Tutorial de ReactJS y NextJS

Participantes del grupo

- Luis Vidal Rico lvr3@gcloud.ua.es
- Andrés Felipe López Franc aflf1@gcloud.ua.es
- Alejandro Aguado aam196@gcloud.ua.es

¿Qué es React?

React es una librería de JavaScript desarrollada por Facebook, que nace en 2013 de la necesidad de cambiar el tradicional paradigma de programación web donde el desarrollo se dividía en HTML, CSS y JavaScript divididos en diferentes carpetas. Lo que querían conseguir con esta librería era que las aplicaciones estuvieran construidas de pequeños componentes que al juntarse pudieran formar una aplicación completa y que estos también se pudieran migrar y reutilizar en diferentes proyectos de una manera fácil. Así, se puede decir que React es un framework orientado a componentes.

¿Qué es Next?

Next es un pequeño framework construido sobre React que nos sirve para tener configurado con solo una dependencia. De esta manera, NextJs parte de la base que nos proporciona React y le añade comportamientos que de otra manera serían pesados de programar desde cero.

Así, y como ya se comentará más adelante, Next nos ofrece la posibilidad de tener server-side rendering configurado desde cero, viene con un Router implementado de base, nos proporciona una api dentro del propio proyecto, viene con internalización o provee de optimización de imágenes, además de muchas otras funcionalidades.

¿Qué diferencias hay entre React y Next?

- **SSR (Server Side Rendering)**: En React no tienes disponible SSR a no ser que lo configures de forma manual, lo cual es posible, pero requiere de mucho más esfuerzo, además el equipo de desarrollo de React no tiene planteado darle soporte en el futuro próximo. A diferencia de Next, donde el SSR está listo para usarse sin ningún tipo de configuración manual ya que ya viene incluido en el framework.
- **Configuración**: React no deja configurarlo a tu gusto, como por ejemplo el webpack, que no puede ser cambiado. En Next casi todo es configurable, archivos como *babelrc, jest.config, eslintrc*, los puedes configurar sin problema.
- **Router**: React router permite SSR, pero no lo implementa. Necesita crear un script de servidor que, al menos, muestre la aplicación en una cadena y se la sirva al cliente . Es posible que deba hacer otras cosas, como servir archivos estáticos. NextJS tiene por defecto un router.

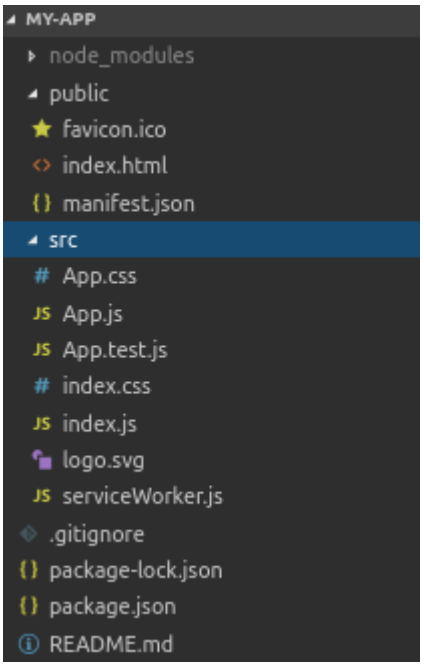
En resumen, Next ofrece más ventajas que React, ya que es mucho más configurable y tiene incluidas características (router, SSR…) que en React habría que configurarlas de forma manual, cosa que es mucho más complicada.

¿Cómo funciona React? ¿Qué es JSX?

Creación de un aplicación con react y ponerla en marcha:

```
npx create-react-app my-app
cd my-app
npm start
```

Una vez ejecutamos este comando, se nos genera la siguiente estructura de archivos, la cual como podemos ver es bastante simple:



En los archivos dentro de la carpeta public encontramos el archivo index.html con la estructura mínima de la aplicación y div identificado como “root” en el cual se insertarán todos nuestros módulos.

A continuación tenemos la carpeta src donde encontramos index.js, el main de nuestra aplicación en React. En este se importa React, el React-dom (ya que trabajamos en entorno web) y los módulos existentes de nuestra aplicación. También aquí es donde se inyecta el contenido en el div “root”.
La forma más sencilla de definir un componente es escribir una función de JavaScript:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Esta función es un componente de React válido porque acepta un solo argumento de objeto props (que proviene de propiedades) con datos y devuelve un elemento de React.
Llamamos a dichos componentes funcionales porque literalmente son funciones JavaScript.
También se puede usar una clase de ES6 para definir un componente:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Las dos formas son válidas. De esta forma, tendremos un componentes completamente reusable dentro de nuestra aplicación, que contendrá su propio estado, el cuál será gestionado por React.

