

Curso de Nextjs

Diego Saavedra

Jul 15, 2024

Table of contents

1	Bienvenido	6
1.1	¿De qué trata este curso?	6
1.2	¿Para quién es este curso?	6
1.3	¿Cómo contribuir?	7
I	Unidad 1: Introducción a Nextjs	8
2	Introducción	9
3	¿Qué es Next.js y por qué aprenderlo si quieres ser frontend senior?	11
3.1	Introducción a Next.js y sus beneficios.	11
3.2	Comparación con otros frameworks.	11
3.3	Casos de uso y ejemplos de proyectos exitosos.	12
3.4	Conclusión.	12
4	Arquitectura de un proyecto de Next.js	13
4.1	Estructura de carpetas y archivos.	13
4.2	Configuración inicial de un proyecto.	14
4.3	Revisión de un proyecto ejemplo.	14
5	Herramientas y stack utilizado en el curso	16
5.1	Herramientas necesarias para el curso.	16
5.2	Configuración del entorno de desarrollo.	16
5.3	Instalación y configuración de dependencias.	16
5.4	Iniciar el servidor local.	17
5.5	Acceder a la aplicación en el navegador.	17
6	Cómo crear rutas en Next.js	18
6.1	Rutas básicas.	18
6.1.1	Componente Index:	18
6.2	Rutas dinámicas.	18
6.3	Nested routes.	19
6.4	Rutas con parámetros.	19
6.5	Rutas con query strings.	19
6.6	Rutas con rutas anidadas.	19
6.7	Rutas con rutas anidadas y parámetros.	19
6.8	Rutas con rutas anidadas y query strings.	19
6.9	Conclusión.	20
6.10	Ejercicios.	20

7	Cómo crear Layout en Next.js	21
7.1	Definición de layouts.	21
7.1.1	Componente Layout:	21
7.1.2	Componente Index :	22
7.1.3	Componente Layout específico:	22
7.1.4	Componente About :	23
7.2	Uso de layouts globales y específicos.	23
7.2.1	Layout global:	23
7.2.2	Layout específico:	24
7.3	Conclusión.	24
7.4	Ejercicios.	24
7.5	Implementación de layouts en un proyecto.	25
8	Cómo funciona la navegación en Next.js	26
8.1	Link component.	26
8.2	Uso de Router.	27
8.3	Navegación programática.	27
8.4	Conclusión.	28
8.5	Ejercicio.	28
9	Manejo de parámetros en rutas en Next.js	29
9.1	Parámetros de ruta.	29
9.2	Parámetros de consulta (query params).	29
9.3	Uso de useRouter para acceder a los parámetros.	30
9.4	Conclusión.	31
9.5	Ejercicios.	31
10	React Server Components en Next.js: notación “use Client”	32
10.1	Introducción a React Server Components.	32
10.1.1	¿Qué es use Client?	32
10.2	Uso de la notación “use Client”.	32
10.3	Ventajas de la notación “use Client”.	33
10.4	Ejemplos prácticos.	33
10.5	Conclusión.	34
10.6	Ejercicios.	34
II	Unidad 2: Manejo de Estilos y Archivos Estáticos	35
11	Manejo de estilos y archivos estáticos en Next.js	36
11.1	Estilos en Next.js	36
11.1.1	CSS global	36
11.2	CSS Modules en Next.js	37
11.2.1	CSS Modules	37
11.2.2	Scoped CSS en Next.js	37
11.2.3	Creación y uso de CSS Modules.	37
11.3	Styled JSX en Next.js	38
11.3.1	Styled JSX	38
11.3.2	Ventajas de Styled JSX	39

11.4	Conclusión	39
11.5	Ejercicios	39
12	Uso de Sass en Next.js	40
12.1	Instalación y configuración de Sass.	40
12.2	Uso de Sass en componentes.	40
12.3	Variables y mixins en Sass.	41
12.4	Conclusión.	42
12.5	Ejercicios.	42
13	Cómo utilizar estilos globales en Next.js	43
13.1	Estilos globales vs. estilos locales.	43
13.2	Implementación de estilos globales.	43
13.3	Uso de reset y normalize.css.	44
13.4	Conclusión.	44
13.5	Ejercicios.	44
14	Cómo agregar archivos estáticos en Next.js	45
14.1	Carpeta public.	45
14.2	Acceso y uso de archivos estáticos.	45
14.3	Ventajas de la carpeta public.	46
14.4	Ejemplos prácticos.	46
14.5	Conclusión.	47
14.6	Ejercicios.	47
15	Manejo y optimización de imágenes con Next Image	48
15.1	Introducción al componente Image.	48
15.2	Configuración y optimización de imágenes.	48
15.3	Uso de imágenes remotas y locales.	49
15.4	Conclusión.	50
16	Optimización de fuentes con Next.js	51
16.1	Uso de fuentes personalizadas.	51
16.2	Optimización de carga de fuentes.	51
16.3	Ejemplos prácticos.	52
16.4	Conclusión.	53
16.5	Ejercicios	53
17	Creando estilos dinámicos aplicando condicionales en Next.js	55
17.1	Estilos dinámicos.	55
17.2	Aplicación de condicionales en estilos.	55
17.3	Ejemplos prácticos.	56
17.4	Conclusión.	57
17.5	Ejercicio.	58
III	Laboratorios	59
18	Laboratorio de Fetch con Next	60
18.1	Pasos	60

18.2 Resultado	65
18.3 Conclusión	65
18.4 Ejercicio	66

1 Bienvenido



Figure 1.1: Typescript

¡Bienvenido al Curso de Typescript!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso es una introducción a TypeScript, un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. TypeScript es un superconjunto de JavaScript que agrega tipado estático opcional y otras características avanzadas a JavaScript.

En este curso, aprenderá los conceptos básicos de TypeScript, incluidos los tipos de datos, las funciones, las clases, los módulos y mucho más. También explorará cómo TypeScript se puede utilizar para crear aplicaciones web modernas y escalables.

Este curso es ideal para principiantes y aquellos con poca o ninguna experiencia en programación. Si eres un estudiante curioso, un profesional que busca cambiar de carrera o simplemente alguien que quiere aprender TypeScript, este curso es para ti.

1.2 ¿Para quién es este curso?

Este curso es para cualquier persona interesada en aprender TypeScript, incluidos:

- Estudiantes que deseen aprender un nuevo lenguaje de programación.
- Profesionales que buscan mejorar sus habilidades de desarrollo web.
- Desarrolladores que deseen aprender TypeScript para crear aplicaciones web modernas y escalables.
- Cualquiera que quiera aprender un lenguaje de programación de código abierto y de alto rendimiento.
- Cualquiera que quiera aprender TypeScript para mejorar su carrera profesional.

- Cualquiera que quiera aprender TypeScript para crear aplicaciones web modernas y escalables.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en línea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

¡Gracias por su interés en este curso y espero que disfrute aprendiendo TypeScript!

Part I

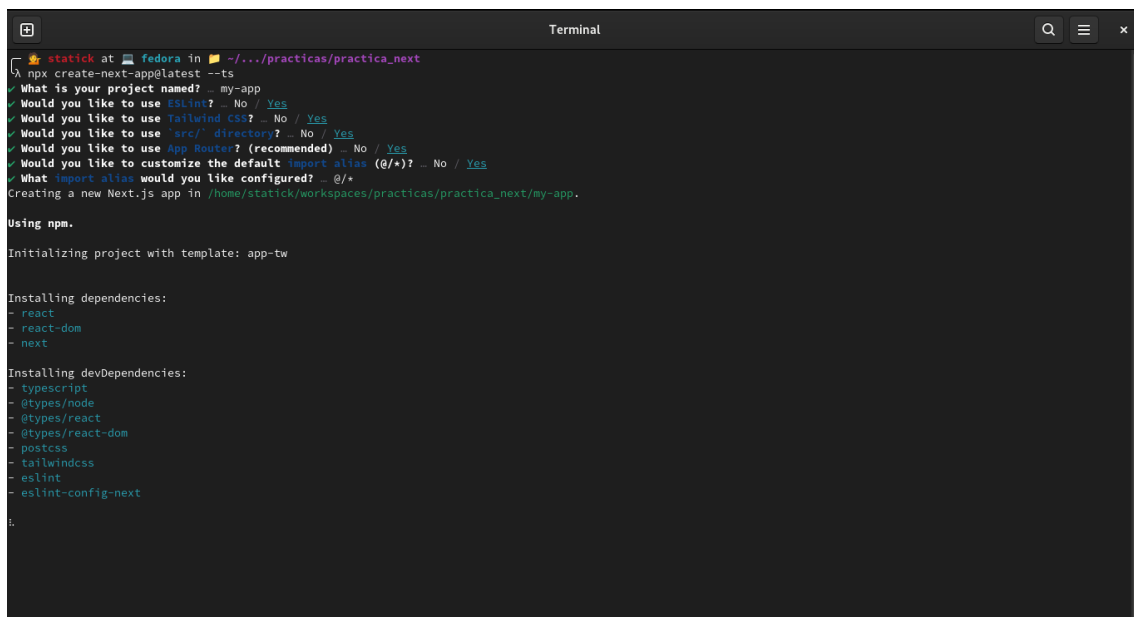
Unidad 1: Introducción a Nextjs

2 Introducción

En este ebook se presentan los conceptos básicos acerca del framework Next JS, el cual es un framework de React que permite la creación de aplicaciones web de forma sencilla y rápida. A lo largo de este ebook se presentarán los conceptos básicos de Next JS, así como ejemplos de su uso.

