**Next Js 14**

Next.js is a React framework for building full-stack web applications and to build interactive, dynamic, and fast React applications.

**Main Features:**

**Routing:** A file-system based router built on top of Server Components that supports layouts, nested routing, loading states, error handling, and more.

**Rendering**: Client-side and Server-side Rendering with Client and Server Components. Further optimized with Static and Dynamic Rendering on the server with Next.js. Streaming on Edge and Node.js runtimes.

**Data Fetching**: Simplified data fetching with async/await in Server Components, and an extended fetch API for request memoization, data caching and revalidation.

**Styling** : Support for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS etc.

**Optimizations:** Image, Fonts, and Script Optimizations to improve your application's Core Web Vitals and User Experience.

**TypeScript:** Improved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.

**App Router vs Pages Router**

Next.js has two different routers: the **App Router(**useRouter() hook imported from next/router for routing and usePathname() hook from next/navigation to know current pathname**)** and the **Pages Router(file-system based routing).** The App Router is a newer router that allows you to use React's latest features, such as Server Components and Streaming. The Pages Router is the original Next.js router, which allowed you to build server rendered. React applications and continues to be supported for older Next.js applications.

**Installation**: **npx create-next-app@latest (is create-next-app is CLI)**

**Folder Str: A screenshot of a phone

Description automatically generated**

|  |  |
| --- | --- |
| A screenshot of a computer program  Description automatically generated | /app: Contains all the routes, components, and logic for your application, this is where you'll be mostly working from.  /app/lib: Contains functions used in your application, such as reusable utility functions and data fetching functions.  /app/ui: Contains all the UI components for your application, such as cards, tables, and forms. To save time, we've pre-styled these components for you.  /public: Contains all the static assets for your application, such as images.  /scripts/: Contains a file that you'll use to populate your database in a later chapter.  Config Files: You'll also notice config files such as next.config.js at the root of your application. Most of these files are created and pre-configured when you start a new project using create-next-app. You will not need to modify them in this course. |

/app/layout.tsx ie Root layout file

import './globals.css'

import type { Metadata } from 'next'

import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {

  title: 'Create Next App',

  description: 'Generated by create next app',}

export default function RootLayout({

  children,

}: {

  children: React.ReactNode

}) {

  return (

    <html lang="en">

      <body className={inter.className}>{children}</body>

    </html>

  )}

And index file being replaced with page.

**Placeholder Data:** When you're building user interfaces, it helps to have some placeholder data. If a database or API is not yet available, you can:

1-Use placeholder data in JSON format or as JavaScript objects.

2-Use a 3rd party service like mockAPI-The easiest way to mock REST APIs

**Type Script Support:** Next.js detects if your project uses TypeScript and automatically installs the necessary packages and configuration. Next.js also comes with a TypeScript plugin for your code editor, to help with auto-completion and type-safety.

We're manually declaring the data types, but for better type-safety, we recommend tools like **Prisma,** which automatically generates types based on your database schema.

**Prisma** is an open-source, next-generation database **toolkit** for developing applications with databases. It provides a set of tools and libraries that simplify database access and management, making it easier for developers to work with databases in their applications. Prisma supports multiple databases, including PostgreSQL, MySQL, SQLite, and SQL Server.

**Key Features oF Prisma:**

**Prisma Client:** Prisma Client is an auto-generated query builder and ORM (Object-Relational Mapping) that allows developers to interact with the database using a type-safe and auto-completed API. Prisma Client is generated based on your database schema, and it provides a modern and ergonomic way to perform database operations.

**Database Migrations:** Prisma supports schema migrations, enabling developers to manage and evolve the database schema over time. Migrations are created and executed using the Prisma CLI.

**Database Agnostic:** Prisma is designed to work with multiple databases. You can switch between different database providers without changing your application code. Prisma's data model definitions remain the same, making it highly portable.

**CSS Styling:**

**1-Global CSS-**Add global styles to your application by navigating to /app/layout.tsx and importing the global.css file. Inside global.css, you'll notice some @tailwind directives

@tailwind base;

@tailwind components;

@tailwind utilities;

**2-Tailwind-**Tailwind is a CSS framework that allowing you to write utility classes directly in your JSX markup.

<h1 className="text-blue-500">I'm blue!</h1>

**3-CSS Modules-**CSS Modules allow you to scope CSS to a component by **automatically creating unique class names**, so you don't have to worry about name collisions. Check Notes.

