to use in the pages. We can directly create those component in the `/about` directory.

Links

Earlier we used to import the `Link` component from `next/link` and then use it like this:

```
<Link href="/about">
  <a className="link">About</a>
</Link>
```

Now, we can directly use the `Link` component like this:

```
<Link href="/about" className="link">
  About
</Link>
```

Layouts

Earlier we used to create a `Layout` component and then wrap the pages with it like this:

```
import Layout from '../components/Layout';

export default function Home() {
  return (
    <Layout>
      <h1>Hello World</h1>
    </Layout>
  );
}
```

Now, we have separate layout components for any page we want. These are named `layout.js` or `layout.jsx`. They basically wrap the whole page as the children of the layout component.

They take `children` as a prop and then render it inside the layout component.

```
const AboutLayout = ({ children }) => {
  return (
    <div>
      <h1>THIS IS ABOUT PAGE LAYOUT</h1>
      {children}
    </div>
  );
};

export default AboutLayout;
```

We can create as many layouts for different pages as we want. Each parent layout will be rendered inside the child layout.

This allows us to have some custom layouts, for example, we can have a modal for the `/about` page and a sidebar for the `/about/team` page.

MetaData

Instead of having a `Head` component like before, we can create the meta tags attributes inside the `layout.js` or the `page.js` for any page.

```
export const metadata = {
  title: 'Yodkwtf Academy',
  description: 'Learn to code with Yodkwtf Academy',
};

export default function Home() {
  return (
    <div>
      <h1>Home page</h1>
    </div>
  );
}
```

If it's in the page component, it'll only show up for that page. If it's in the layout component, it'll show up for all the pages that use that layout, basically all the nested pages will have those meta tags too.

Importing Fonts

Instead of importing fonts via css, we can directly import them into the components.

```
import { Poppins } from 'next/font/google';

const poppins = Poppins({
  weight: ['400', '600', '700'],
  styles: ['italic'],
  subsets: ['latin-ext'],
});
```

This will give us a `poppins` variable which we can use in the `className` attribute of the component.

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={poppins.className}>{children}</body>
    </html>
  );
}
```

Server Components

In NextJS 13, your components are server-rendered by default. This means, if you want to use the useState hook for example and make it interactive, you need to make it a client component otherwise you'll get an error. We can do that simply by adding **use client** to the top of the file.