Lo primero que se debe hacer es instalar Next JS en su computadora. Para ello, se debe tener instalado Node JS en su computadora. Si no lo tiene instalado, puede descargarlo desde la página oficial de Node JS. Una vez que tenga Node JS instalado, puede instalar Next JS utilizando el siguiente comando:

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



```
statik at fedora in ~/.../practicas/practica_next
$ npx create-next-app@latest --ts
✔ What is your project named?  my-app
✔ Would you like to use eslint?  No / Yes
✔ Would you like to use Tailwind CSS?  No / Yes
✔ Would you like to use 'src/' directory?  No / Yes
✔ Would you like to use App Router? (recommended)  No / Yes
✔ Would you like to customize the default import alias (@/*)?  No / Yes
✔ What import alias would you like configured?  @/*
Creating a new Next.js app in /home/statik/workspaces/practicas/practica_next/my-app.

Using npm.

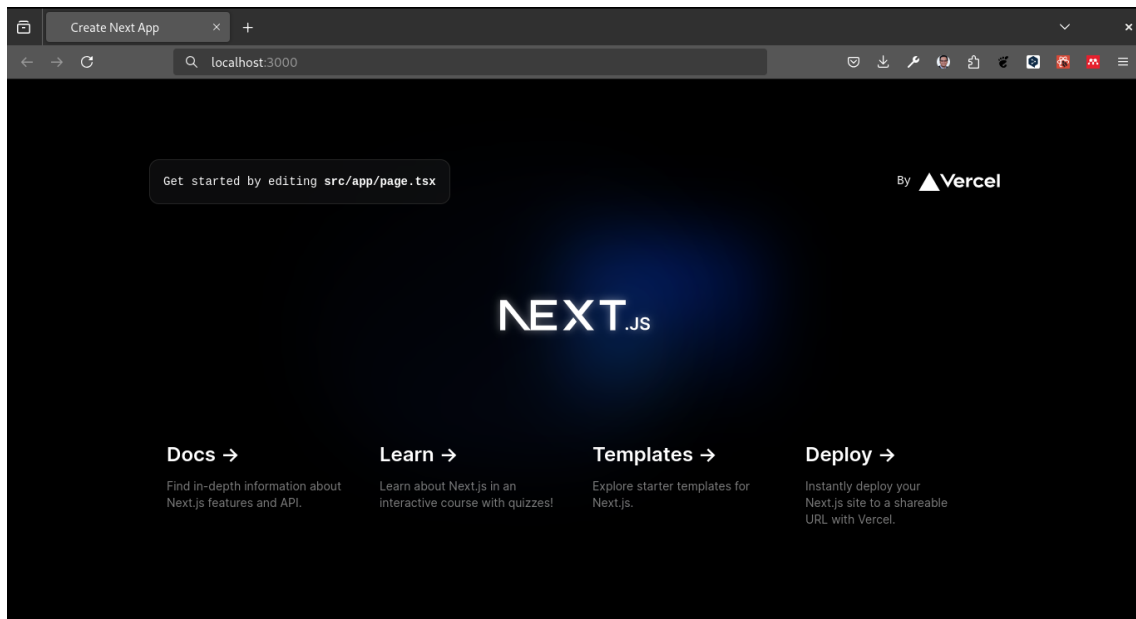
Initializing project with template: app-tw

Installing dependencies:
- react
- react-dom
- next

Installing devDependencies:
- typescript
- @types/node
- @types/react
- @types/react-dom
- postcss
- tailwindcss
- eslint
- eslint-config-next
b.
```

Este comando creará una nueva aplicación de Next JS en su computadora. Una vez que la aplicación se haya creado, puede ejecutarla utilizando el siguiente comando:

```
npm run dev
```



Este comando iniciará un servidor local en su computadora y podrá ver la aplicación en su navegador web. A partir de aquí, puede comenzar a desarrollar su aplicación utilizando Next JS.

En los siguientes capítulos se presentarán los conceptos básicos de Next JS, así como ejemplos de su uso. Espero que este ebook le sea de utilidad y le ayude a comprender mejor el framework Next JS.

3 ¿Qué es Next.js y por qué aprenderlo si quieres ser frontend senior?

Next.js es un framework de React que permite la creación de aplicaciones web de forma sencilla y rápida. A lo largo de este ebook se presentarán los conceptos básicos de Next.js, así como ejemplos de su uso.

3.1 Introducción a Next.js y sus beneficios.

Para introducirnos en Next.js, primero debemos entender qué es un framework. Un framework es un conjunto de herramientas y librerías que facilitan el desarrollo de aplicaciones web. En el caso de Next.js, es un framework de React que nos permite crear aplicaciones web de forma sencilla y rápida.

Next.js nos ofrece una serie de beneficios que lo hacen una excelente opción para el desarrollo de aplicaciones web:

- **Rendimiento:** Next.js nos ofrece un rendimiento óptimo gracias a su capacidad de renderizado en el servidor y en el cliente. Esto nos permite crear aplicaciones web rápidas y eficientes.
- **SEO:** Next.js nos ofrece una serie de herramientas que nos permiten optimizar nuestras aplicaciones web para los motores de búsqueda. Esto nos ayuda a mejorar el posicionamiento de nuestras aplicaciones en los resultados de búsqueda.
- **Escalabilidad:** Next.js nos ofrece una arquitectura escalable que nos permite crear aplicaciones web de cualquier tamaño. Esto nos permite crear aplicaciones web que puedan crecer con el tiempo y adaptarse a las necesidades de nuestros usuarios.
- **Facilidad de uso:** Next.js nos ofrece una serie de herramientas y librerías que nos facilitan el desarrollo de aplicaciones web. Esto nos permite crear aplicaciones web de forma sencilla y rápida, sin necesidad de tener un conocimiento profundo de React.

3.2 Comparación con otros frameworks.

Podemos comparar Next.js con otros frameworks de React, como Create React App. A continuación, se presentan algunas diferencias entre Next.js y Vite:

- **Rendimiento:** Next.js ofrece un rendimiento óptimo gracias a su capacidad de renderizado en el servidor y en el cliente. Vite, por otro lado, ofrece un rendimiento óptimo gracias a su capacidad de renderizado en el cliente.

- **SEO:** Next.js nos ofrece una serie de herramientas que nos permiten optimizar nuestras aplicaciones web para los motores de búsqueda. Vite, por otro lado, nos ofrece una serie de herramientas que nos permiten optimizar nuestras aplicaciones web para los motores de búsqueda.
- **Escalabilidad:** Next.js nos ofrece una arquitectura escalable que nos permite crear aplicaciones web de cualquier tamaño. Vite, por otro lado, nos ofrece una arquitectura escalable que nos permite crear aplicaciones web de cualquier tamaño.
- **Facilidad de uso:** Next.js nos ofrece una serie de herramientas y librerías que nos facilitan el desarrollo de aplicaciones web. Vite, por otro lado, nos ofrece una serie de herramientas y librerías que nos facilitan el desarrollo de aplicaciones web.

3.3 Casos de uso y ejemplos de proyectos exitosos.

Next.js es utilizado por una gran cantidad de empresas y desarrolladores en todo el mundo. Algunos ejemplos de proyectos exitosos que utilizan Next.js son:

- **Vercel:** Vercel es una plataforma de desarrollo y alojamiento de aplicaciones web que utiliza Next.js como su framework principal. Vercel nos ofrece una serie de herramientas y servicios que nos permiten crear aplicaciones web de forma sencilla y rápida.
- **Spotify:** Spotify es una plataforma de streaming de música que utiliza Next.js para su aplicación web. Spotify nos ofrece una experiencia de usuario rápida y eficiente gracias a Next.js.
- **Netflix:** Netflix es una plataforma de streaming de películas y series que utiliza Next.js para su aplicación web. Netflix nos ofrece una experiencia de usuario rápida y eficiente gracias a Next.js.

3.4 Conclusión.

Next.js es un framework de React que nos permite crear aplicaciones web de forma sencilla y rápida. A lo largo de este ebook se presentarán los conceptos básicos de Next.js, así como ejemplos de su uso. Espero que este ebook le sea de utilidad y le ayude a comprender mejor el framework Next.js.

4 Arquitectura de un proyecto de Next.js

En este capítulo se presentará la arquitectura de un proyecto de Next.js, así como las carpetas y archivos que conforman un proyecto

4.1 Estructura de carpetas y archivos.

La estructura de un proyecto de Next.js se compone de las siguientes carpetas y archivos:

- **pages:** En esta carpeta se encuentran las páginas de la aplicación. Cada archivo en esta carpeta representa una página de la aplicación. Por ejemplo, si se crea un archivo `index.js` en esta carpeta, se creará una página de inicio en la aplicación.
- **public:** En esta carpeta se encuentran los archivos estáticos de la aplicación, como imágenes, estilos y scripts. Estos archivos se pueden acceder directamente desde la URL de la aplicación.
- **styles:** En esta carpeta se encuentran los estilos globales de la aplicación. Estos estilos se aplican a toda la aplicación y se pueden importar en cualquier archivo de la aplicación.
- **components:** En esta carpeta se encuentran los componentes de la aplicación. Estos componentes se pueden reutilizar en diferentes partes de la aplicación.
- **lib:** En esta carpeta se encuentran las librerías y utilidades de la aplicación. Estas librerías se pueden importar en cualquier archivo de la aplicación.
- **api:** En esta carpeta se encuentran los endpoints de la API de la aplicación. Estos endpoints se pueden acceder desde la URL de la aplicación.
- **config:** En esta carpeta se encuentran los archivos de configuración de la aplicación. Estos archivos se pueden utilizar para configurar diferentes aspectos de la aplicación.
- **test:** En esta carpeta se encuentran los archivos de pruebas de la aplicación. Estos archivos se pueden utilizar para probar diferentes aspectos de la aplicación.
- **.env:** En este archivo se encuentran las variables de entorno de la aplicación. Estas variables se pueden utilizar para configurar diferentes aspectos de la aplicación.