If we inspect this module.css element in dev tool, we found unique class name and this unique class name has been generated by Post Css.

home.module.css

 .shape {height: 0; width: 0;}

**/app/page.tsx**

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

<div className={styles.shape}></div>;

Using **clsx library** to toggle class names for conditionally styling the element.

import clsx from 'clsx';

export default function InvoiceStatus({ status }: { status: string }) {

  return (

    <span

      className={clsx(

        'inline-flex items-center rounded-full px-2 py-1 text-sm',

        {

          'bg-gray-100 text-gray-500': status === 'pending',

          'bg-green-500 text-white': status === 'paid',

},)})}

4-Sass which allows you to import .css and .scss files.

5-CSS-in-JS libraries such as styled-jsx, styled-components, and emotion we also can use.

**Optimizing Fonts and Images:**

Next.js automatically optimizes fonts in the application when you use the next/font module.

Next.js automatically optimizes fonts in the application when you use the next/font module. It does so by downloading font files at build time and hosting them with your other static assets. This means when a user visits your application, there are no additional network requests for fonts which would impact performance.

Import the Inter font from the next/font/google module - this will be your primary font and latin is subset of Inter font.

import './globals.css'

import type { Metadata } from 'next'

import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {

  title: 'Create Next App',

  description: 'Generated by create next app',

}

export default function RootLayout({

  children,

}: {

  children: React.ReactNode

}) {

  return (

    <html lang="en">

      <body className={inter.className}>{children}</body>

    </html>

  )

}

**Image Optimization:**

The **<Image> Component** is an extension of the HTML <img> tag, and comes with automatic image optimization, such as:

1-Preventing layout shift automatically when images are loading means It generates multiple sizes of each image for different devices.

2-Resizing images to avoid shipping large images to devices with a smaller viewport.

3-**Lazy loading images by default** (images load as they enter the viewport).

4-Serving images in m odern formats, like WebP and AVIF, when the browser supports it.

export default function Page() {

  return (

    <div className="flex items-center justify-center p-6 md:w-3/5 md:px-28 md:py 12">

      <Image

        src="/hero-desktop.png"

        width={1000}

        height={760}

        className="hidden md:block"

        alt="Screenshots of the dashboard project"

      />

    </div>

  );}

**Routing: File-system based routing**

**Nested Routing:** Next.js uses file-system routing where folders are used to create nested routes. Each folder represents a route segment that maps to a URL segment.

page.tsx is a special Next.js file that exports a React component containing the UI for the route.

Layout file in Next.Js is to share UI across multiple pages.

**Navigating Between Pages**

To link between pages, traditionally use the <a> that makes full page refresh on each page navigation. In Next.js we use <Link> Component with **no** full-page refresh that allows you to do client-side navigation with JavaScript.

**<Link> component** will prefetches the code linked to route.

**import Link from 'next/link';**

**Client-Side Navigation:**

By using <a> tag for navigation it will make page refresh/Reloaded means entire parts of the app will get Redownloaded, to avoid this we use Link component in Next.js, it will only load the content of the link component not the entire component is called client-side navigation.

Import link from ‘next/link’ ;

Next.js provides two main ways to handle routing and navigation in your application: useRouter from next/router and usePathname from next/navigation. Each of these hooks serves a different purpose and has unique capabilities:

**1-useRouter () hook from next/router:**

useRouter is the most commonly used hook for handling routing and navigation in Next.js applications. It provides access to the full range of routing capabilities, including routing events and various properties of the route.

You can use useRouter to get the pathname, query, and other routing information.

It's well-suited for building complex navigation logic and responding to routing events, such as when the route changes.

You can navigate between pages using methods like **push, replace**, **reload,** and **back** provided by useRouter.

Example: import { useRouter } from 'next/router';

function MyComponent() {

  const router = useRouter();

  const navigateToAbout = () => {

    router.push('/about');};

  return (

    <button onClick={navigateToAbout}>Go to About Page</button>);}

**2-usePathname () hook from next/navigation:**

usePathname is a hook provided by Next.js specifically for obtaining the current pathname (or URL). It's a more specialized hook for cases where you only need to access the current pathname and don't require the full range of routing capabilities.

usePathname is a lightweight option if you only need to know the current route's pathname and don't need to navigate between pages.

  import { usePathname } from 'next/navigation';

function MyComponent() {

  const pathname = usePathname();

  return (

    <p>Current Pathname: {pathname}</p>);}

**Pattern Showing Active link:** to show an active link to indicate to the user what page they are currently on. To do this, you need to get the user's current path from the URL. Next.js provides a hook called **usePathname()** that you can use to check the path.

