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