4.2 Configuración inicial de un proyecto.

Para realizar la configuración inicial de un proyecto de Next.js, se deben seguir los siguientes pasos:

1. Crear un nuevo proyecto de Next.js utilizando el siguiente comando:

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

2. Instalar las dependencias del proyecto utilizando el siguiente comando:

```
npm install
```

3. Iniciar el servidor local utilizando el siguiente comando:

```
npm run dev
```

4. Acceder a la aplicación en el navegador web utilizando la URL `http://localhost:3000`.

Con estos pasos, se habrá creado un nuevo proyecto de Next.js y se podrá comenzar a desarrollar la aplicación.

4.3 Revisión de un proyecto ejemplo.

A continuación se presenta un ejemplo de la estructura de un proyecto de Next.js:

```
my-next-app/  
  pages/  
    index.tsx  
    about.tsx  
    contact.tsx  
  public/  
    images/  
      logo.png  
    styles/  
      main.css  
  styles/  
    global.css  
  components/  
    header.tsx  
    footer.tsx  
    button.tsx  
  lib/  
    api.ts  
    utils.ts  
  api/  
    users.ts
```

```
    posts.ts
  config/
    app.config.ts
  test/
    app.test.ts
  .env
```

En este ejemplo, se puede observar la estructura de un proyecto de Next.js, así como las carpetas y archivos que conforman el proyecto

5 Herramientas y stack utilizado en el curso

En este curso utilizaremos las siguientes herramientas y tecnologías:

5.1 Herramientas necesarias para el curso.

Para poder seguir este curso, necesitarás tener instaladas las siguientes herramientas en tu computadora:

- **Node.js:** Es un entorno de ejecución para JavaScript que nos permite ejecutar código JavaScript en el servidor. Puedes descargar Node.js desde la página oficial de Node.js.
- **npm:** Es el gestor de paquetes de Node.js que nos permite instalar y gestionar las dependencias de nuestros proyectos. npm viene incluido con Node.js, por lo que no es necesario instalarlo por separado.
- **Visual Studio Code:** Es un editor de código fuente desarrollado por Microsoft que nos permite escribir y editar código de forma sencilla. Puedes descargar Visual Studio Code desde la página oficial de Visual Studio Code.
- **Git:** Es un sistema de control de versiones que nos permite gestionar el código fuente de nuestros proyectos. Puedes descargar Git desde la página oficial de Git.

5.2 Configuración del entorno de desarrollo.

Creemos un nuevo proyecto de Next.js utilizando el siguiente comando:

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

5.3 Instalación y configuración de dependencias.

Instalamos las dependencias del proyecto utilizando el siguiente comando:

```
npm install
```


5.4 Iniciar el servidor local.

Iniciamos el servidor local utilizando el siguiente comando:

```
npm run dev
```

5.5 Acceder a la aplicación en el navegador.

Accedemos a la aplicación en el navegador web utilizando la URL `http://localhost:3000`.

6 Cómo crear rutas en Next.js

En este capítulo se presentarán los conceptos básicos de cómo crear rutas en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

6.1 Rutas básicas.

Las rutas en Next.js se crean utilizando la carpeta **pages**. Cada archivo en esta carpeta representa una ruta en la aplicación. Por ejemplo, si se crea un archivo **index.js** en esta carpeta, se creará una ruta de inicio en la aplicación.

6.1.1 Componente Index:

```
const Index = () => {  
  return (  
    <div>  
      <h1>Home Page</h1>  
      <p>Welcome to my Next.js App</p>  
    </div>  
  )  
}  
  
export default Index
```

En el ejemplo anterior, se crea una ruta de inicio en la aplicación con el componente **Index** que renderiza el contenido de la página.

6.2 Rutas dinámicas.

Las rutas dinámicas en Next.js se crean utilizando corchetes `[]` en el nombre del archivo. Por ejemplo, si se crea un archivo **[id].js** en esta carpeta, se creará una ruta dinámica en la aplicación.

6.3 Nested routes.

Las rutas anidadas en Next.js se crean utilizando la carpeta **pages** y subcarpetas. Por ejemplo, si se crea una carpeta **blog** y un archivo **[slug].js** en esta carpeta, se creará una ruta anidada en la aplicación.

6.4 Rutas con parámetros.

Las rutas con parámetros en Next.js se crean utilizando corchetes `[]` en la ruta. Por ejemplo, si se crea una ruta `/blog/[slug]` en la aplicación, se creará una ruta con parámetros en la aplicación.

6.5 Rutas con query strings.

Las rutas con query strings en Next.js se crean utilizando el signo de interrogación `?` en la ruta. Por ejemplo, si se crea una ruta `/blog?slug=hello-world` en la aplicación, se creará una ruta con query strings en la aplicación.⁹

6.6 Rutas con rutas anidadas.

Las rutas con rutas anidadas en Next.js se crean utilizando la carpeta **pages** y subcarpetas. Por ejemplo, si se crea una carpeta **blog** y un archivo **[slug].js** en esta carpeta, se creará una ruta anidada en la aplicación.

6.7 Rutas con rutas anidadas y parámetros.

Las rutas con rutas anidadas y parámetros en Next.js se crean utilizando la carpeta **pages** y subcarpetas. Por ejemplo, si se crea una carpeta **blog** y un archivo **[slug].js** en esta carpeta, se creará una ruta anidada con parámetros en la aplicación.

6.8 Rutas con rutas anidadas y query strings.

Las rutas con rutas anidadas y query strings en Next.js se crean utilizando la carpeta **pages** y subcarpetas. Por ejemplo, si se crea una carpeta **blog** y un archivo **[slug].js** en esta carpeta, se creará una ruta anidada con query strings en la aplicación.

6.9 Conclusión.

En este capítulo se presentaron los conceptos básicos de cómo crear rutas en Next.js. A lo largo de este capítulo se presentaron los siguientes temas: rutas básicas, rutas dinámicas, rutas anidadas, rutas con parámetros, rutas con query strings, rutas con rutas anidadas, rutas con rutas anidadas y parámetros, y rutas con rutas anidadas y query strings. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo crear rutas en Next.js.

6.10 Ejercicios.

1. Crea una ruta básica en Next.js.
2. Crea una ruta dinámica en Next.js.
3. Crea una ruta anidada en Next.js.
4. Crea una ruta con parámetros en Next.js.
5. Crea una ruta con query strings en Next.js.
6. Crea una ruta con rutas anidadas en Next.js.
7. Crea una ruta con rutas anidadas y parámetros en Next.js.
8. Crea una ruta con rutas anidadas y query strings en Next.js.

Espero que estos ejercicios le sean de utilidad y le ayuden a practicar cómo crear rutas en Next.js.

7 Cómo crear Layout en Next.js

En este capítulo se presentarán los conceptos básicos de cómo crear Layouts en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

7.1 Definición de layouts.

Para definir un layout en Next.js, se debe crear un componente de React que contenga la estructura del layout. Por ejemplo, si se desea crear un layout con un encabezado y un pie de página, se puede crear un componente **Layout** que contenga estos elementos.

7.1.1 Componente Layout:

```
import Head from 'next/head'

const Layout = ({ children }) => {
  return (
    <div>
      <Head>
        <title>My Next.js App</title>
      </Head>
      <header>
        <h1>Header</h1>
      </header>
      <main>
        {children}
      </main>
      <footer>
        <p>Footer</p>
      </footer>
    </div>
  )
}

export default Layout
```

En el componente **Layout**, se define la estructura del layout con un encabezado, un pie de página y un contenedor principal para el contenido de la página. El componente recibe como prop **children** el contenido de la página que se renderizará en el contenedor principal.

7.1.2 Componente Index:

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

const Index = () => {
  return (
    <Layout>
      <h1>Home Page</h1>
      <p>Welcome to my Next.js App</p>
    </Layout>
  )
}

export default Index
```

En el componente **Index**, se importa el componente **Layout** y se renderiza el contenido de la página dentro del layout. De esta forma, se crea una estructura de layout que se puede reutilizar en diferentes páginas de la aplicación.

7.1.3 Componente Layout específico:

```
import Head from 'next/head'

const Layout = ({ title, children }) => {
  return (
    <div>
      <Head>
        <title>{title}</title>
      </Head>
      <header>
        <h1>Header</h1>
      </header>
      <main>
        {children}
      </main>
      <footer>
        <p>Footer</p>
      </footer>
    </div>
  )
}

export default Layout
```

En el componente **Layout**, se define un prop **title** que permite personalizar el título de la página. De esta forma, se puede crear un layout específico para cada página de la aplicación.

7.1.4 Componente About:

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

const About = () => {
  return (
    <Layout title="About Page">
      <h1>About Page</h1>
      <p>Learn more about my Next.js App</p>
    </Layout>
  )
}

export default About
```

En el componente **About**, se importa el componente **Layout** y se renderiza el contenido de la página dentro del layout. Se utiliza el prop **title** para personalizar el título de la página. De esta forma, se crea un layout específico para la página **About**.

7.2 Uso de layouts globales y específicos.

En Next.js, se pueden crear layouts globales que se aplican a todas las páginas de la aplicación, así como layouts específicos que se aplican a páginas específicas de la aplicación. De esta forma, se puede personalizar el diseño de cada página de la aplicación de forma independiente.

7.2.1 Layout global:

Para crear un layout global en Next.js, se puede definir un componente **Layout** en un archivo separado y importarlo en todas las páginas de la aplicación. De esta forma, se aplica el mismo diseño a todas las páginas de la aplicación.