```
// app/components/Navbar.js

'use client';
```

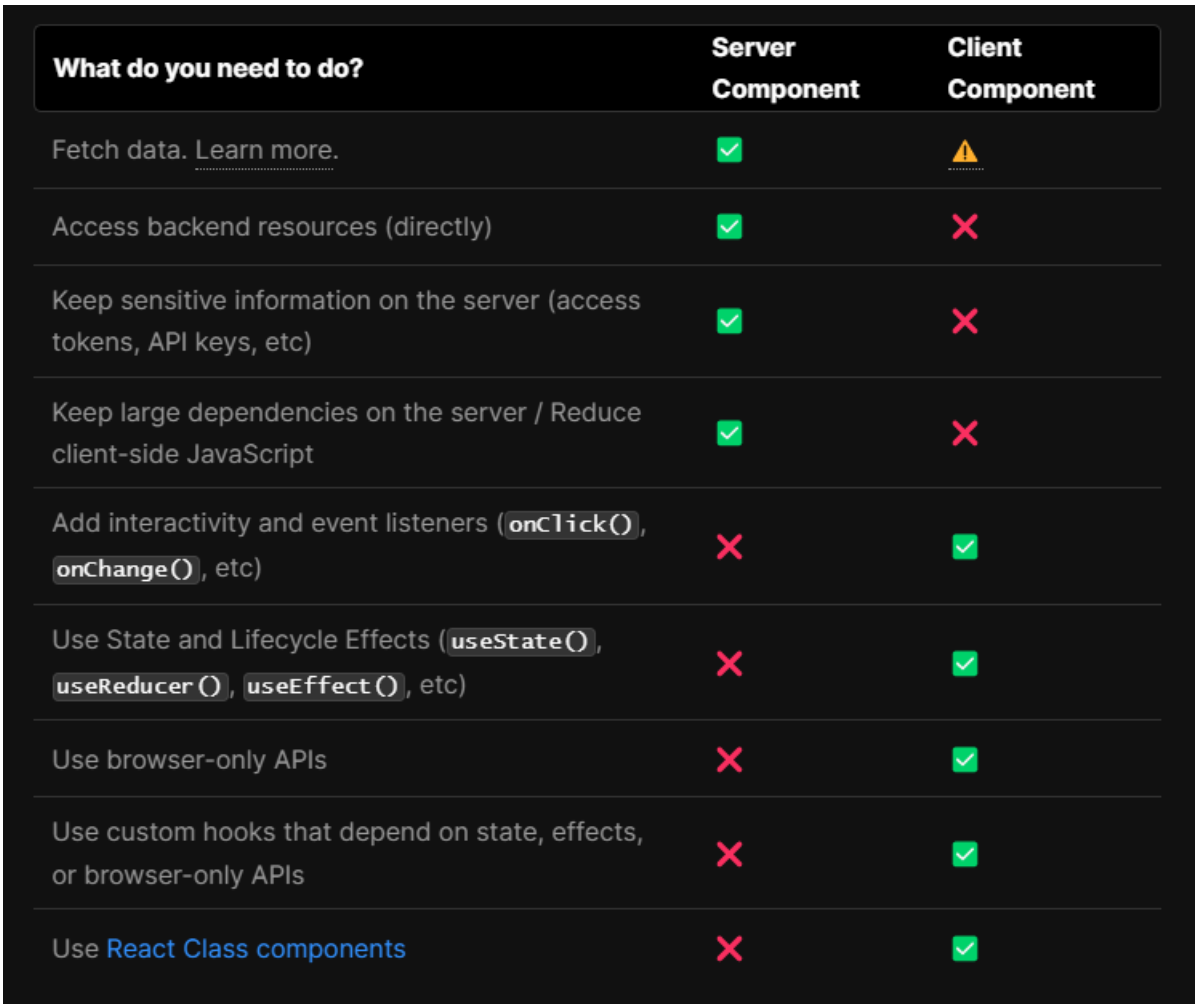
Advantages of RSC:

- Load faster - Don't have to wait for the JavaScript to load
- Smaller client bundle size
- SEO friendly
- Access to resources the client can't access
- Hide sensitive data from the client
- More secure against XSS attacks
- Improved developer experience

Just like with anything else, there are also disadvantages:

- Not as interactive
- No component state. We can not use the `useState` hook.
- No component lifecycle methods. We can not use the `useEffect` hook.

Here is a chart from the NextJS website that shows when to use a server component vs a client component.



What do you need to do?	Server Component	Client Component
Fetch data. Learn more.	✅	⚠️
Access backend resources (directly)	✅	❌
Keep sensitive information on the server (access tokens, API keys, etc)	✅	❌
Keep large dependencies on the server / Reduce client-side JavaScript	✅	❌
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	❌	✅
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	❌	✅
Use browser-only APIs	❌	✅
Use custom hooks that depend on state, effects, or browser-only APIs	❌	✅
Use React Class components	❌	✅

Data Fetching

Earlier we used to use the `getStaticProps` and `getServerSideProps` functions to fetch data from the server. We also had to work with `useEffect` and dependency arrays to fetch data from the server, which at times could be a bit confusing.

Now fetching data in RSC (react server components) is much simpler. We can use an async function to make the api call similar to like we did in vanilla javascript.

```
// app/repos/page.js
const fetchRepos = async () => {
  const response = await fetch(
    'https://api.github.com/users/Yodkwtf-Academy/repos'
  );
  const repos = await response.json();
  return repos;
};
```

We can then use the data returned from the function in our async functional component.

```
// app/repos/page.js
export default async function Repos() {
  const repos = await fetchRepos();

  return (
    <div>
      <h1>Repos</h1>
      <ul>
        {repos.map((repo) => (
          <li key={repo.id}>{repo.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

> Note: Any console.log statements from the server components will not show up in the browser. They will only show up in the terminal.

Custom Loading/Error Page

If we are fetching data in a server component and it takes some time to load, we can show a custom loading page. We can do that just by creating a `loading.js` or `loading.jsx` file inside the `app` directory. It doesn't require any conditional rendering or anything. It'll automatically show up when the page is loading.

```
// app/loading.js
const LoadingPage = () => {
  return (
    <div className="loader">
      <div className="spinner"></div>
    </div>
  );
};

export default LoadingPage;
```

The component name can be anything but the file name should be `loading.js` only.

We can also do the same for the error page. We can create a `error.js` or `error.jsx` file inside the `app` directory and it'll automatically show up when there is an error.

```
// app/error.js

const ErrorPage = () => {
  return (
    <div>
      <h1>Something went wrong</h1>
    </div>
  );
};
```

Dynamic Routes

Create a directory based on how nested or where you want the dynamic route to be. For example, if you want the dynamic route to be in the `/products/id` page, create a directory named `products` inside the `app` directory and then create another directory within the `products` directory named `[id]`.

Get Identifiers from the URL

We can get the identifiers from the URL by passing `params` as the props

```
const SingleProduct = ({ params }) => {
  return (
    <div className="card">
      <h2>{product.id}</h2>
    </div>
  );
};
```

If you choose `name` or `title` instead of `id` in the dynamic route, you can access it using `params.name` or `params.title`.

Suspense Boundaries

Suspense boundaries are used to show a fallback component when the data is being fetched from the server. We can use it to show a loading component or an error component for some specific component.

