

UT5 - Integración de
contenido interactivo React

UT6 - Diseño de webs
accesibles e implementación de
usabilidad en la web

Indice

1.- React - Integración de contenido interactivo.....	5
1.1.- React.....	5
1.2.- Lo necesario para utilizar React.....	5
1.3.- Preparación del entorno.....	6
1.3.1.- ¿Qué es Node.js?.....	6
1.3.2.- ¿Qué es npm?.....	6
1.3.3.- Instalación de Node.js.....	6
1.4.- Primeros pasos en React.....	7
1.5.- Formas de escribir código en React.....	10
1.6.- Variables en React.....	10
1.7.- Componentes en React.....	10
1.8.- Componentes de clase.....	12
1.9.- Componentes simples.....	14
1.10.- Props.....	15
1.11.- State – Permite modificar los datos de un componente.....	19
1.12.- Tratamiento de información en ficheros JSON.....	25
1.13.- Envío de formularios de datos.....	26
1.14.- Obtener datos de una API.....	29
1.15.- Manejo de rutas en React.....	33
1.16.- Construir y desplegar una aplicación en React.....	35
1.17.- Ejercicios.....	36
2.- UT6 - Diseño de webs accesibles e implementación de usabilidad en la web.....	38
2.1.- Selectores de atributos.....	38
2.2.- Box-sizing.....	41
2.3.- Formularios– Estilos en formularios.....	42
2.4.- Etiquetado campo radio.....	46
2.5.- Transition.....	46
2.6.- Distribución mediante grid.....	48
2.7.- Medida FR.....	50

1.- React - Integración de contenido interactivo.

1.1.- React

React es una biblioteca de JavaScript para crear interfaces de usuario. Este proyecto es impulsado por Facebook Open Source y es la base de multitud de creaciones muy importantes hoy en día.

Está diseñado para crear SPA (Single Page Application). En este tipo de aplicaciones web, todo lo que se muestra y se procesa está dentro de la misma página, así que al pasar de una opción a otra no hace falta recargar el navegador y lo normal es que trate de un único archivo desde el que se reproduce absolutamente todo.

Utiliza programación orientada a componentes. Estos componentes se representan como clases que heredan de la clase **Component**.

- Otros detalles:

No se suele considerar un framework.

Se utiliza para crear interfaces en el front-end.

Es la capa vista en un Modelo Vista Controlador

- Conceptos importantes:
 - Modelo Vista Controlador
 - Hay que recordar cómo funciona un Modelo Vista Controlador:
 1. El cliente (usuario) hace una petición al servidor.
 2. El controlador procesa la petición que hace el usuario y solicita los datos al modelo.
 3. El modelo se comunica con la base de datos y retorna la información.
 4. El modelo retorna los datos al controlador.
 5. El controlador retorna la vista al cliente con los datos solicitados.
 - Front-end y Back-end
 - El front-end abarca todo aquello que se refiere a la interfaz de usuario y la usabilidad de la aplicación.
 - El back-end procesa las interacciones del usuario (que le pasa el front-end) y realiza todos los procesos con los datos recibidos.

La documentación oficial está disponible en el siguiente [enlace](#).

1.2.- Lo necesario para utilizar React

Necesitamos tener conocimientos sobre las siguientes tecnologías:

- HTML y CSS
- JavaScript
- DOM: Document Object Model – Define la estructura básica de los elementos en documentos HTML y permite realizar modificaciones en tiempo real mediante la utilización de lenguajes como JavaScript).

- ES6, también llamada ECMAScript 2015 (especificación lanzada en 2015 que define estándares para JavaScript).
- Node.js y npm instalados de forma global.

1.3.- Preparación del entorno

Instalación de Node.js y npm de forma global (disponible para todos los proyectos, no para únicamente de forma local para un proyecto)

1.3.1.- ¿Qué es Node.js?

Node.js es un entorno en tiempo de ejecución que utiliza código abierto de JavaScript y que ha sido creado para generar aplicaciones web de forma muy optimizada.

En este caso cambia el concepto que tenemos de JavaScript, que normalmente lo hemos usado únicamente en la parte de cliente.