¿Qué es JSX?

JSX es una extensión de sintaxis de JavaScript que nos permite mezclar JS y HTML (XML), de ahí su nombre JavaScript XML.
Esta extensión nos facilita la vida pues nos permite escribir un código más limpio, sin tantas repeticiones, y con muy pocos factores o condiciones a tener en cuenta. Un ejemplo de JSX es el siguiente:

```
<select className="form-control mt-3 mb-4" id="selectEtiqueta" onClick={cambiarEtiqueta} style={{ visibility: "hidden" }}
  {etiquetas && etiquetas.map(etiqueta => (
    <option key={etiqueta.id}>{etiqueta.nombre}</option>
  ))}
</select>
```

Como se puede ver en la imágen, para utilizar JS se utiliza la expresión “{}”. Así, en la segunda línea se comprueba si existe la variable *etiquetas*, y si lo hace, se recorre el array y se devuelve un <option> por cada uno de los valores. Esta sintaxis es parecida a la de otros frameworks como Vue o Blade, con la diferencia de que en este se llama a JS directamente, en vez de llamar a directivas ofrecidas por el framework.

¿Cómo funciona?

Las expresiones JSX se convierten en llamados regulares de funciones de JavaScript que finalmente evalúan a un Objeto.

```
<h1>Hello world!</h1>
```

evaluaría a algo como:

```
React.createElement("h1", null, "Hello world!");
```

React.createElement es el método que se invoca cada vez que vamos a crear un elemento. Este método recibe 3 argumentos. El primero es el tipo o una cadena de texto (String) en este caso “h1”, el segundo son los atributos o propiedades del elemento en este

caso null, y por último, desde el tercer argumento hacia adelante, los hijos del elemento siendo otros elemento de React ***React.createElement();*** o una cadena de texto que en este caso sería “Hello world!”.

Veamos un ejemplo más claro:

```
var Icon = (
  <div className="icon-container">
    <img src="icon-react.png" className="icon-image" />
  </div>
);

ReactDOM.render(Icon, document.getElementById("app"));
```

Como puedes ver es mucho más práctico y legible esta sintaxis. Usando JSX conseguimos crear un elemento *img* usando código de HTML, pero sigue siendo JavaScript.

¿Qué son los ‘hooks’ de React? ¿Para qué sirven?

Los *Hooks* son una herramienta que permite *enganchar* los componentes funcionales a las características que ofrece React. Esto nos permite conocer el estado de los componentes creados y modificarlos a través de las herramientas que React nos ofrece. De esta forma, a través de las funciones que nos ofrece React podemos gestionar de forma sencilla cosas como el estado o tener referencias a elementos del DOM.

Antes de que esta herramienta existiera obligaba a los programadores a desarrollar los componentes de la siguiente manera:

```
import React, { Component } from 'react'

class Contador extends Component {
  state = { count: 0 } // inicializamos el state a 0

  render () {
    const { count } = this.state // extraemos el count del state

    return (
      <div>
        <p>Has hecho click {count} veces</p>
        { /* Actualizamos el state usando el método setState */ }
        <button onClick={() => this.setState({ count: count + 1 })}>
          Haz click!
        </button>
      </div>
    )
  }
}
```

Ahora con esta funcionalidad se puede conseguir lo mismo utilizando una *función* e importando el hook *useState* , como podemos ver en la siguiente imagen:

```jsx
// importamos useState, el hook para crear un state en nuestro componente
import React, { useState } from 'react'

function Contador() {
  // useState recibe un parámetro: el valor inicial del estado (que será 0)
  // y devuelve un array de dos posiciones:
  //  la primera (count), tiene el valor del estado
  //  la segunda (setCount), el método para actualizar el estado
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Has hecho click {count} veces</p>
      { /* actualizamos el state al hacer click con setCount */ }
      <button onClick={() => setCount(count + 1)}>
        Haz click!
      </button>
    </div>
  )
}
```

El hook useState

El hook useState permite a React mantener el estado entre re-renderizados. Este método devuelve un array de dos elementos, el primero es el estado actual y el segundo es una función para actualizarlo. Para usar este hook debes importarlo desde ‘react’.