Since usePathname() is a hook, you'll need to turn nav-links.tsx into a Client Component. Add React's "use client" directive to the top of the file, then import usePathname() from next/navigation:

The [useRouter](https://nextjs.org/docs/pages/api-reference/functions/use-router) from next/router is to be used in the [pages](https://nextjs.org/docs/getting-started/project-structure) folder, the initial way of setting up routes in Next.js. Since v13, they introduced a new directory called [app](https://nextjs.org/docs/getting-started/project-structure) (used when you say Yes to the last question shown in the below image), built on top of [Server Components](https://nextjs.org/docs/getting-started/react-essentials#server-components), where you define routes differently and use [useRouter](https://nextjs.org/docs/app/api-reference/functions/use-router) from next/navigation:

A screenshot of a computer program

Description automatically generated

Dynamic Routing: [page] folder then page.tsx file with useRouter () hook.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

'use client';

      import Link from 'next/link';

      import { usePathname } from 'next/navigation';

      export default function NavLinks() {

        const pathname = usePathname();

        return (

          <>

            {links.map((link) => {

              const LinkIcon = link.icon;

              return (

                <Link

                  key={link.name}

                  href={link.href}>

                  <LinkIcon className="w-6" />

                  <p className="hidden md:block">{link.name}</p>

                </Link>

              );})}</>);}

|  |  |
| --- | --- |
| **Version 14**  A screenshot of a computer program  Description automatically generated | **Version 13**  A screenshot of a computer program  Description automatically generated |

**What is streaming?**

Streaming is a data transfer technique that allows you to break down a route into smaller "chunks" and progressively stream them from the server to the client as they become ready.

By streaming, you can prevent slow data requests from blocking your whole page. This allows the user to see and interact with parts of the page without waiting for all the data to load before any UI can be shown to the user.

Data fetching and rendering are initiated in parallel, so the user can see the UI as it becomes ready. This is different from the traditional waterfall approach, where data fetching and rendering are initiated sequentially, blocking the UI from rendering until all the data is ready.

A screenshot of a computer

Description automatically generated

Route groups allow you to organize files into logical groups without affecting the URL path structure. When you create a new folder using parentheses (), the name won't be included in the URL path. So /dashboard/(overview)/page.tsx becomes /dashboard.

A screenshot of a computer

Description automatically generated

So far, you're streaming a whole page. But, instead, you can be more granular and stream specific components using React Suspense.

Suspense allows you to defer rendering parts of your application until some condition is met (e.g. data is loaded). You can wrap your dynamic components in Suspense. Then, pass it a fallback component to show while the dynamic component loads.

**Partial Prerendering: New feature**

In Next.js 14, there is a preview of a new compiler optimization called Partial Prerendering. Partial Prerendering is an experimental feature that allows you to render a route with a static loading shell, while keeping some parts dynamic. In other words, you can isolate the dynamic parts of a route.

When a user visits a route:

-A static route shell is served; this makes the initial load fast.

-The shell leaves places where dynamic content will load in async.

-The async holes is loaded in parallel, reducing the overall load time of the page.

-By adding the **'use server'**, you mark all the exported functions within the file as server functions. These server functions can then be imported into Client and Server components, making them extremely versatile.

**Action object:**

in React, the action attribute is considered a special prop - meaning React builds on top of it to allow actions to be invoked. Rather than calling an API explicitly, you can pass a function to action.

return (<form action={createInvoice}>)

**Authentication** is about making sure the user is who they say they are. You're proving your identity with something you have like a username and password.

**Authorization** is the next step. Once a user's identity is confirmed, authorization decides what parts of the application they are allowed to use.

**NextAuth.js- npm install next-auth@beta bcrypt,**

1-Create Environment

2-Confique the environment

3-Add configured env to middleware for extra validation.

4-password hashing by cryptic library

5- you can use **zod** to validate the email and password before checking if the user exists in the database:

6- import bcrypt from 'bcrypt'; to compare the has password

7-adding signIn functionality and update the form according. Check error in case

8-Add logout functionality.

|  |  |
| --- | --- |
| 1-Setting up Environnent   AUTH\_SECRET=your-secret-key   AUTH\_URL=http://localhost:3000/api/auth | 2-Confiqure it  import type { NextAuthConfig } from 'next-auth';          export const authConfig = {          pages: {signIn: '/login',},}; |
|  |  |
|  |  |

3-The authorized callback is used to verify if the request is authorized to access a page via Next.js Middleware. It is called before a request is completed, and it receives an object with the auth and request properties. The auth property contains the user's session, and the request property contains the incoming request.