Ejemplo:

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

const MyApp = ({ Component, pageProps }) => {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  )
}
```

```

    </Layout>
  )
}

export default MyApp

```

7.2.2 Layout específico:

Para crear un layout específico en Next.js, se puede definir un componente **Layout** en un archivo separado y personalizarlo con props específicas para cada página. De esta forma, se puede crear un diseño único para cada página de la aplicación.

Ejemplo:

```

import Layout from '../components/Layout'

const About = () => {
  return (
    <Layout title="About Page">
      <h1>About Page</h1>
      <p>Learn more about my Next.js App</p>
    </Layout>
  )
}

export default About

```

En este ejemplo, se crea un layout específico para la página **About** con un título personalizado. De esta forma, se puede personalizar el diseño de cada página de la aplicación de forma independiente.

7.3 Conclusión.

En este capítulo se presentaron los conceptos básicos de cómo crear Layouts en Next.js. A lo largo de este capítulo se presentaron los siguientes temas: definición de layouts, uso de layouts globales y específicos. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo crear Layouts en Next.js.

7.4 Ejercicios.

1. Crea un layout con un encabezado y un pie de página en Next.js.
2. Crea un layout con un título personalizado en Next.js.
3. Crea un layout global que se aplique a todas las páginas de la aplicación.

4. Crea un layout específico que se aplique a una página específica de la aplicación.
5. Crea un layout con un menú de navegación en Next.js.
6. Crea un layout con un formulario de contacto en Next.js.

Espero que estos ejercicios le sean de utilidad y le ayuden a practicar cómo crear Layouts en Next.js.

7.5 Implementación de layouts en un proyecto.

8 Cómo funciona la navegación en Next.js

La navegación en Next.js se realiza a través de la librería `next/router`. Esta librería nos permite navegar entre las diferentes páginas de nuestra aplicación de forma programática o mediante enlaces.

Es necesario eliminar el directorio `src/app` y el archivo “`src/app.js`” para que la aplicación funcione correctamente.

8.1 Link component.

El componente **Link** de Next.js nos permite crear enlaces entre las diferentes páginas de nuestra aplicación. Este componente se utiliza de la siguiente forma:

```
//index.jsx
import Link from "next/link";

export default function Home() {
  return (
    <div>
      Hello World.{" "}
      <Link href="/about">
        About
      </Link>
    </div>
  );
}
```

En el ejemplo anterior, se crea un enlace a la página **About** utilizando el componente **Link**. Al hacer clic en el enlace, se navegará a la página **About** de forma rápida y sin recargar la página.

```
//_app.js
export default function App({ Component, pageProps }) {
  return <Component {...pageProps} />;
}
```

Este archivo es el punto de entrada de nuestra aplicación y se utiliza para envolver todos los componentes de la aplicación con un componente de diseño o estilo común.

```
//about.jsx
export default function About() {
  return <div>About</div>;
}
```

En este archivo se crea la página **About** de la aplicación. Al navegar a esta página, se mostrará el contenido del componente **About**.

8.2 Uso de Router.

La librería **next/router** nos permite navegar entre las diferentes páginas de nuestra aplicación de forma programática. Para utilizar esta librería, se debe importar el objeto **Router** de la siguiente forma:

```
import { useRouter } from 'next/router'

const About = () => {
  const router = useRouter()

  const handleClick = () => {
    router.push('/')
  }

  return (
    <div>
      <h1>About Page</h1>
      <button onClick={handleClick}>Go to Home Page</button>
    </div>
  )
}

export default About
```

En el ejemplo anterior, se importa el objeto **Router** de la librería **next/router** y se utiliza el método **push** para navegar a la página de inicio al hacer clic en un botón.

8.3 Navegación programática.

La navegación programática nos permite navegar entre las diferentes páginas de nuestra aplicación de forma dinámica. Esto nos permite crear experiencias de usuario más interactivas y personalizadas.

8.4 Conclusión.

La navegación en Next.js se realiza a través de la librería **next/router** y el componente **Link**. Estas herramientas nos permiten crear enlaces entre las diferentes páginas de nuestra aplicación y navegar de forma programática. Conocer cómo funciona la navegación en Next.js es fundamental para crear aplicaciones web modernas y eficientes.

8.5 Ejercicio.

Crea un enlace en la página de inicio que te lleve a la página **About** y un botón en la página **About** que te lleve a la página de inicio. Utiliza tanto el componente **Link** como la librería **next/router** para realizar la navegación entre las páginas de la aplicación.

9 Manejo de parámetros en rutas en Next.js

En este capítulo se presentará cómo manejar parámetros en las rutas de Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

9.1 Parámetros de ruta.

Los parámetros de ruta en Next.js se utilizan para pasar información a una ruta a través de la URL. Por ejemplo, si se desea mostrar el detalle de un producto en una tienda en línea, se puede utilizar un parámetro de ruta para pasar el ID del producto a la ruta.

Ejemplo:

```
// pages/product/[id].js
import { useRouter } from 'next/router'

const Product = () => {
  const router = useRouter()
  const { id } = router.query

  return (
    <div>
      <h1>Product Detail</h1>
      <p>Product ID: {id}</p>
    </div>
  )
}

export default Product
```

En el ejemplo anterior, se crea una ruta dinámica `/product/[id]` que recibe un parámetro `id` a través de la URL. Al acceder a la ruta `/product/123`, se mostrará el detalle del producto con el ID `123`.

9.2 Parámetros de consulta (query params).

Los parámetros de consulta en Next.js se utilizan para pasar información a una ruta a través de la URL utilizando el signo de interrogación `?`. Por ejemplo, si se desea filtrar una lista de productos por categoría, se puede utilizar un parámetro de consulta para pasar la categoría a la ruta.

Ejemplo:

```
// pages/products.js
import { useRouter } from 'next/router'

const Products = () => {
  const router = useRouter()
  const { category } = router.query

  return (
    <div>
      <h1>Products</h1>
      <p>Category: {category}</p>
    </div>
  )
}

export default Products
```

En el ejemplo anterior, se crea una ruta **/products** que recibe un parámetro de consulta **category** a través de la URL. Al acceder a la ruta **/products?category=electronics**, se mostrarán los productos de la categoría **electronics**.

9.3 Uso de useRouter para acceder a los parámetros.

La librería **next/router** proporciona el hook **useRouter** para acceder a los parámetros de ruta y de consulta en una página de Next.js. Este hook nos permite obtener los parámetros de la URL y utilizarlos en la página.

Ejemplo:

```
import { useRouter } from 'next/router'

const Product = () => {
  const router = useRouter()
  const { id } = router.query

  return (
    <div>
      <h1>Product Detail</h1>
      <p>Product ID: {id}</p>
    </div>
  )
}

export default Product
```

En el ejemplo anterior, se importa el hook **useRouter** de la librería **next/router** y se utiliza para acceder al parámetro **id** de la URL. Este parámetro se puede utilizar en la página para mostrar el detalle del producto.

9.4 Conclusión.

En este capítulo se presentó cómo manejar parámetros en las rutas de Next.js. A lo largo de este capítulo se presentaron los siguientes temas: parámetros de ruta, parámetros de consulta (query params) y uso de **useRouter** para acceder a los parámetros. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo manejar parámetros en las rutas de Next.js.

9.5 Ejercicios.

1. Crea una ruta dinámica en Next.js que reciba un parámetro de ruta.
2. Crea una ruta en Next.js que reciba un parámetro de consulta y muestre el valor en la página.
3. Utiliza el hook **useRouter** para acceder a los parámetros de ruta y de consulta en una página de Next.js.
4. Crea una aplicación en Next.js que utilice parámetros de ruta y de consulta en diferentes páginas.
5. Experimenta con diferentes formas de pasar parámetros a las rutas en Next.js y observa cómo se comporta la aplicación.
6. Investiga cómo validar los parámetros de ruta y de consulta en Next.js y aplica la validación en tu aplicación.
7. Comparte tus experiencias y aprendizajes sobre cómo manejar parámetros en las rutas de Next.js con tus compañeros de clase.
8. Investiga cómo manejar parámetros en las rutas de Next.js utilizando la librería **next/router** y comparte tus hallazgos con tus compañeros de clase.

10 React Server Components en Next.js: notación “use Client”

En este capítulo se presentarán los conceptos básicos de cómo utilizar React Server Components en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

10.1 Introducción a React Server Components.

React Server Components es una nueva característica de React que permite renderizar componentes en el servidor y enviar solo los datos necesarios al cliente. Esto mejora el rendimiento de la aplicación al reducir la cantidad de datos que se envían al cliente.

10.1.1 ¿Qué es use Client?

La notación “use Client” en React Server Components se utiliza para indicar que un componente se renderizará en el cliente en lugar de en el servidor. Esto permite que el componente se actualice de forma dinámica en el cliente sin tener que volver a renderizarlo en el servidor.

10.2 Uso de la notación “use Client”.

La notación “use Client” se utiliza en un componente de React Server Components para indicar que el componente se renderizará en el cliente. Por ejemplo:

```
import { useClient } from 'react-server-components'

const MyComponent = () => {
  const data = useClient(fetchData)

  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.content}</p>
    </div>
  )
}

export default MyComponent
```


En el ejemplo anterior, el componente **MyComponent** utiliza la notación “use Client” para indicar que se renderizará en el cliente. El componente llama a la función **fetchData** para obtener los datos necesarios y los muestra en la interfaz de usuario.

10.3 Ventajas de la notación “use Client”.

La notación “use Client” en React Server Components ofrece las siguientes ventajas:

- Permite renderizar componentes en el cliente de forma dinámica.
- Reduce la cantidad de datos que se envían al cliente.
- Mejora el rendimiento de la aplicación al evitar renderizaciones innecesarias en el servidor.

10.4 Ejemplos prácticos.

1. Crea un componente de React Server Components que utilice la notación “use Client”.