Mediante Node.js se proporciona un entorno de ejecución del lado del servidor que compila y ejecuta aplicaciones a velocidades muy altas. Utiliza un modelo asíncrono y dirigido por eventos.

1.3.2.- ¿Qué es npm?

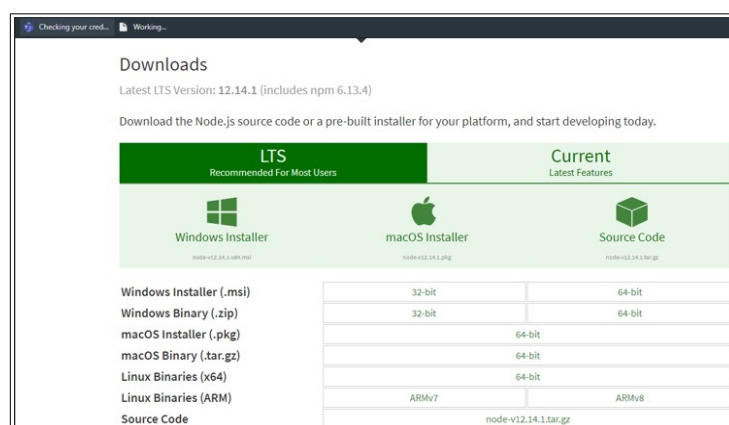
Es el sistema de gestión de paquetes que utiliza por defecto Node.js. Gracias a él podremos tener cualquier librería disponible con sólo una línea de código. Npm nos ayudará a administrar módulos, distribuir paquetes y agregar dependencias de una manera sencilla.

1.3.3.- Instalación de Node.js

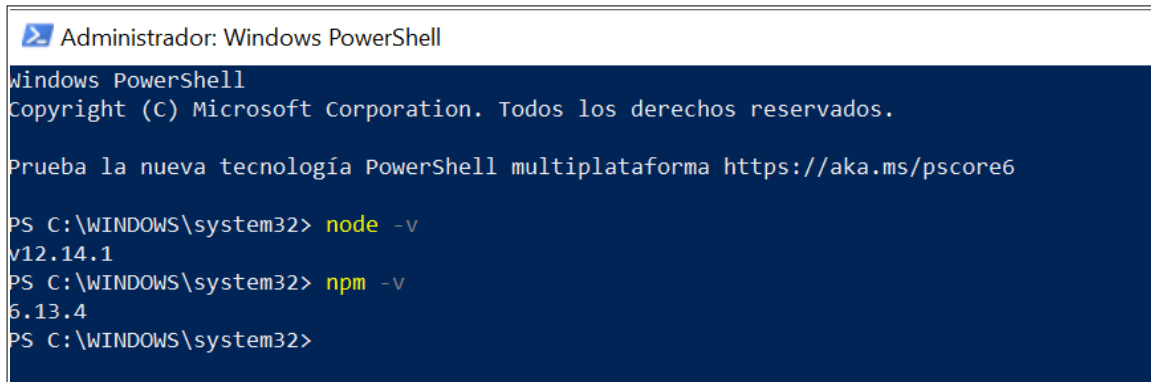
En este caso vamos a hacer la instalación en Windows, pero también podríamos implantarlo en Linux o Mac OS X porque se trata de software multiplataforma.

Debemos realizar la descarga desde la [web oficial](#). Nos interesa la versión LTS, que es mucho más estable.

Los pasos de instalación son muy sencillos:



Dejaremos las opciones de instalación por defecto, mediante PowerShell, podemos comprobar que **Node.js** y **npm** está correctamente instalados:



```
Administrador: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\WINDOWS\system32> node -v
v12.14.1
PS C:\WINDOWS\system32> npm -v
6.13.4
PS C:\WINDOWS\system32>
```