**Metadata**

metadata provides additional details about a webpage. Metadata is not visible to the users visiting the page. Instead, it works behind the scenes, embedded within the page's HTML, usually within the <head> element. This hidden information is crucial for search engines and other systems that need to understand your webpage's content better.

**Why Metadata is Important:**

Metadata plays a significant role in enhancing a webpage's SEO, making it more accessible and understandable for search engines and social media platforms. Proper metadata helps search engines effectively index webpages, improving their ranking in search results.

<title>Page Title</title>

      <meta name="keywords" content="keyword1, keyword2, keyword3" />

      <meta property="og:title" content="Title Here" />

      <link rel="icon" href="path/to/favicon.ico" />

**Adding metadata:**

Next.js has a Metadata API that can be used to define your application metadata. There are two ways you can add metadata to your application:

**1-Config-based:** Export a static metadata object {} or a dynamic generateMetadata() function in a layout.js or page.js file.

**2-File-based:** Next.js has a range of special files are specifically recognized for metadata purposes:ex-

**favicon.ico,** apple-icon.jpg, and icon.jpg: Utilized for favicons and icons

**opengraph**-image.jpg and twitter-image.jpg: Employed for social media images

**robots.txt**: Provides instructions for search engine crawling

**sitemap.xml:** Offers information about the website's structure

You have the flexibility to use these files for static metadata, or you can generate them programmatically within your project. With both these options, Next.js will automatically generate the relevant <head> elements for your pages.

**Client-Side Navigation:**

**By using <a> tag for navigation it will make page refresh/Reloaded means entire parts of the app will get Redownloaded, to avoid this we use Link component in Next.js, it will only load the content of the link component not the entire component is called client-side navigation.**

**Import link from ‘next/link’ ;**

**Import {useRouter} from next/router**

**SSR VS CSR-**

**CSR-**

1-In client-side rendering we have to bundle all our components and send them to the client for rendering this means as our application grows so does our bundle size because it must contain all of our components now the larger the bundle the more memory, we need on the client to load all these components so this approach is resource **heavy**.

2-the other problem is that search engine bot Parts which are machines that browse and index our websites can't view our content because they can't execute JavaScript code so they cannot render our components like a web browser

3-and last but not least any sensitive data we have in our components or their dependencies like API keys will be exposed to the client.

**SSR-**

1-now if we render our components on the server we can get rid of all these problems we only send the essential components to the client and prevent our bundle from becoming unnecessarily large also because the server handles most of the rendering we need less resources on the client plus because rendering is done on the server and we send the actual content to the client search engine Bots can view and index our pages and finally we can keep sensitive data like API keys on the server so these are all the great benefits of server-side rendering

however, with server-side rendering we lose interactivity so server components which are components that are rendered on the server cannot listen to browser events like Click Change submit and so on they cannot access browser apis like the local storage they cannot maintain state or use effects these functionalities are only available in

client components

so, in real-world applications we often use a mixture of server and client components we should default to server components and use client components only when we

absolutely need them.

Ex- let's imagine we want to build a page to show a list of products to build this page we probably need several components like navbar sidebar product list product card pagination and footer now in standard react applications we have to package all these components and send them to the client for rendering.

But in next.js we can keep all these components on the server and minimize the bundle size there is just one exception to add a product to a shopping cart we need to handle the click event of a button typically we implement this functionality in the product card component so we have to make it a client component that's one option

but there is a better way we can keep this component on the server and do most of the rendering there and instead extract a small component that only contains the add button with this change we only ship that tiny component to the client and keep everything else on the server let's see this in action back to our project in next.js all components inside the app folder are server components by default.

**Next JS key features:**

**1-Server-side rendering (SSR**) is a technique that allows web pages to be rendered on the server before being sent to the client's browser. Next.js is a popular React framework that provides built-in support for server-side rendering.

To enable server-side rendering in Next.js, you need to create a page in the pages directory and export a function called **getServerSideProps().** This function is responsible for fetching the necessary data and returning it as props to the page component.

      function HomePage({ data }) {

        return (

          <div>

            {data.map(item => (

              <div key={item.id}>{item.title}</div>

            ))}

          </div>

        );

      }

      export async function getServerSideProps() {

        const res = await fetch('https://api.example.com/data');

        const data = await res.json();

        return {

          props: { data }

        };

      }

      export default HomePage;

In this example, the getServerSideProps function fetches data from an API and returns it as props to the HomePage component. When the page is requested, Next.js will render the page on the server and send the HTML to the client's browser.

Server-side rendering can improve performance and SEO because it allows the initial page load to be faster and ensures that search engines can properly index the page content.