El único argumento que recibe useState es el estado inicial, que solo se ejecutará en el primer render. Además, puedes utilizar la función para cambiar el valor de la variable desde cualquier parte del código. Podemos decir que React nos proporciona un *setter* para nuestros estados, ya que estos son inmutables. De esta forma, cada vez que actualicemos un dato React podrá responder de forma inteligente a este, optimizando así el funcionamiento.

```jsx
import React, { useState } from 'react';

function Example() {
  // Declara una nueva variable de estado, que llamaremos "count".
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

El hook useEffect

El hook useEffect proporciona la capacidad de realizar efectos secundarios desde un componente funcional. Cuando llamas a useEffect le estás diciendo que ejecute tu función de efecto después de vaciar los campos en el DOM. Es decir, cada vez que se acabe el renderizado de un componentes.

Este *hook* se declara en el mismo componente para que tenga acceso a las props y al estado. React ejecuta estos efectos después de cada renderizado, incluyendo el primer renderizado, aunque a este *hook* también se le puede pasar un array de dependencias como segundo parámetro. De esta forma, la función se ejecutará cada vez que alguna de estas dependencias cambie de valor.

Con esta array de dependencias hay que llevar cuidado, ya que si modificamos alguno de los valores de los que depende useEffect dentro del mismo método, este se ejecutará como un bucle sin fin, provocando un gasto de memoria imposible de abordar.
Un ejemplo del hook useEffect es el siguiente:

```jsx
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar a componentDidMount y componentDidUpdate:
  useEffect(() => {
    // Actualiza el título del documento usando la Browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```jsx
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

El hook useRef

Esencialmente useRef es una *caja* en la que puedes guardar una variable mutable en su propiedad *.current*. Esto es, se puede usar como referencia para acceder a ciertos campos del DOM. Básicamente useRef es un puntero que nos permite apuntar a ciertos elementos del DOM.

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` apunta al elemento de entrada de texto montado
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

Otros hooks, cómo useContext o useMemo

Existen otros hooks cuyo uso menos común que pueden ser útiles, como por ejemplo useContext que permite suscribirse al contexto de React sin introducir el anidamiento, o también el hook useMemo que sirve como caché para guardar el estado de ciertas variables cuyo estado es el producto de cálculos complejos los cuales consumen mucha memoria de procesamiento.

Vue vs React, ¿en qué se diferencian?

Hay 4 grandes diferencias que son:

- React usa JSX que es una extensión de JavaScript mezclado con código HTML directamente para crear sus componentes creando un DOM virtual, lo cual hace que los cambios en el DOM sean mucho más ligeros que en si fueran en el DOM real.

  Por el contrario, Vue utiliza HTML puro para desarrollar sus templetes lo cual hace que se pueda integrar en diferentes proyectos más fácilmente.

- La gestión del estado y propiedades del objeto: en React necesitas una función setState para poder modificar el estado del objeto mientras que vue no necesita de un objeto de estado puesto que los datos se administran desde la propiedad data del objeto vue.
- React Native: React tiene una librería que permite que se puedan desarrollar aplicaciones de forma nativa para android y IOS directamente con HTML CSS y JavaScript lo cual le da un plus a este framework aunque vue también tiene su librería para esto, Weex aunque no es tan potente ni tan conocida como React Native.
- Vue es puramente reactivo, mientras React no lo es (como vimos en clase) aunque según este desarrollador esto no debería importarnos.
- Como diferencias secundarias podríamos destacar:
    - Vue tiene una menor curva de aprendizaje
    - Vue no tiene una empresa grande que lo respalde, a diferencia de React que tiene a Facebook respaldando (en este caso lo respalda la propia comunidad).
    - Vue tiene menos oferta que React, ya que las empresas no han apostado tanto por adoptar su uso, precisamente por que no hay una empresa grande detrás respaldando.
    - React tiene, a parte de Facebook, una gran comunidad que lo respalda y crea soporte y librerías para este Framework
    - React tiene una gran oferta de trabajo por lo que es el mejor framework para empezar y conseguir trabajo fácil y rápidamente

A continuación se muestran dos imágenes, a la izquierda de React y a la derecha de Vue, de dos componentes, cada uno implementado con un Framework:

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

```
<template>
  <div id="app">
    <todo-list />
  </div>
</template>

<script>
import TodoList from './components/TodoList.vue'

export default {
  name: 'app',
  components: {
    TodoList
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

En conclusión ambos frameworks son bastante similares, tienen en general la misma función ambos son frameworks de front-end, dependiendo de cual sea la situación del usuario debe decantarse por uno u otro, vue por su parte tiene una curva de aprendizaje más corta que React, en contraposición, React tiene más respaldo y es más robusto, también puede ser incluso más rápido que Vue debido a su DOM virtual pero añade la complejidad del JSX

Frameworks y Librerías disponibles en React

Frameworks de React:

- **React Bootstrap**: Te ofrece multitud de componentes con los estilos de Bootstrap pero basados en React listos para utilizar y conseguir una interfaz atractiva instantáneamente.
- **React Material UI**: Material es el estilo de interfaz móvil más empleado. Es el estilo de diseño que ha creado y promueve Google, y el que se utiliza por defecto en todas sus aplicaciones Web y en Android.
- **Blueprint**: Blueprint es una colección de componentes de la interfaz de usuario de React que cubre la mayoría de los elementos, patrones e interacciones comunes de la interfaz en la web. El uso de Blueprint asegura que terminará con una interfaz de usuario elegante y fácil de usar, lo que te permitirá concentrarte en la construcción de tu producto, no en las piezas atómicas que lo componen.
- **Ant Design**: Ant Design ofrece un lenguaje de diseño de interfaz de usuario de clase empresarial para aplicaciones web con un conjunto de componentes React de alta calidad listos para usar. Está escrito en TypeScript con tipos definidos completos.
- **React MD:** Los objetivos de React MD son poder crear un sitio web con estilo de diseño de materiales totalmente accesible utilizando React Components y Sass. Con la separación de estilos en Sass en lugar de estilos en línea, debería ser fácil crear componentes personalizados con los estilos existentes.
- **Next.js**: Del que ya hemos hablado anteriormente pero había que mencionar como framework de React.
- **Gatsby.js**: Gatsby es un framework gratuito y open source basado en React que permite desarrollar de forma rápida sitios web y aplicaciones web. Gatsby quiere ser el framework universal de JavaScript que tiene la visión de que todo el “sitio web es un aplicación web y toda aplicación web es un sitio web”.

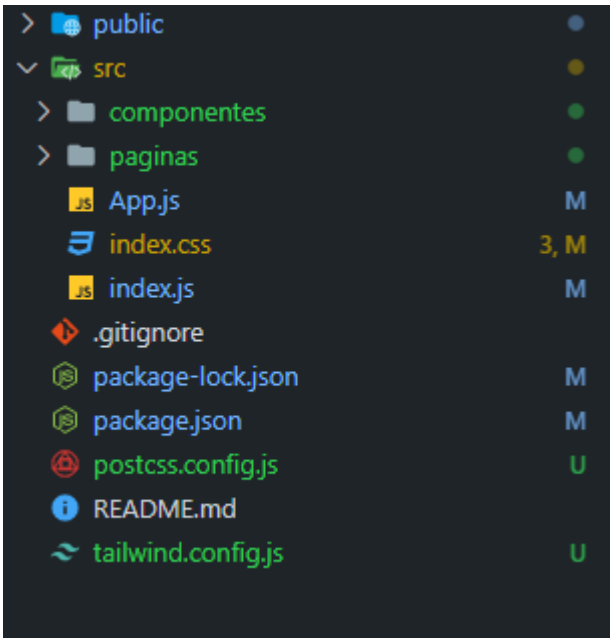
Aplicación ToDoList en React

Se ha desarrollado la misma aplicación ToDoList en ambos frameworks, React y Next, con el objetivo de mostrar las principales diferencias entre ambos. Así, para crear la aplicación de React, hemos ejecutado el siguiente comando:

```
C:\Users\Usuario>npx create-react-app todolistreact
```

Este comando creará una aplicación por defecto, como se podía hacer con vue-cli, que contendrá varios archivos sobre los que podremos empezar a programar. Así, de todos los archivos que nos proporciona React dentro de la carpeta *src* nos quedamos únicamente con el archivo App.js, que contiene el jsx de partida de nuestra aplicación, el archivo index.css, en el que estará el css que creemos y el archivo index.js, que es donde se inicializa React a partir del archivo App.js nombrado anteriormente.

De esta forma, la estructura de directorios es la siguiente:



Donde:

- public: Contiene el archivo index.html (donde React inyectará todo nuestro código según vaya necesitando) y algunos logos con los que se inicia la app.
- src: Contiene todo el código de nuestra aplicación. En nuestro caso hemos creado las carpetas *componentes*, que contendrá los componentes de la app y *paginas*, que contendrá las páginas.

Así, el archivo App.js, que es el punto de entrada, contendrá el siguiente código:

```
import {Routes, Route} from 'react-router-dom';

import Home from './paginas/Home'
import About from './paginas/About'

function App() {

  return (
      <Routes className="text-center">
        <Route path="about" element={<About/>}/>
        <Route path="/" element={<Home/>} />
      </Routes>
  );
}

export default App;
```

En React no viene por defecto un Router, hay que añadirlo (como en Vue). En este caso, envolvemos con el Router (el componente *Routes*) el resto de Componentes para que este pueda ser usado desde cualquier parte de nuestra aplicación (como si fuera un contexto). Como se ve en la imágen, en la ruta por defecto, "/", cargamos la parte principal de nuestra aplicación, la página Home.

Como cada página de React, esta consta de algunos *Hooks* que implementan el funcionamiento que debe tener. Estos son:

```
const [nextId, setNextId] = useState(1)
const [todos, setTodos] = useState([])
const [newToDoText, setNewToDoText] = useState("")

useEffect(() => {
  setTodos([{text: "Mi primer ToDo!", id: 0}])
}, [])

function añadirToDo(){
  if(newToDoText === "")
    return

  setTodos((current) => current.concat({text: newToDoText, id: nextId}))
  setNextId(nextId + 1)

  setNewToDoText("")
}

function deleteChild(id){
  setTodos(current => current = current.filter(child => child.id !== id))
}
```

A través del *hook* useState() creamos 3 variables, *nextId, todos y newToDoText*. La primera, como se indica en el paréntesis, será un entero, la segunda será un array y la tercera una cadena de texto. Como se ha nombrado en el apartado de este hook, también se proporcionan las funciones que permiten asignar nuevos valores a estas. Así, el funcionamiento de cada una de ellas es el siguiente:

- **nextId**: Guarda cual debe ser el id del próximo todo. Se actualiza cada vez que se crea uno nuevo.
- **todos**: Array de objetos que contendrá todos los todos de la aplicación. Cada todo será un objeto con un texto y un id.
- **newToDoText**: Cadena que se usa para guardar la información que se escribe en el input de crear nuevo todo. Cada vez que el usuario escriba algo, esta variable se actualizará a ese nuevo valor.

A continuación en el código se usa el *hook* useEffect(), el cual se ejecutará cada vez que se re-renderice la aplicación, ya que no se indica ninguna variable en la array de dependencias. En este, se inicializa el array de todos con uno por defecto (es un poco tontería, pero quería mostrar el funcionamiento de este hook. En Next este tendrá más sentido).

Las siguientes dos funciones se encargan de añadir y eliminar todos al array de todos nombrado anteriormente. Serán llamadas desde el código jsx que se muestra a continuación:

```
return (
  <div className="flex flex-col w-full items-center justify-center mt-8 space-y-2">
    <h1 className="text-white text-4xl">Listado de ToDos</h1>
    <div className="flex space-x-3 justify-center">
      <input type="text" value={newToDoText} onChange={(e) => {setNewToDoText(e.target.value)}} className="b
      <button className="px-2 py-1 bg-emerald-600 rounded-full text-white text-sm" onClick={añadirToDo}>Añ
    </div>
    <div className="flex flex-col space-y-2 w-1/2 border border-emerald-600 p-4 rounded-md">
      {todos.map((todo) => (
        <ToDoCard text={todo.text} key={todo.id} onClick={deleteChild} id={todo.id}/>
      ))}
    </div>
    <Link to="/about" className="px-4 py-2 bg-teal-700 rounded-full text-white text-md">About</Link>
  </div>
);
}
```

Como se puede apreciar, a través de jsx se mezcla html convencional con JS. Así, por ejemplo, cada vez que haya un cambio en el input se llamará a la función setNewToDoText(), que actualizará el valor de este. De esta forma le damos a React el control sobre el input.

El siguiente aspecto interesante es cuando queremos mostrar un bucle. En vue, llamábamos a la directiva *v:for*, pero en este caso, usando {} podemos usar JS y llamar al método *map* del array de todos para mostrar un componente *ToDoCard* por cada elemento del array. Al igual que en Vue, tendremos que indicar una *key* para que React pueda tratar de forma eficiente estos componentes.

Nombrar también el uso del componentes Link. En este caso, solamente con importarlo ya lo podemos usar para navegar a través de la página, en este caso al link de “/about”.

De momento solamente hemos visto el uso de páginas en React. Ahora mostraremos el uso de componentes. El componentes ToDoList, nombrado anteriormente, tiene la siguiente definición:

```jsx
import { useState } from "react";

function ToDoCard(props){
    const [tachado, setTachado] = useState(false)

    function toggleTachado(){
        setTachado(!tachado)
    }

    return (
        <div className="flex flex-col justify-center items-center p-2 border border-gray-500 rounded-md space-y-2 shadow-xl">
            <p className={`text-white cursor-pointer ${tachado ? "line-through" : ""}`} onClick={toggleTachado}>{props.text}</p>
            <button className="px-2 py-1 bg-red-800 rounded-full text-white text-xs" onClick={props.onClick.bind(null, props.id)}>Eliminar</button>
        </div>
    );
}

export default ToDoCard;
```

La función ToDoCard recibe un parámetro *props*. En este se encuentran todos los atributos que el padre le pasa al hijo. En este caso, recibe el texto, una función con la que enviar el evento de eliminar un todo, y un id.

Como se puede observar, este componente consta de su propio estado, y se usan en el los *hooks* de React. En este caso el estado se usa para guardar cuándo un todo está tachado o no.

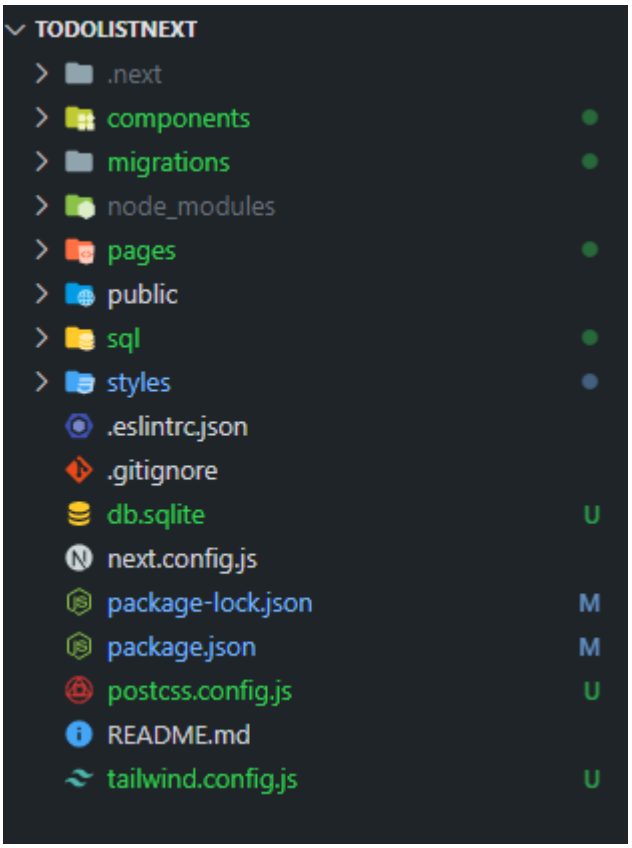
Una vez instalada y inicializada la aplicación, se ve de la siguiente manera:

Aplicación ToDoList en Next

Se ha realizado exactamente la misma aplicación que React, pero usando las funcionalidades que este framework trae por defecto. Así, para iniciar la aplicación se ha usado el siguiente comando:

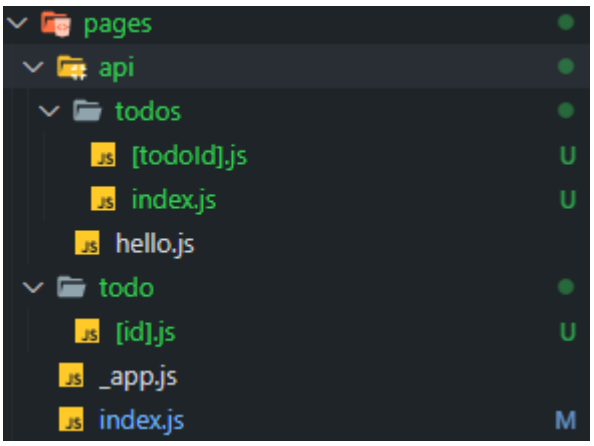
```
C:\Users\Usuario>npx create-next-app todolistnext
```

Una vez creada la aplicación, esta se crea, al igual que con React, con archivos por defecto que sirven como punto de partida para iniciar la aplicación. Eliminamos los que no queremos, y la estructura de directorios resultante es la siguiente:



Como se puede observar, hay varios cambios. En este caso, no existe la carpeta *src*, que es sustituida por *pages*. Las carpetas *componentes*, *migrations* y *sql* las creamos nosotros para estructurar mejor nuestro código. Además, en este framework la carpeta *public* no viene con el archivo index.html.

Un cambio importante de Next respecto a React es la forma de generar las URLs. Al contrario que en React, Next viene con un Router por defecto, y su uso es de lo más curioso. Dentro de la carpeta pages se crearán nuevas carpetas, cuyo nombre coincidirá con la URL con la que queramos acceder a la aplicación. Así, en nuestro caso, dentro de la carpeta *pages* tenemos el siguiente contenido:



De la carpeta api hablaremos un poc más adelante. Como hemos dicho en el párrafo anterior, el nombre de cada carpeta es una ruta en nuestra aplicación. Así, cada vez que accedamos a la url “/todo/:id”, estaremos accediendo al archivo [id].js de la carpeta *todo*. Es decir, si queremos ver en detalle un todo con el id 3, accederemos a “/todo/3”. Ahora imaginemos que en nuestra app tenemos usuarios, y los queremos listar. Tendríamos que crear la carpeta “users”, y dentro meter un archivo “index.js”. De esta forma, cada vez que accedamos a la URL “/users”, estaremos accediendo al archivo index.js de esta carpeta.

Así, logramos tener una aplicación escalable, que desde el primer momento nos obliga a estructurar nuestras páginas utilizando los principios REST.

Otra funcionalidad que implementa NEXT es una pequeña API, a la cuál se accede desde la carpeta *api* de *pages*. El funcionamiento de esta es igual que el de las URLs. Es decir, en nuestro proyecto disponemos del endpoint “/api/todos”, que devuelve todos los Todos de la app, y el endpoint “/api/todos/:id”, que puede ser llamado por GET o por DELETE, según se necesite. El código de *[todoId].js* es el siguiente:

```
import getDbConnection from "../../../sql/getDbConnection"

export default async function handler(req, res) {
    const { todoId } = req.query
    const db = await getDbConnection()
    const todo = await db.get(`SELECT * from todo where id = ${todoId}`)

    if(!todo)
      res.status(404).send({error: `No se ha encontrado el todo con id ${todoId}`})

    if(req.method === "GET")
      res.status(200).send(todo)
    else if(req.method === "DELETE"){
      try{
        await db.exec(`
          DELETE FROM todo
          WHERE id = ${todoId}
        `)
        res.status(200).send({ok: "ok"})
      }catch(e){
        res.status(500).send({error: `Error al eliminar el todo con id ${todoId}`})
      }
    }

}
```

La forma de disponer los archivos en Next puede ser confusa, pero una vez te acostumbras a ella tiene más ventajas que inconvenientes. El uso de esta funcionalidad es mucho más extensa que la comentada en este documento. Aquí se puede leer más sobre ella.

Volviendo a Next, el punto de partida de nuestra aplicación está en el *index.js* que se encuentra en la raíz de *pages*. El contenido de este prácticamente igual que de la página de Home de React. Usa los mismos ganchos, las mismas funciones y el mismo jsx. La única diferencia entre este y el anterior es que en este se realizan llamadas al API implementado para crear, borrar y obtener los todos de la aplicación. Así, la lógica de la aplicación es la siguiente:

```
const [todos, setTodos] = useState([])
const [newToDoText, setNewToDoText] = useState("")

useEffect(async () => {
  const todos = await axios.get("http://localhost:3000/api/todos")
  setTodos(todos.data)
}, [])

async function añadirToDo(){
  if(newToDoText === "")
    return

  const newTodo = await axios.post("http://localhost:3000/api/todos", {
    text: newToDoText
  })

  setTodos((current) => current.concat(newTodo.data))

  setNewToDoText("")
}

async function deleteChild(id){
  await axios.delete("http://localhost:3000/api/todos/" + id)

  setTodos(current => current = current.filter(child => child.id !== id))
}
```

Como se puede apreciar, en este caso el uso de useEffect si que tiene sentido. En este caso, se llama al api y se inicializa la variable *todos* con el resultado de esta llamada. De esta forma, cada vez que reiniciemos la app no se borrarán los todos, como sí pasaba en React.

Las funciones *añadirToDo* y *deleteChild* son básicamente lo mismo que en React, con la diferencia de que llaman al API para actualizar la base de datos.