1.4.- Primeros pasos en React

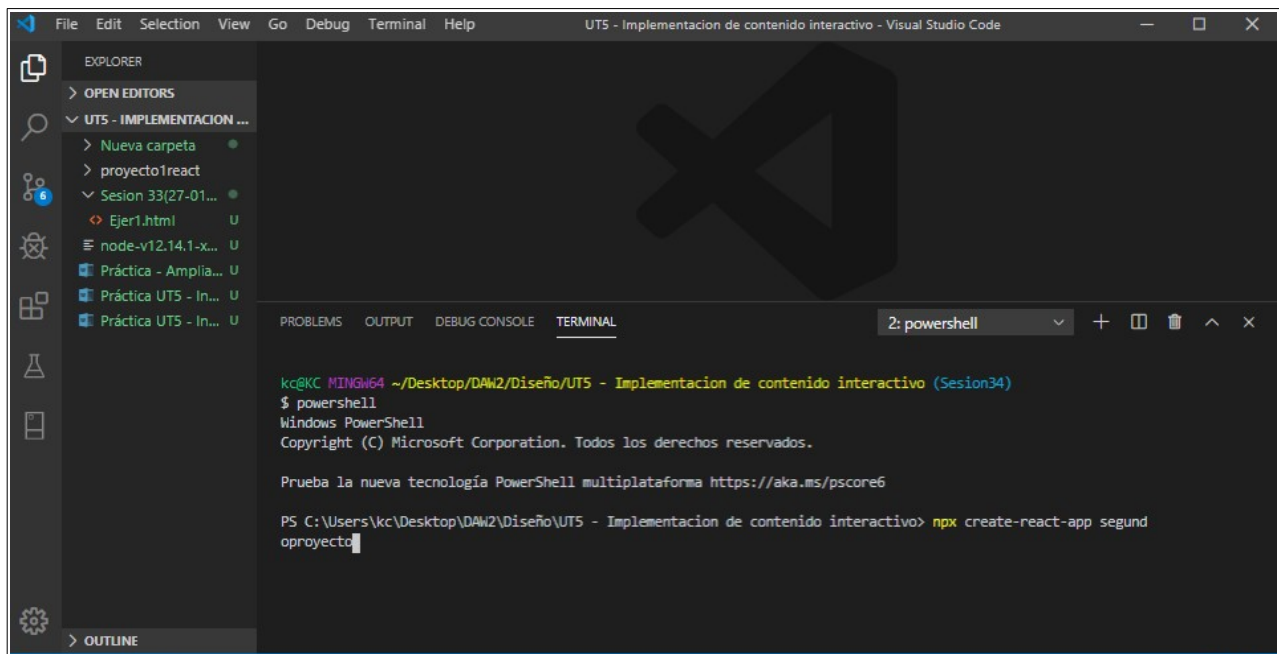
Creación de una aplicación React

Es posible realizar los preparativos de React mediante la utilización de un fichero estático html llamado “index.html” y la carga de varias librerías escritas en JavaScript, pero vamos a hacerlo de otra forma más directa y sencilla.

Para evitar lo antes nombrado, que puede ser bastante engorroso e ineficiente, Facebook ha creado un entorno preconfigurado llamado “**Create React App**”. Tenemos mucha documentación oficial [aquí](#). En nuestro caso lo vamos a hacer lo siguiente.

Para preparar “create-react-app” tenemos que ejecutar las siguientes órdenes en la terminal (cuidado, **React no permite mayúsculas en el nombre de los proyectos**). Vamos a proceder a la creación de nuestro primer proyecto mediante mediante “**create-react-app**”

```
npx create-react-app primerproyectorreact
```



```
File Edit Selection View Go Debug Terminal Help
UT5 - Implementacion de contenido interactivo - Visual Studio Code

EXPLORER
> OPEN EDITORS
> UT5 - IMPLEMENTACION ...
  > Nueva carpeta
  > proyecto1react
  > Sesión 33(27-01...
    < Ejer1.html
    < node-v12.14.1-x...
    < Práctica - Amplia...
    < Práctica UT5 - In...
    < Práctica UT5 - In...

TERMINAL
2: powershell
kc@KC MINGW64 ~/Desktop/DAW2/Diseño/UT5 - Implementacion de contenido interactivo (Sesion34)
$ powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\kc\Desktop\DAW2\Diseño\UT5 - Implementacion de contenido interactivo> npx create-react-app segund
oproyecto
```