For example, we have a product page where the name and description show up immediately but the images show up after the data is fetched from the server. We can use suspense boundaries to show a specific placeholder while the images are being fetched.

```
import { Suspense } from 'react';

export default function SingleProduct({ params }) {
  return (
    <>
      <h2>{product.id}</h2>
      <Suspense fallback={<div>Loading...</div>}>
        <img src={product.image} alt={product.name} />
      </Suspense>
    </>
  );
}
```

The fallback component will show up until the data is fetched from the server.

The benefit of using this is that even if the images take a long time to load, the name and description will show up immediately instead of the whole page loading.

Catching and Revalidating

In the previous versions of NextJS, we had to use the `getStaticProps` and `getServerSideProps` functions to fetch data from the server.

Now, the pages are cached by default in the production build. It is great for performance but it can also cause some issues if the data is changing very often (think about news sites). For example, if we have a page that shows the current time, it will show the same time for all the users. If we want to show the current time for each user, we need to revalidate the page.

We can change this by adding some options with the `fetch` functions.

Revalidate

It tells NextJS how often it should check for new data. It takes a number in seconds after which it should look for new data.

```
const response = await fetch(
  'https://api.github.com/users/Yodkwtf-Academy/repos',
  {
    next: {
      revalidate: 60,
    },
  }
);
```

Basically, it'll cache the data for 60 seconds and then it'll check and fetch new data.

> If we don't provide the revalidate option, the data is fetched and cached by Next.js every time we build a new version of our application and hence it'll only update the data when the site is rebuilt.

This is only needed if the data is changing very often. If the data is not changing very often, we can just leave it as it is.

If we want to revalidate the page every time the user visits the page, we can set the revalidate option to `0`.

```
const response = await fetch(
  'https://api.github.com/users/Yodkwtf-Academy/repos',
  {
    next: {
      revalidate: 0,
    },
  }
);
```

This can also be done by adding a different option in the `fetch` function.

```
const response = await fetch(
  'https://api.github.com/users/Yodkwtf-Academy/repos',
  {
    cache: 'no-store',
  }
);
```

API Route Handlers

We can create API routes in NextJS using the `app/api` directory. We can create a file inside the `api` directory and it'll automatically create an API route.

We can also do it in any page directory but the common practice is to create it in the `api` directory since it prefixes the api route with `/api`.

Structure

The structure of the API route is similar to the structure of the page. We create a directory with the route's name inside the `api` directory and then create a file called `route.js` inside the directory.

We name the function whatever HTTP method we want to use.

```
// app/api/videos/route.js

import { NextResponse } from 'next/server';
import videos from './data.json'; // dummy data

export async function GET(request) {
  return NextResponse.json(videos);
}
```

They can be used the same way as any external API.

```
const fetchVideos = async () => {
  const res = await fetch('http://localhost:3000/api/videos');
  const videos = await res.json();
  return videos;
};

const Videos = async () => {
  const videos = await fetchVideos();
  // ...
};
```

Query Params

Get the query params from the URL by following the below steps.

1. Get the URL from the request object

```
export async function GET(request) {
  console.log(request.url);
  // returns /api/videos/search?query=react
}
```

2. Create a new URL object using the URL constructor

```
const urlData = new URL(request.url);
console.log(urlData);
// returns URL { href: '/api/videos/search?query=react', origin: 'http://localhost:3000', protocol: 'http:', username: '', password:
```

3. Get the search params from the URL object

```
const searchParams = urlData.searchParams;
console.log(searchParams);
// returns URLSearchParams { 'query' => 'react' }
```

4. Get the query param from the search params

```
const query = searchParams.get('query');
console.log(query);
// returns react
```

5. Use the query param to filter the data

```
const filteredVideos = videos.filter((video) => {
  return video.title.toLowerCase().includes(query.toLowerCase());
});
return NextResponse.json(filteredVideos);
```

Getting Body Data from POST Requests

We can get the body data from the POST requests from the `request.json()` function.

```
export async function POST(request) {
  const body = await request.json();
  console.log(body);
  // returns { title: 'React Hooks Course', description: 'React Hooks are awesome' }
}
```

Create a new video using the body data.

```
export async function POST(request) {
  const body = await request.json();
  const newVideo = {
    id: videos.length + 1,
    title: body.title,
    description: body.description,
  };
  videos.push(newVideo);
  return NextResponse.json(videos); // return the updated videos
}
```

Client Components

Client components are components that are only rendered on the client side. They are not rendered on the server-side. They are used when we want to add any interactive features to the page.

By default, all Next.js 13 components are server components. If we want to use a client component, we need to add the line `use client` at the top of the component.

```
// app/components/VideoPlayer.js

'use client';

const VideoPlayer = () => {
  return (
    <div>
      <video controls>
        <source src="https://www.youtube.com/watch?v=6ZnG9vBw8K4" />
      </video>
    </div>
  );
};
```

We can use hooks like `useState` and `useEffect` only in the client components.

Outro

Thank you for visiting this cheatsheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.