```
import { useClient } from 'react-server-components'

const MyComponent = () => {
  const data = useClient(fetchData)

  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.content}</p>
    </div>
  )
}

export default MyComponent
```

2. Utiliza la notación “use Client” en un componente de React Server Components para renderizar datos dinámicamente en el cliente.

```
import { useClient } from 'react-server-components'

const MyComponent = () => {
  const data = useClient(fetchData)

  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.content}</p>
    </div>
  )
}
```

```
)  
}  
  
export default MyComponent
```

10.5 Conclusión.

En este capítulo se presentaron los conceptos básicos de cómo utilizar React Server Components en Next.js. A lo largo de este capítulo se presentaron los siguientes temas: introducción a React Server Components, notación “use Client”, uso de la notación “use Client” y ventajas de la notación “use Client”. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo utilizar React Server Components en Next.js.

10.6 Ejercicios.

1. Crea un componente de React Server Components que utilice la notación “use Client”.
2. Utiliza la notación “use Client” en un componente de React Server Components para renderizar datos dinámicamente en el cliente.
3. Crea una aplicación en Next.js que utilice React Server Components y la notación “use Client” en diferentes componentes.

Espero que estos ejercicios le sean de utilidad y le ayuden a practicar cómo utilizar React Server Components en Next.js.

Part II

Unidad 2: Manejo de Estilos y Archivos Estáticos

11 Manejo de estilos y archivos estáticos en Next.js

En este capítulo se presentarán los conceptos básicos de cómo manejar estilos y archivos estáticos en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

11.1 Estilos en Next.js

Los estilos en Next.js se pueden manejar de diferentes formas, como CSS global, CSS Modules y Styled JSX. Cada una de estas formas tiene sus propias ventajas y desventajas, y se pueden utilizar según las necesidades del proyecto.

11.1.1 CSS global

El CSS global en Next.js se puede utilizar para aplicar estilos a toda la aplicación. Para utilizar CSS global en Next.js, se puede crear un archivo **styles.css** en la carpeta **public** y enlazarlo en el archivo ****_app.js****.

```
/* styles.css */

body {
  font-family: 'Arial', sans-serif;
  background-color: #f0f0f0;
}
```

```
// _app.js
import '../public/styles.css'

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

11.2 CSS Modules en Next.js

11.2.1 CSS Modules

Los CSS Modules en Next.js permiten crear estilos locales para cada componente. Para utilizar CSS Modules en Next.js, se puede crear un archivo **styles.module.css** en la carpeta del componente y importarlo en el archivo del componente.

```
/* Button.module.css */
```

```
.button {  
  background-color: #007bff;  
  color: #fff;  
  padding: 10px 20px;  
  border: none;  
  border-radius: 5px;  
}
```

```
// Button.js
```

```
import styles from './Button.module.css'
```

```
const Button = () => {  
  return <button className={styles.button}>Click me</button>  
}
```

```
export default Button
```

11.2.2 Scoped CSS en Next.js

En Next.js, los estilos se aplican de forma local por defecto, lo que significa que los estilos de un componente no afectan a otros componentes. Esto se conoce como scoped CSS y ayuda a evitar conflictos de estilos entre componentes.

11.2.3 Creación y uso de CSS Modules.

Para crear y utilizar CSS Modules en Next.js, se deben seguir los siguientes pasos:

1. Crear un archivo **styles.module.css** en la carpeta del componente.
2. Definir los estilos en el archivo **styles.module.css** utilizando la sintaxis de CSS.
3. Importar los estilos en el archivo del componente y utilizar la clase generada por CSS Modules.

```
/* styles.module.css */
```

```
.button {  
  background-color: #007bff;  
  color: #fff;  
  padding: 10px 20px;  
  border: none;  
  border-radius: 5px;  
}
```

```
// Button.js
```

```
import styles from './styles.module.css'
```

```
const Button = () => {  
  return <button className={styles.button}>Click me</button>  
}
```

```
export default Button
```

11.3 Styled JSX en Next.js

11.3.1 Styled JSX

Styled JSX en Next.js permite escribir estilos en línea dentro de los componentes. Para utilizar Styled JSX en Next.js, se puede utilizar la etiqueta **style** y definir los estilos dentro de ella.

```
// Button.js
```

```
const Button = () => {  
  return (  
    <button>  
      Click me  
      <style jsx>{`  
        button {  
          background-color: #007bff;  
          color: #fff;  
          padding: 10px 20px;  
          border: none;  
          border-radius: 5px;  
        }  
      `}</style>  
    </button>  
  )  
}
```

```
export default Button
```

11.3.2 Ventajas de Styled JSX

Las ventajas de Styled JSX en Next.js son las siguientes:

- Permite escribir estilos en línea de forma sencilla.
- Permite utilizar variables y funciones de JavaScript en los estilos.
- Permite aplicar estilos de forma local a un componente.

11.4 Conclusión

En este capítulo se presentaron los conceptos básicos de cómo manejar estilos y archivos estáticos en Next.js. A lo largo de este capítulo se presentaron las diferentes formas de manejar estilos en Next.js, como CSS global, CSS Modules y Styled JSX. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo manejar estilos y archivos estáticos en Next.js.

11.5 Ejercicios

1. Crea un archivo **styles.css** en la carpeta **public** y enlázalo en el archivo ****_app.js**** para aplicar estilos globales a la aplicación.
2. Crea un archivo **styles.module.css** en la carpeta de un componente y utilízalo para aplicar estilos locales al componente.
3. Utiliza Styled JSX en un componente de Next.js para aplicar estilos en línea al componente.

Espero que estos ejercicios le sean de utilidad y le ayuden a practicar cómo manejar estilos y archivos estáticos en Next.js.

12 Uso de Sass en Next.js

En este capítulo se presentará cómo utilizar Sass en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

12.1 Instalación y configuración de Sass.

Para utilizar Sass en Next.js, se debe instalar la librería **sass** y configurarla en el archivo **next.config.js**. Para instalar la librería **sass**, se puede utilizar el siguiente comando:

```
npm install sass
```

Para configurar Sass en Next.js, se debe crear un archivo **next.config.js** en la raíz del proyecto y agregar la siguiente configuración:

```
// next.config.js
const path = require('path')

module.exports = {
  sassOptions: {
    includePaths: [path.join(__dirname, 'styles')],
  },
}
```

En el ejemplo anterior, se configura Sass en Next.js para que pueda importar archivos Sass desde la carpeta **styles**.

12.2 Uso de Sass en componentes.

Para utilizar Sass en los componentes de Next.js, se puede crear un archivo **styles.scss** en la carpeta del componente y importarlo en el archivo del componente.

```
// Button.module.scss

.button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  border: none;
```



```
border-radius: 5px;
}
```

```
// Button.js
import styles from './Button.module.scss'

const Button = () => {
  return <button className={styles.button}>Click me</button>
}

export default Button
```

En el ejemplo anterior, se crea un archivo **styles.scss** con estilos Sass para el componente **Button** y se importa en el archivo del componente.

12.3 Variables y mixins en Sass.

Sass permite utilizar variables y mixins para reutilizar estilos en los componentes de Next.js. Para utilizar variables y mixins en Sass, se puede crear un archivo **styles.scss** con las variables y mixins necesarias y importarlo en los componentes.

```
// styles.scss

$primary-color: #007bff;

@mixin button-styles {
  background-color: $primary-color;
  color: #fff;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
}
```

```
// Button.module.scss
@import 'styles.scss';

.button {
  @include button-styles;
}
```

En el ejemplo anterior, se crea un archivo **styles.scss** con variables y mixins para los estilos del componente **Button** y se importa en el archivo **Button.module.scss**.

12.4 Conclusión.

En este capítulo se presentó cómo utilizar Sass en Next.js. A lo largo de este capítulo se presentaron los siguientes temas: instalación y configuración de Sass, uso de Sass en componentes, variables y mixins en Sass. Espero que

12.5 Ejercicios.

1. Crea un archivo **styles.scss** con estilos Sass para un componente en Next.js.
2. Crea variables y mixins en Sass para reutilizar estilos en los componentes de Next.js.
3. Importa un archivo Sass en un componente de Next.js y aplica los estilos en el componente.

13 Cómo utilizar estilos globales en Next.js

En este capítulo se presentarán los conceptos básicos de cómo utilizar estilos globales en Next.js. A lo largo de este capítulo se presentarán los siguientes temas:

13.1 Estilos globales vs. estilos locales.

Los estilos globales en Next.js se pueden utilizar para aplicar estilos a toda la aplicación, mientras que los estilos locales se utilizan para aplicar estilos a componentes específicos.

Ejemplo:

```
/* styles.css */

body {
  font-family: 'Arial', sans-serif;
  background-color: #f0f0f0;
}

.button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
}
```

En el código anterior, se crea un archivo **styles.css** con estilos globales para la aplicación. Los estilos se aplican al cuerpo de la página y al componente **Button**.

