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.

# Extendiendo el App

En el archivo \_app podemos inyectarle anábolicos (providers, themes, props, o cualquier cosa que necesitemos) a toda nuestra aplicación.

La forma en la que Next.js renderiza en el navegador los contenidos es en el siguiente orden:

* Document
  + App
    - Nuestra Aplicación

para usar el app de nuestra aplicación lo que haremos sera utilizar el template de la documentación de Next.js para un custom app, el cual debemos crear un archivo llamado “\_app.js” en pages

**import** { AppProps } **from** 'next/app'

**function** **MyApp**({ Component, pageProps }: AppProps) {

// Providers - Context/Providers, Theme, data

// Layout

// props adicionales

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

}

export default MyApp

**RESUMEN:** Para crear contenedores de nuestra aplicaciones, que deseemos aplicarlo a nivel global, es muy util usar el app de Next.js creando el archivo \_app.js

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

# Path alias

Si están en Windows sí es necesario que pongan el slash en baseUrl, pero en iOS y Linux no.  
Además, con JavaScript también lo pueden hacer creando el archivo jsconfig.json y poniendo la configuración de la siguiente manera.

{

"compilerOptions": {

"baseUrl": "./",

"paths": {

"@components/\*": [

"components/\*"

]

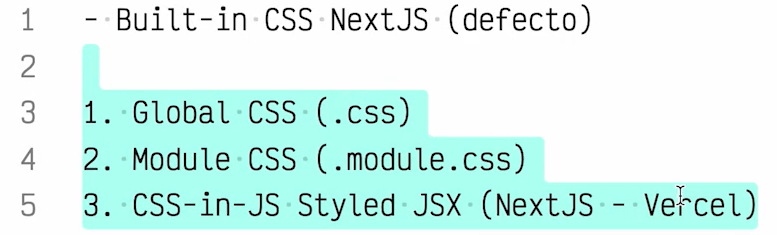
}

}

}



# Explora las soluciones de CSS en NextJS y su flexibilidad



Next.js no nos ofrece una recomendación en particular para los estilos. Pero nos dan tres formas de dar estilos sin cambiar nada en su configuración.

**Built-in CSS Next.js**

**1.- Global CSS (.css)**

Todos nuestros estilos se encontraran en un solo lugar para ello debemos hacer lo siguiente

Crear un archivo CSS en la raíz de nuestro proyecto

.container {

background-color: salmon;

}

/\* agregarlo a la app \*/

/\* utilizar la prop \*/

importar el archivo en \_app.js

...

import '../style.css'

**function** **MyApp**({ Component, pageProps }: AppProps) {

**return** (...)

}

**export** **default** MyApp

usar la clase

**const** Layout: React.FC = ({ children }) => {

**return** (

<div className="container">...</div>

)

}

**export** **default** Layout

**2.- Module CSS (.module.css)**

Usa CSS modules para usarlo debemos crear un archivo CSS el cual debe tener la siguiente nomenclatura

[nombre].module.css

Y para usarlo debemos importarlo como si fuese un archivo JS

...

import styles **from** './layout.module.css'

**const** Layout: React.FC = ({ children }) => {

**return** (

<https://www.youtube.com/watch?v=NzQNZSIEoW8> <div className={styles.container}>...</div>

)

}

**export** **default** Layout

la ventaja es que al momento de aplicarlo genera una clase única con un hash

**3.- CSS-in-JS Styled JSX (Next.js - Vercel)**

Esta es la forma de crear por defect en Next.js CSS-in-JS la cual se usa poniendo la etiqueta <style> dentro de nuestro componente, marcando la propiedad de JSX y en su contenido agregando un template literal

Ejemplo

**const** Layout: React.FC = ({ children }) => {

**return** (

<div>

...

<footer className="container">this is the footer</footer>

<style jsx>{`

.container {

background: salmon;

}

`}</style>

</div>

)

}

Sus ventajas es que el alcance de los estilos solo afectan al componente en si. ademas que genera clases con un hash para no colisionar estilos.

**RESUMEN:** Se tienen tres formas de usar estilos median estilos globales, módulos y CSS-in-JS. Cada uno tiene ventajas y desventajas.

Data Fetching y Pre-rendering

# Introducción a los pre-render modes

Client Side Rendering (**CSR**): Significa que toda la UI, va a ser renderizada por el navegador de nuestro usuario.

Server Side Rendering (**SSR**): Significa que ahora todo el frontend se va a *renderizar* en el server donde esta hosteado el sitio o la web app. Y por lo que entiendo esto significa que si alguien tiene un dispositivo muy viejo o modesto, la app o el sitio se comportará igual de rápido que si tuviera un dispositivo de gama alta; claro suponiendo que ambos tienen exactamente el mismo servicio de internet/datos.

