Curso NEXt.js

# Instalando NextJS

Requisitos

* + Node (>12,>10)
  + Git
  + Browser - Goggle Chrome - para utilizar la DevTools
* Instalacion
  + CLI - Automatica:
* yarn **create** **next**-app
  + Manual
* **mkdir** platzi-nextjs
* cd platzi-nextjs
* npm init -y
* yarn add **next** react react-dom
* npm add **next** react react-dom
* Agregar el siguiente codigo al "script"
* "scripts": {
* "dev": "next",
* "build": "next build",
* "start": "next start"

},

Inicializar NPM

npm init -y

Instalar con NPM

npm **install** **next** react react-dom

Scripts

"dev": "next",

"build": "next build",

"start": "next start"

Creamos directorio **pages**

mkdir pages

Iniciar con NPM

npm run dev

Routing

# Rutas básicas

<h4>Ideas/conceptos claves</h4>

**Rutas Estáticas:** Son las que son sencillas

**Rutas Dinámicas:** Son aquellas que se generan de una forma consecutiva dependiendo de variables

<h4>Apuntes</h4>

Todo lo que esta relacionado con paginas y navegación se entiende con el nombre de routing. En Next.js ya cuenta con una solución para no preocuparnos del tema. Cuenta con dos soluciones:

**Rutas Estáticas**

/about/

**Rutas Dinámicas**

/user/platzi

Si se quiere hacer rutas dinámicas con parámetros, nombramos el archivo de esta forma:

[parametro].js

para leer el parámetro se puede usar el hook de Next.js

**import** { useRouter } **from** "next/router";

**const** ProductItem = () => {

**const** {

query: { productId },

} = useRouter();

**return** <div>Esta es la pagina del producto: {productId}</div>;

};

**export** **default** ProductItem;

Los cambios que ejecutamos vienen con Hot Reload y fast refresh. Es decir que se aplican rápidamente en en el navegador.

Cada vez que creamos un archivo en pages se agregan a las rutas de la aplicación. Esta manera de hacer rutas se denomina routing basado en file system

**RESUMEN:** Crear rutas en Next.js es bastante sencillo debido a que usa routing basado en file system



# #UnderTheHood setup y páginas: optimizaciones ocultas

h4>Ideas/conceptos claves</h4>

**chunk**.- pedazo de código

**CSR**.- Client Side Rendering

**SSR**.- Server Side Rendering

<h4>Apuntes</h4>

"scripts": {

"dev": "next", // desarrollo

"build": "next build", // build prod

"start": "next start" // server prod

},

Al momento de compilar separa el código con la técnica **code splitting**. Ademas de aplicar hashes a los assets. Ademas que para cada pagina generara un chunk especifico y mantendrá el código que se usara durante toda la aplicación

Podemos ver que las paginas generadas son SSR dándonos más SEO que con CSR

**RESUMEN:** Next.js aplica al bundle final diferentes optimizaciones como ser vendor files que hacen que no nos preocupemos de la configuración del proyecto. Ademas que nos da un mejor SEO por el server side rendering.

Una de las mejores cosas que trae next por defecto es este tipo de configuración para producción, tanto el manejo de assets como la generación de bundles y por sobre todo su sistema de SSR y Prefetching de los componentes de cada página.

Ah pero mira, Next.js es el APACHE de React. ❤️

# #UnderTheHood páginas: pre rendering de páginas

Viendo el código fuente se sabe que se hace un pre-render, en aplicaciones construidas con create-react-app solo se puede ver una etiqueta en el body:

**<div ="root></div>**

(Firefox) Cmd + U, se abre la previsualización del código fuente y se puede ver que Nextjs renderiza todo nuestro contenido.

El SSR (server side rendering), todo viene prerenderizado desde el servidor y permite que la web sea indexable por los motores de busqueda (google).

El SSR envia un html y css prerenderizado hacia el cliente, es decir, ejecuta el JavaScript que incrusta todos estos elementos en el DOM en el servidor y deja las tareas secundarias (como los eventos, fetch, etc) para que se ejecute en el lado del cliente. Por lo que si agregamos un botón a la página About que ejecute un Alert:

**import** React **from** 'react';

**const** About = () => {

**const** handleClick = () => {

alert('clicked!');

}

**return** (

<>

Esta es la pagina de about

<button onClick={handleClick}>Click me!</button>

</>

);

}

export default About;

Y deshabilitamos el JavaScript en el navegador, el botón no realizará acción alguna, pero sí se renderizará.

Hay varias maneras de revisar si el contenido viene prerenderizado del servidor. Podriamos hacer un curl a la dirección del servidor y luego examinar si el contenido que se muestra ya viene completo o si hay llamadas de javascript que completan el código html faltante.

**curl** **localhost**:3000 > **archivo**.html

Lo que me lleva a la segunda manera que es abriendo devtools y mirando la pestaña de network para ver examinar las peticiones web que se hacen posteriores al archivo principal.