13.2 Implementación de estilos globales.

Para utilizar estilos globales en Next.js, se puede crear un archivo **styles.css** en la carpeta **public** y enlazarlo en el archivo ****_app.js****.

```
// _app.js
import '../public/styles.css'

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

```
}  
  
export default MyApp
```

13.3 Uso de reset y normalize.css.

Para restablecer los estilos predeterminados del navegador y garantizar una apariencia consistente en diferentes navegadores, se pueden utilizar las bibliotecas **reset.css** o **normalize.css**.

Ejemplo:

```
// _app.js  
import 'normalize.css'  
import '../public/styles.css'  
  
function MyApp({ Component, pageProps }) {  
  return <Component {...pageProps} />  
}  
  
export default MyApp
```

En el ejemplo anterior, se importa la biblioteca **normalize.css** en el archivo ****_app.js**** para restablecer los estilos predeterminados del navegador.

13.4 Conclusión.

En este capítulo se presentaron los conceptos básicos de cómo utilizar estilos globales en Next.js. A lo largo de este capítulo se presentaron los siguientes temas: estilos globales vs. estilos locales, implementación de estilos globales, uso de reset y normalize.css. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo utilizar estilos globales en Next.js.

13.5 Ejercicios.

1. Crea un archivo **styles.css** con estilos globales para tu aplicación Next.js.
2. Utiliza la biblioteca **normalize.css** para restablecer los estilos predeterminados del navegador en tu aplicación Next.js.
3. Crea un componente **Button** con estilos globales y locales en tu aplicación Next.js.

14 Cómo agregar archivos estáticos en Next.js

14.1 Carpeta public.

Los archivos estáticos en Next.js se pueden agregar en la carpeta **public**. Esta carpeta se utiliza para almacenar archivos estáticos como imágenes, fuentes, hojas de estilo y scripts.

Ejemplo:

```
public/  
  images/  
    logo.png  
  styles/  
    styles.css
```

En el ejemplo anterior, se crean las carpetas **images** y **styles** en la carpeta **public** para almacenar archivos estáticos como imágenes y hojas de estilo.

14.2 Acceso y uso de archivos estáticos.

Los archivos estáticos en la carpeta **public** se pueden acceder y utilizar en los componentes de Next.js utilizando la ruta relativa a la carpeta **public**.

Ejemplo:

```
// Image.js  
  
const Image = () => {  
  return ;  
}  
  
export default Image;
```

En el ejemplo anterior, se crea un componente **Image** que muestra la imagen **logo.png** almacenada en la carpeta **images** de la carpeta **public**.

14.3 Ventajas de la carpeta **public**.

Las ventajas de utilizar la carpeta **public** en Next.js son las siguientes:

- Permite almacenar archivos estáticos de forma organizada.
- Permite acceder y utilizar archivos estáticos en los componentes de forma sencilla.
- Permite mejorar el rendimiento de la aplicación al almacenar archivos estáticos en caché.

14.4 Ejemplos prácticos.

1. Crea una carpeta **public** en la raíz del proyecto y agrega archivos estáticos como imágenes, fuentes, hojas de estilo y scripts.

Ejemplo:

```
public/  
  images/  
    logo.png  
  styles/  
    styles.css
```

En el ejemplo anterior, se crea la carpeta **public** en la raíz del proyecto y se agregan archivos estáticos como la imagen **logo.png** y la hoja de estilo **styles.css**.

2. Accede y utiliza los archivos estáticos en los componentes de Next.js utilizando la ruta relativa a la carpeta **public**.

Ejemplo:

```
// Image.js  
  
const Image = () => {  
  return ;  
}  
  
export default Image;
```

En el ejemplo anterior, se crea un componente **Image** que muestra la imagen **logo.png** almacenada en la carpeta **images** de la carpeta **public**.

3. Comprueba que los archivos estáticos se cargan correctamente en la aplicación y se muestran en los componentes.

Ejemplo:

```
// App.js

import Image from './components/Image';

const App = () => {
  return (
    <div>
      <h1>Static Files in Next.js</h1>
      <Image />
    </div>
  );
}

export default App;
```

En el ejemplo anterior, se crea un componente **App** que muestra el título de la aplicación y el componente **Image** que muestra la imagen **logo.png** almacenada en la carpeta **images** de la carpeta **public**.

14.5 Conclusión.

En este capítulo se presentaron los conceptos básicos de cómo agregar archivos estáticos en Next.js. A lo largo de este capítulo se presentaron las ventajas de utilizar la carpeta **public** para almacenar archivos estáticos y cómo acceder y utilizar los archivos estáticos en los componentes de Next.js. Espero que este capítulo le sea de utilidad y le ayude a comprender mejor cómo agregar archivos estáticos en Next.js.

14.6 Ejercicios.

1. Crea una carpeta **public** en la raíz del proyecto y agrega archivos estáticos como imágenes, fuentes, hojas de estilo y scripts.
2. Accede y utiliza los archivos estáticos en los componentes de Next.js utilizando la ruta relativa a la carpeta **public**.
3. Comprueba que los archivos estáticos se cargan correctamente en la aplicación y se muestran en los componentes.

15 Manejo y optimización de imágenes con Next Image

Las imágenes son un elemento importante en el desarrollo web, ya que pueden mejorar la apariencia y la experiencia del usuario. En Next.js, se puede utilizar el componente **Image** para mostrar imágenes de forma eficiente y optimizada. En esta unidad, se explorará cómo manejar y optimizar imágenes con Next Image.

15.1 Introducción al componente Image.

El componente **Image** en Next.js se utiliza para mostrar imágenes de forma eficiente y optimizada. El componente **Image** utiliza el formato de imagen WebP y la carga perezosa para mejorar el rendimiento de la aplicación.

Ejemplo:

```
// Image.js

import Image from 'next/image';

const MyImage = () => {
  return <Image src="/images/my-image.jpg" alt="My Image" width={500} height={300} />;
}

export default MyImage;
```

En el ejemplo anterior, se crea un componente **MyImage** que muestra la imagen **my-image.jpg** con un ancho de 500 píxeles y una altura de 300 píxeles utilizando el componente **Image**.

15.2 Configuración y optimización de imágenes.

Para configurar y optimizar las imágenes en Next.js, se pueden utilizar las siguientes propiedades del componente **Image**:

- **src**: La ruta de la imagen.
- **alt**: El texto alternativo de la imagen.
- **width**: El ancho de la imagen en píxeles.

- **height**: La altura de la imagen en píxeles.
- **layout**: La disposición de la imagen (responsive, fixed, fill, intrinsic).
- **objectFit**: La forma en que la imagen se ajusta al contenedor.
- **objectPosition**: La posición de la imagen dentro del contenedor.

Ejemplo:

```
// Image.js

import Image from 'next/image';

const MyImage = () => {
  return <Image src="/images/my-image.jpg" alt="My Image" width={500} height={300} layout="fill" objectFit="cover" objectPosition="center" />;
}

export default MyImage;
```

En el ejemplo anterior, se configura la imagen **my-image.jpg** con un ancho de 500 píxeles, una altura de 300 píxeles, una disposición responsive, un ajuste de objeto de cubierta y una posición de objeto centrada.

15.3 Uso de imágenes remotas y locales.

En Next.js, se pueden utilizar imágenes remotas y locales en el componente **Image**. Las imágenes remotas se pueden cargar desde una URL externa, mientras que las imágenes locales se pueden cargar desde la carpeta **public**.

Ejemplo:

```
// Image.js

import Image from 'next/image';

const MyImage = () => {
  return (
    <div>
      <Image src="https://example.com/my-remote-image.jpg" alt="My Remote Image" width={500} height={300} />
      <Image src="/images/my-local-image.jpg" alt="My Local Image" width={500} height={300} />
    </div>
  );
}

export default MyImage;
```

En el ejemplo anterior, se muestra una imagen remota **my-remote-image.jpg** y una imagen local **my-local-image.jpg** utilizando el componente **Image**.

15.4 Conclusión.

El componente **Image** en Next.js permite mostrar imágenes de forma eficiente y optimizada en la aplicación. Al utilizar el componente **Image**, se puede mejorar el rendimiento de la aplicación y proporcionar una mejor experiencia al usuario al cargar imágenes de manera rápida y eficiente.

En esta unidad, se exploró cómo manejar y optimizar imágenes con Next Image, incluyendo la configuración de propiedades, el uso de imágenes remotas y locales, y la importancia de utilizar el componente **Image** para mejorar el rendimiento de la aplicación.

16 Optimización de fuentes con Next.js

En Next.js, se pueden optimizar las fuentes para mejorar el rendimiento y la carga de la aplicación. Para optimizar las fuentes en Next.js, se pueden utilizar las siguientes técnicas:

16.1 Uso de fuentes personalizadas.

Para utilizar fuentes personalizadas en Next.js, se pueden agregar las fuentes en la carpeta **public** y enlazarlas en el archivo ****_app.js****.

Ejemplo:

```
// _app.js

import '../public/fonts.css'

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

En el ejemplo anterior, se enlaza el archivo **fonts.css** que contiene las fuentes personalizadas en el archivo ****_app.js****.

16.2 Optimización de carga de fuentes.

Para optimizar la carga de fuentes en Next.js, se pueden utilizar las propiedades **preload** y **prefetch** en el archivo ****_document.js****.

Ejemplo:

```
// _document.js

import Document, { Html, Head, Main, NextScript } from 'next/document'

class MyDocument extends Document {
  render() {
    return (
```

```

    <Html>
      <Head>
        <link rel="preload" href="/fonts/my-font.woff2" as="font" type="font/woff2" cro
        <link rel="prefetch" href="/fonts/my-font.woff2" as="font" type="font/woff2" cr
      </Head>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
}

export default MyDocument

```

En el ejemplo anterior, se utiliza la propiedad **preload** para cargar la fuente **my-font.woff2** de forma anticipada y la propiedad **prefetch** para cargar la fuente **my-font.woff2** de forma diferida.