**Static**: Se refiere a los assets o a cierto contenido que no va a cambiar, para esto se guarda en la carpeta public, la cual debe de estar ubicada en el root de nuestro proyecto.

CSR ➡ ❌ Afecta el SEO  
✔ algo positivo no tenemos problemas de TTFB (Time To First Byte)

SSR ➡ ❌ Afecta el redimiendo de nuestro sitio web con el TTFB (Time To First Byte)  
✔ algo positivo es que mejora el SEO

SSG ➡ ❎ Lo mejor de los dos mundos  
✔ No tenemos problemas de TTFB ni tampoco con el SEO

<https://medium.com/@wilbo/server-side-vs-client-side-routing-71d710e9227f>

# #UnderTheHood Server Side Rendering: getServerSideProps

Usamos useEffect para traer la información de la API, debido a esto estamos renderizando desde el cliente. Es decir que desde el servidor no esta información de los elementos.

Para poder extraer los datos desde el servidor lo que debemos hacer es usar la siguiente función:

**export** **const** getServerSideProps = **async** () => {

// la función fetch debe de venir desde una libreria que nos ayude con la tarea

**const** response = **await** fetch('https://platzi-avo-alpha.vercel.app/api/avo')

**const** { data: productList }: TAPIAvoResponse = **await** response.json()

// Devuelve un objecto el cual luego se pasara como prop

// en el componente

**return** {

props: {

productList,

},

}

}

**const** HomePage = ({ productList }: { productList: TProduct[] }) => {...}

🚧 importante las rutas de las peticiones no se deben hacer con una ruta relativa, debemos tener un servidor listo con una ruta absoluta.

**RESUMEN:** La carga desde el cliente se mueve hacia el servidor con la función getServerSideProps y retornar un objeto con una propiedad llamada prop, entonces si la API tarda en responder la página tardara en dar una respuesta.

# #UnderTheHood Static Generation: getStaticProps

Funciona de igual forma que getServerSideProps. A diferencia del server side el cual funciona on demand, cada vez que hay un request del usuario el server debe hacer otro request a la API que deseemos consumir. En cambio, de esta manera este request solo se da una sola vez, cuando lo compilamos

**RESUMEN:** Esta manera de hacer Request nos ayuda de una manera increíble de tal forma que los datos estén directamente listos.

### [¿Cuándo debo usar getStaticProps?](https://nextjs.org/docs/basic-features/data-fetching#when-should-i-use-getstaticprops)

Debe usar getStaticPropssi:

* Los datos necesarios para representar la página están disponibles en el momento de la compilación antes de la solicitud del usuario.
* Los datos provienen de un CMS sin cabeza.
* Los datos se pueden almacenar en caché públicamente (no específicos del usuario).
* La página debe estar previamente renderizada (para SEO) y ser muy rápida: getStaticPropsgenera archivos HTML y JSON, los cuales pueden ser almacenados en caché por un CDN para mejorar el rendimiento.

<https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation>

<https://nextjs.org/blog/next-9-4#improved-built-in-fetch-support>

# #UnderTheHood Static Dynamic Static Generation: getStaticPaths

[Basic Features: Data Fetching | Next.js](https://nextjs.org/docs/basic-features/data-fetching#getstaticpaths-static-generation)

Cuando necesitamos hacer request con getStaticProps de cosas especificas entonces usamos la función que conocemos. Pero a getStaticProps le llega un contexto, el cual contiene nuestra ruta actual dentro del objeto params

**export** **const** getStaticProps: GetStaticProps = **async** ({ params }) => {

**const** id = params?.id **as** string

**const** response = **await** fetch(

`https://platzi-avocados.vercel.app//api/avo/${id}`

)

**const** { data: product }: TAPIAvoResponse = **await** response.json()

**return** {

props: {

product,

},

}

}

Pero de antemano debemos decirle a Next.js cuales serán nuestras paginas que tendremos con la función getStaticPaths

**export** **const** getStaticPaths = **async** () => {

**const** response = **await** fetch('https://platzi-avocados.vercel.app/api/avo')

**const** { data: productList }: TAPIAvoResponse = **await** response.json()

**const** paths = productList.map(({ id }) => ({

params: {

id,

},

}))

**return** {

paths,

// Incremental static generation

// 404 for everything else

fallback: **false**,

}

}

**RESUMEN:** Para generar rutas dinámicas y con extracción del servidor es necesario tener los dos métodos getStaticPaths para devolver un objeto el cual tenga las posibles rutas y getStaticProps con su context el cual nos dará la ruta actual

<https://nextjs.org/docs/basic-features/data-fetching#getstaticpaths-static-generation>