```
return (
  <div className="flex flex-col w-full items-center justify-center mt-8 space-y-2">
    <h1 className="text-white text-4xl">Listado de ToDos</h1>
    <div className="flex space-x-3 justify-center">
      <input type="text" value={newToDoText} onChange={(e) => {setNewToDoText(e.target.value)}} className="bg-gray-600 border
      <button className="px-2 py-1 bg-emerald-600 rounded-full text-white text-sm" onClick={añadirToDo}>Añadir</button>
    </div>
    <div className="flex flex-col space-y-2 w-1/2 border border-emerald-600 p-4 rounded-md">
      {todos.map((todo) => (
        <ToDoCard text={todo.text} key={todo.id} onClick={deleteChild} id={todo.id} showDltButton={true}/>
      ))}
    </div>
  </div>
);
```

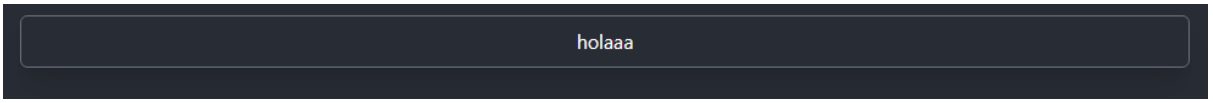
Como se puede ver, el código jsx es esencialmente el mismo.

```jsx
import Link from "next/link";
import { useState } from "react";

function ToDoCard(props){
    const [tachado, setTachado] = useState(false)

    function toggleTachado(){
        setTachado(!tachado)
    }

    return (
        <div className="flex flex-col justify-center items-center p-2 border border-gray-500 rounded-md space-y-2 shadow-xl">
            <p className={`text-white cursor-pointer z-10 w-full text-center ${tachado ? "line-through" : ""}`} onClick={toggleTachado}>{props.text}</p>
            { props.showDltButton ?
                (<>
                    <button className="px-2 py-1 bg-red-800 rounded-full text-white text-xs z-10" onClick={props.onClick.bind(null, props.id)}>Eliminar</button>
                    <Link href={`/todo/${props.id}`}>
                        <button className="px-2 py-1 bg-emerald-700 rounded-full text-white text-xs z-10">Ver en detalle</button>
                    </Link>
                    </>
                )

                :
                ""

            }

        </div>
    );
}

export default ToDoCard;
```

En la imagen anterior se muestra el componente ToDoCard. Es esencialmente el mismo que en la aplicación anterior, aunque hay algo de lógica en el código jsx que cambia.

Así, el resultado de la misma es el siguiente:

Para mostrar como funciona el Routing en Next, se ha creado una página para ver en detalle cada todo:



Cuya URL, como hemos nombrado anteriormente, es ‘/todo/12’.

Código fuente de las aplicaciones

Aquí está el repositorio Github con las aplicaciones creadas.

Conclusiones

Como se puede observar, Next comparte el mismo núcleo que React, pero añade varias funcionalidades extra que facilitan la vida a los desarrolladores. En este tutorial solamente se ha hecho uso del Routing y del api que viene incorporada por defecto, pero hay muchas más ventajas, como Server-side Rendering, Generación Estática o Middlewares.

Además, en este vídeo se explican 5 razones por las que se debería usar Next sobre React, y en este otro, se explica qué es Next, cuáles son sus ventajas y se realiza un breve tutorial.

Referencias

- ¿Que es React?
- ¿ Que es Next?
- Diferencias entre NextJs y npx
- Diferencias entre el Router de React y el de Next
- Estructura de un proyecto React
- Componentes en React
- ¿Qué es JSX?
- ¿Que son los hooks?
- Tutorial React básico
- useState
- useEffect
- useRef
- useContext
- useMemo
- Diferencias entre Vue y React
- Hacker Nómada diferencias entre frameworks de JavaScript
- Frameworks en React
- Gatsby