16.3 Ejemplos prácticos.

1. Crea una carpeta **fonts** en la carpeta **public** y agrega las fuentes personalizadas en formato WOFF2.

```

public/
  fonts/
    my-font.woff2

```

2. Crea un archivo **fonts.css** en la carpeta **public** y enlaza las fuentes personalizadas.

```

/* fonts.css */

@font-face {
  font-family: 'My Font';
  src: url('/fonts/my-font.woff2') format('woff2');
}

```

3. Enlaza el archivo **fonts.css** en el archivo ****_app.js**** para utilizar las fuentes personalizadas en la aplicación.

```

// _app.js

import '../public/fonts.css'

function MyApp({ Component, pageProps }) {

```

```

    return <Component {...pageProps} />
  }

export default MyApp

```

4. Optimiza la carga de las fuentes en Next.js utilizando las propiedades **preload** y **prefetch** en el archivo ****_document.js****.

```

// _document.js

import Document, { Html, Head, Main, NextScript } from 'next/document'

class MyDocument extends Document {
  render() {
    return (
      <Html>
        <Head>
          <link rel="preload" href="/fonts/my-font.woff2" as="font" type="font/woff2" crossOrigin="anonymous" />
          <link rel="prefetch" href="/fonts/my-font.woff2" as="font" type="font/woff2" crossOrigin="anonymous" />
        </Head>
        <body>
          <Main />
          <NextScript />
        </body>
      </Html>
    )
  }
}

export default MyDocument

```

En el ejemplo anterior, se optimiza la carga de la fuente **my-font.woff2** utilizando las propiedades **preload** y **prefetch** en el archivo ****_document.js****.

16.4 Conclusión.

La optimización de fuentes en Next.js es importante para mejorar el rendimiento y la carga de la aplicación. Al utilizar fuentes personalizadas y optimizar su carga, se puede garantizar una mejor experiencia de usuario y una mayor eficiencia en la aplicación.

16.5 Ejercicios

1. Agrega una fuente personalizada a tu aplicación Next.js y enlázala en el archivo ****_app.js****.

2. Optimiza la carga de la fuente personalizada utilizando las propiedades **preload** y **prefetch** en el archivo `**_document.js**`.
3. Comprueba que la fuente personalizada se carga correctamente en la aplicación y se muestra en los componentes.

17 Creando estilos dinámicos aplicando condicionales en Next.js

En Next.js, se pueden crear estilos dinámicos aplicando condicionales en los componentes para cambiar la apariencia de los elementos según ciertas condiciones. Esto permite personalizar la apariencia de los componentes de forma dinámica en función de la lógica de la aplicación.

17.1 Estilos dinámicos.

Los estilos dinámicos en Next.js se pueden aplicar utilizando condicionales en los estilos de los componentes. Esto permite cambiar la apariencia de los elementos en función de ciertas condiciones, como el estado de un componente o los datos recibidos de una API.

Ejemplo:

```
// Button.js

const Button = ({ primary }) => {
  return (
    <button style={{ backgroundColor: primary ? '#007bff' : '#f0f0f0', color: primary ? 'white' : 'black' }}>
      Click me
    </button>
  );
}

export default Button;
```

En el ejemplo anterior, se crea un componente **Button** con un estilo dinámico que cambia el color de fondo y el color del texto en función de la prop **primary**. Si la prop **primary** es verdadera, se aplican los estilos para un botón principal; de lo contrario, se aplican los estilos para un botón secundario.

17.2 Aplicación de condicionales en estilos.

Para aplicar condicionales en los estilos de los componentes en Next.js, se pueden utilizar operadores ternarios o funciones condicionales para determinar los estilos a aplicar en función de ciertas condiciones.

Ejemplo:

```
// Button.js

const Button = ({ primary }) => {
  const buttonStyles = {
    backgroundColor: primary ? '#007bff' : '#f0f0f0',
    color: primary ? '#fff' : '#000',
    padding: '10px 20px',
    border: 'none',
    borderRadius: '5px',
  };

  return (
    <button style={buttonStyles}>
      Click me
    </button>
  );
}

export default Button;
```

En el ejemplo anterior, se crea un objeto **buttonStyles** con los estilos del botón y se utiliza un operador ternario para aplicar los estilos en función de la prop **primary**. Esto permite personalizar la apariencia del botón de forma dinámica en función de la prop recibida.

17.3 Ejemplos prácticos.

1. Crea un componente **Button** con estilos dinámicos que cambien la apariencia del botón en función de una prop **primary**.

Ejemplo:

```
// Button.js

const Button = ({ primary }) => {
  const buttonStyles = {
    backgroundColor: primary ? '#007bff' : '#f0f0f0',
    color: primary ? '#fff' : '#000',
    padding: '10px 20px',
    border: 'none',
    borderRadius: '5px',
  };

  return (
    <button style={buttonStyles}>
      Click me
    </button>
  );
}
```



```

    </button>
  );
}

export default Button;

```

En el ejemplo anterior, se crea un componente **Button** con estilos dinámicos que cambian la apariencia del botón en función de la prop **primary**. Si la prop **primary** es verdadera, se aplican los estilos para un botón principal; de lo contrario, se aplican los estilos para un botón secundario.

2. Utiliza el componente **Button** en otros componentes de la aplicación y prueba los estilos dinámicos cambiando el valor de la prop **primary**.

Ejemplo:

```

// App.js

import Button from './components/Button';

const App = () => {
  return (
    <div>
      <h1>Dynamic Styles in Next.js</h1>
      <Button primary>Primary Button</Button>
      <Button>Secondary Button</Button>
    </div>
  );
}

export default App;

```

En el ejemplo anterior, se importa el componente **Button** en el componente **App** y se utiliza el componente con diferentes valores de la prop **primary** para probar los estilos dinámicos. El botón se mostrará con estilos diferentes en función del valor de la prop **primary**.

17.4 Conclusión.

En Next.js, se pueden crear estilos dinámicos aplicando condicionales en los componentes para cambiar la apariencia de los elementos en función de ciertas condiciones. Esto permite personalizar la apariencia de los componentes de forma dinámica y adaptativa, lo que mejora la experiencia del usuario y la interactividad de la aplicación.

Los estilos dinámicos son una herramienta poderosa para crear interfaces atractivas y funcionales en Next.js, ya que permiten adaptar la apariencia de los elementos según el contexto y la interacción del usuario. Al aplicar condicionales en los estilos de los

componentes, se pueden crear interfaces más dinámicas y personalizadas que se ajusten a las necesidades y preferencias de los usuarios.

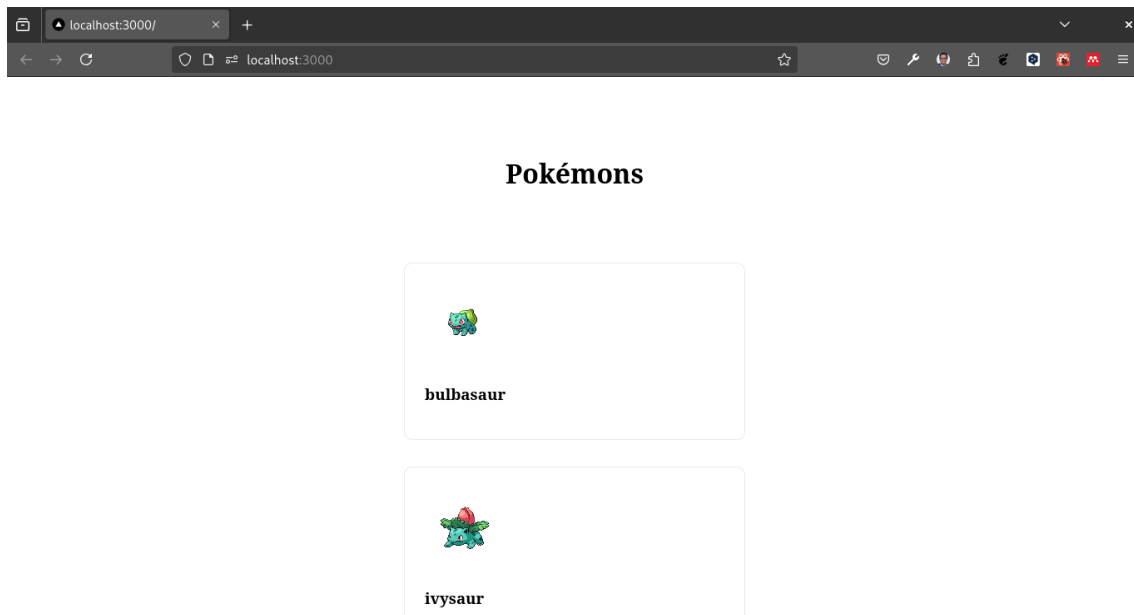
17.5 Ejercicio.

Crea un componente **Card** con estilos dinámicos que cambien el color de fondo y el color del texto en función de una prop **variant**. Utiliza diferentes valores de la prop **variant** para probar los estilos dinámicos y observa cómo cambia la apariencia de la tarjeta en función de la variante seleccionada.

Part III

Laboratorios

18 Laboratorio de Fetch con Next



En este laboratorio crearemos una aplicación de Next.js que obtiene datos de una API de Pokemons y los muestra en una lista.

18.1 Pasos

1. Crear una nueva aplicación de Next.js

```
npx create-next-app@latest 1_laboratorio_fetch
```

2. Crear un archivo llamado **lib/getsPokemons.tsx** en la raíz del proyecto.

```
export async function getPokemons() {
  const headers = new Headers({
    "Content-Type": "application/json"
  });

  const requestOptions = {
    method: 'GET',
    headers: headers,
    redirect: 'follow' as RequestRedirect
  };
}
```

```

try {
  const response = await fetch("https://pokeapi.co/api/v2/pokemon?limit=10", requestOptions);
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  const data = await response.json();
  const pokemonDetails = await Promise.all(
    data.results.map(async (pokemon: any) => {
      const res = await fetch(pokemon.url);
      const details = await res.json();
      return details;
    })
  );
  return pokemonDetails;
} catch (error) {
  console.error('Failed to fetch pokemons:', error);
  return [];
}
}

```

En el archivo mencionado, se ha creado una función llamada `getPokemons`. Esta función tiene la responsabilidad de realizar una petición a la API de Pokemons y obtener los primeros 10 pokemons.