En realidad lo que esta haciendo Next en este caso no es SSR, sino Static rendering, la diferencia es que en SSR cada vez que llegue una nueva consulta Next va a ejecutar el codigo y va a devolver la vista renderizada en html, en cambio en Static rendering Next solo va a ejecutar el codigo UNA sola vez (eso sucede en el momento en que hacemos deploy) y a partir de aqui cada vez que llegue una consulta nueva siempre va a devolver la misma vista html que renderizo la primera vez, esto es mucho mas eficiente ya que el codigo solo se renderiza en el servidor una sola vez y al generar los archivos estaticos estos pueden ser alojados y distribuidos en un CDN lo que los hacen increiblemente rapidos

Revisar

# Enlazando páginas

Next.JS reuiere que dentro del componente de Link se encuentre una etiqueta <a></a>, de forma que sea amigable para el SEO. Si no la agregas, de igual forma funciona, pero verás un warning de parte de Next.JS.

Pueden ver un poco más de información de como trabajar con Styled Components o Componentes que envuelven la etiqueta <a> aquí: [Next Link](https://nextjs.org/docs/api-reference/next/link)

**Anchor**.- Anclaje, nos sirve para hacer hipervínculos

<h4>Apuntes</h4>

Para poder hacer un enlace de paginas debemos importar el componente Link de Next.js, como propiedad se pasa un href el cual sera la ruta que deseemos ir y este debe contener una etiqueta anchor

**import** React **from** "react";

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

**const** Navbar = () => {

**return** (

<nav>

<menu>

<Link href="/">

<a>Home</a>

</Link>

<Link href="/about">

<a>About</a>

</Link>

</menu>

</nav>

);

};

**export** **default** Navbar;

**RESUMEN: Con la etiqueta Link de Next.js podemos direccionar las rutas de nuestra aplicación**

# #UnderTheHood enlazando páginas: prefetching automático

Next.js cuando usamos la etiqueta Link para nuestras rutas. Si hacemos hover o estamos encima de uno de los elementos lo que intentara hacer internamente sera pedir al servidor que prepare ese chunk de código.

**RESUMEN:** Next.js nos ayuda bastante con técnicas las cuales son bastante inteligentes, como en este caso el prefetching de paginas.

Sí, usar prefetch es recomendable para cargar páginas pesadas, es el mayor propósito. El navegador en últimas tomará la decisión de qué tanto guardar en el prefetch.

Qué hay que tener en cuenta para prefetch:

* Que si los archivos a hacer prefetch son muy pesados, es posible que nunca se realice el prefetch porque nunca acaba.

Es por eso que Next.JS usa un enfoque de múltiples archivos hacerlos portables y pequeños

Extendiendo NextJS

# Extendiendo el Document de nextjs

**document**.- Archivo principal de nuestra pagina donde los cambios se pueden aplicar a una escala global

**documentation de Next.js custom document:**

[Advanced Features: Custom Document | Next.js](https://nextjs.org/docs/advanced-features/custom-document)

<h4>Apuntes</h4>

La estructura de nuestro proyecto suele ser

* Nuestra Aplicación
  + Document
    - App

Normalmente solemos editar los primeros dos items

Para poder editarlo creamos un archivo en pages llamado “\_**document.js**” para usar un template definido podemos ir a la **documentation de Next.js para un custom document**

El código quedaría de la siguiente manera

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

**class** **MyDocument** **extends** **Document** {

// Cosas que podemos hacer desde el server

**static** **async** getInitialProps(ctx) {

// todos los cambios aplican para todas las paginas

**const** initialProps = **await** Document.getInitialProps(ctx)

**return** { ...initialProps }

}

render() {

**return** (

<Html>

<Head>

{/\* lo modificamos para \*/}

{/\* favicon \*/}

{/\* webfonts personalizadas \*/}

{/\* stylesheet externos \*/}

{/\* script/js externos \*/}

</Head>

{/\* No es muy comun editar adentro del body \*/}

{/\* pero se puede usar para agregar una clase \*/}

<body className="my-body-class">

<Main />

<NextScript />

</body>

</Html>

)

}

}

export default MyDocument

De esta forma modificamos y customizar el document del HTML para hacer cosas personalizable de forma visible que no podemos hacer con Next.js

**RESUMEN:** En esta clase podemos observar que para editar directamente el document debemos crear una pagina en pages, con el nombre de “\_document.js” y usar como template el codígo de la documentación

<https://nextjs.org/docs/advanced-features/custom-document>

**Advertencias sobre extender el Document:**

* Ya que el Document se renderiza en servidor, los eventos como onClick no funcionarán.
* Los componentes de React fuera de <Main /> no serán inicializados por el navegador. No añadir lógica de la aplicación aquí o CSS personalizado (como styled-jsx). Si necesita componentes compartidos en todas sus páginas (como un menú o una barra de herramientas).
* getInitialPropsfunción de Document no se llama durante las transiciones del lado del cliente, ni cuando una página está optimizada estáticamente.