La API de Pokemons es una interfaz de programación de aplicaciones que permite acceder a información relacionada con los pokemons, como sus nombres, habilidades, tipos, entre otros datos. Al realizar una petición a esta API, podemos obtener los datos de los pokemons de manera programática.

En el código proporcionado, la función `getPokemons` se encarga de manejar los posibles errores que puedan ocurrir durante la petición a la API. Si ocurre algún error, en lugar de lanzar una excepción o detener la ejecución del programa, la función retorna un arreglo vacío.

Esta estrategia de manejo de errores es útil porque permite que el programa continúe ejecutándose sin interrupciones, incluso si la petición a la API falla. En lugar de detenerse por completo, el programa puede tomar medidas alternativas o mostrar un mensaje de error al usuario.

En resumen, la función `getPokemons` es responsable de obtener los primeros 10 pokemons de la API de Pokemons y manejar los errores de manera adecuada. Esto garantiza que el programa pueda continuar ejecutándose sin problemas, incluso en caso de fallos en la petición.

3. Crear un archivo llamado **pages/index.tsx** en la raíz del proyecto.

```

import styles from '../styles/Home.module.css'; //<1>
import { useEffect, useState } from 'react'; //<2>
import { getPokemons } from '../lib/getPokemons'; //<3>

```

```

export default function Home() { //<4>
  const [pokemons, setPokemons] = useState<Array<any>>([]); //<5>

  useEffect(() => { //<6>
    const fetchPokemons = async () => { //<7>
      console.log('Fetching pokemons...'); //<8>
      const pokemonData = await getPokemons(); //<9>
      console.log('Fetched pokemons:', pokemonData); //<10>
      setPokemons(pokemonData); //<11>
    };

    fetchPokemons(); //<12>
  }, []); //<13>

  return (
    <div className={styles.container}> //<14>
      <main className={styles.main}> //<15>
        <h1>Pokémons</h1> //<16>
        <div className={styles.grid}> //<17>
          {pokemons.map(pokemon => ( //<18>
            <div key={pokemon.id} className={styles.card}> //<19>
              <img src={pokemon.sprites.front_default} alt={pokemon.name} /> //<20>
              <h3>{pokemon.name}</h3> //<21>
            </div> //<22>
          ))}
        </div>
      </main>
    </div>
  );
}

```

En el archivo anterior creamos un componente llamado **Home** que muestra una lista de los primeros 10 pokemons obtenidos de la API de Pokemons. Este componente es una página de la aplicación Next.js y se encuentra en el archivo **pages/index.tsx**.

El componente **Home** utiliza el hook **useEffect** para realizar una petición a la API de Pokemons cuando se monta. El hook **useEffect** se ejecuta después de que el componente se haya renderizado en el navegador. En este caso, se utiliza para obtener los datos de los pokemons y actualizar el estado del componente con la lista de pokemons obtenidos.

La función **fetchPokemons** es una función asíncrona que se define dentro del hook **useEffect**. Esta función se encarga de realizar la petición a la API de Pokemons utilizando la función **getPokemons** que hemos definido en el archivo **lib/getPokemons.tsx**. La función **getPokemons** devuelve una promesa que se resuelve con los datos de los pokemons obtenidos de la API.

Una vez que se obtienen los datos de los pokemons, se actualiza el estado del componente utilizando la función **setPokemons**. La función **setPokemons** es una función proporcionada por el hook **useState** que se utiliza para actualizar el valor de la variable de estado

pokemons. Al actualizar el estado, el componente se vuelve a renderizar y muestra la lista de pokemons en la interfaz de usuario.

La lista de pokemons se muestra utilizando el método **map** de JavaScript. El método **map** se utiliza para iterar sobre cada elemento de un arreglo y devolver un nuevo arreglo con los resultados de aplicar una función a cada elemento. En este caso, se utiliza el método **map** para iterar sobre la lista de pokemons y generar un elemento de la lista para cada pokemon.

Cada elemento de la lista de pokemons se representa como un ******

****** con una imagen y el nombre del pokemon. La imagen se muestra utilizando la propiedad **sprites.front_default** del objeto pokemon, que contiene la URL de la imagen del pokemon. El nombre del pokemon se muestra utilizando la propiedad **name** del objeto pokemon.

En resumen, el componente **Home** es responsable de obtener los datos de los primeros 10 pokemons de la API de Pokemons y mostrarlos en una lista en la interfaz de usuario. Utiliza el hook **useEffect** para realizar la petición a la API cuando se monta y el hook **useState** para almacenar y actualizar la lista de pokemons en el estado del componente. La lista de pokemons se muestra utilizando el método **map** y se representan como elementos ******

****** con una imagen y el nombre del pokemon.

4. Agregamos CSS

Creamos el archivo **styles/Home.module.css** en la raíz del proyecto con el siguiente contenido:

```
.container {
  padding: 0 2rem;
}

.main {
  min-height: 100vh;
  padding: 4rem 0;
  flex: 1;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}

.grid {
  display: flex;
  align-items: center;
  justify-content: center;
  flex-wrap: wrap;
  max-width: 800px;
  margin-top: 3rem;
}
```

```

}

.card {
  margin: 1rem;
  flex-basis: 45%;
  padding: 1.5rem;
  text-align: left;
  color: inherit;
  text-decoration: none;
  border: 1px solid #eaeaea;
  border-radius: 10px;
  transition: color 0.15s ease, border-color 0.15s ease;
}

.card:hover,
.card:focus,
.card:active {
  color: #0070f3;
  border-color: #0070f3;
}

.card img {
  max-width: 100%;
  border-radius: 10px;
}

```

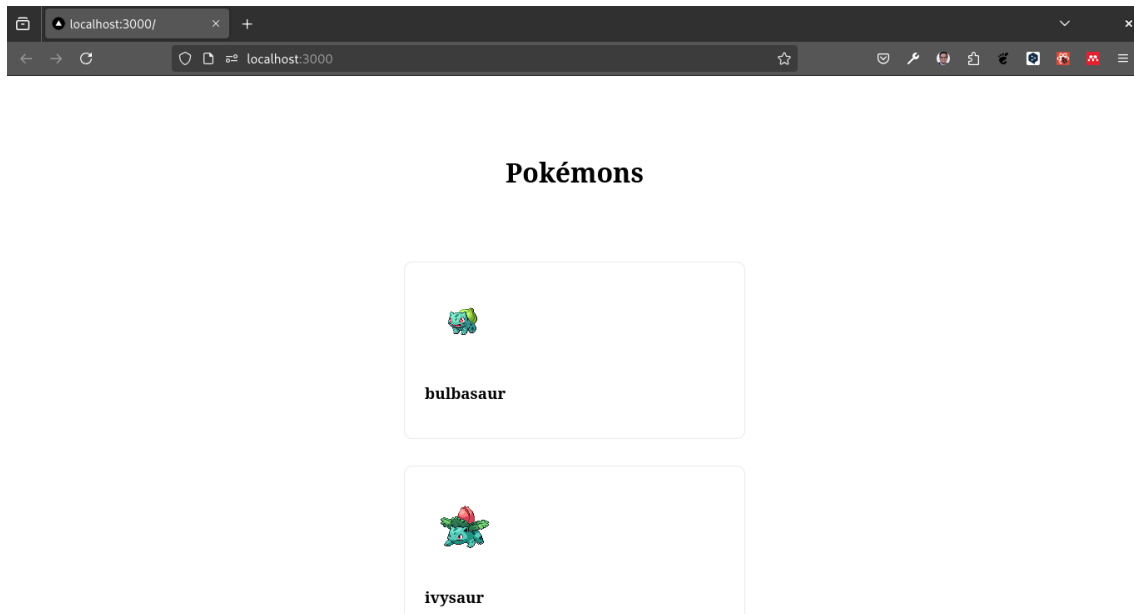
En este archivo se definen los estilos CSS para la aplicación. Los estilos se aplican a los elementos HTML utilizando clases CSS y se utilizan para dar formato y diseño a la interfaz de usuario.

5. Ejecutar la aplicación

```
npm run dev
```

6. Abrir la aplicación en el navegador <http://localhost:3000>

18.2 Resultado



Como puedes observar en la imagen anterior, la aplicación muestra una lista de los primeros 10 pokemons obtenidos de la API de Pokemons.

18.3 Conclusión

En este laboratorio, aprendimos cómo obtener datos de una API en una aplicación de Next.js y mostrarlos en una lista. También aprendimos cómo usar el hook **useEffect** para realizar una petición a la API cuando el componente se monta.

Además, vimos cómo manejar los posibles errores que puedan ocurrir durante la petición a la API utilizando un bloque try-catch. Esto nos permite manejar los errores de manera adecuada y evitar que la aplicación se detenga por completo en caso de fallos en la petición.

Para mostrar los datos de los pokemons en la lista, utilizamos el método **map** de JavaScript para iterar sobre la lista de pokemons y generar un elemento de la lista para cada pokemon. Cada elemento de la lista se representa como un **

** con una imagen y el nombre del pokemon.

En resumen, este laboratorio nos ha enseñado cómo obtener y mostrar datos de una API en una aplicación de Next.js, así como cómo manejar los errores de manera adecuada. Estos conceptos son fundamentales para el desarrollo de aplicaciones web que interactúan con servicios externos y proporcionan una experiencia de usuario fluida y sin interrupciones.

18.4 Ejercicio

Desarrollar una aplicación en Next.js que utilice la función `fetch` para obtener datos de una API externa de Pokemons y mostrarlos dinámicamente en una lista.

:::