Después de la preparación de “create-react-app”, estamos listos para cambiar al directorio del proyecto e iniciarlo mediante la orden “npm start”:

```
PS C:\Users\tiwx\OneDrive - Educacyl\IES María Moliner 2019-2020\Materiales alumnos\DIW\NuevoTemarioDesarrollado\ProyectoReact> cd .\primerproyectoreact\
PS C:\Users\tiwx\OneDrive - Educacyl\IES María Moliner 2019-2020\Materiales alumnos\DIW\NuevoTemarioDesarrollado\ProyectoReact\primerproyectoreact> npm start
> primerproyectoreact@0.1.0 start C:\Users\tiwx\OneDrive - Educacyl\IES María Moliner 2019-2020\Materiales alumnos\DIW\NuevoTemarioDesarrollado\ProyectoReact\primerproyectoreact
> react-scripts start
```

Una vez que hemos lanzado esta orden, se abrirá una pestaña en el navegador con el proyecto que acabamos de crear. Una peculiaridad respecto al desarrollo web al que estamos acostumbrados, es que no es necesario recargar la página ante los cambios, todo esto debido a que el servidor está en modo escucha ante posibles modificaciones y compilará el código después de una orden de guardado.

En la siguiente imagen podemos ver la página principal del proyecto (podemos modificar el código html para visualizar los cambios):

Si miramos la estructura de directorios del proyecto, vemos que es más compleja de lo normal, pero por el momento sólo nos vamos a fijar en el directorio src que contiene todo nuestro código React.

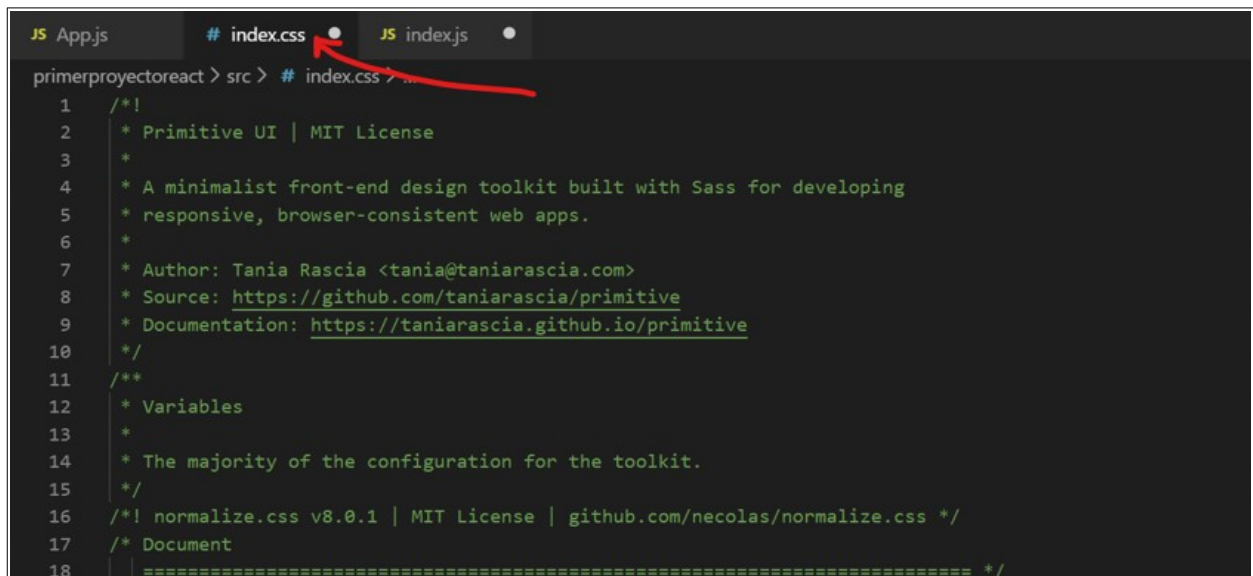
Vamos a borrar todos los ficheros del directorio /src, salvo:

- **index.js**
- **index.css**

V2 > Diseño > UT5 - Implementacion de contenido interactivo > proyecto1react > src					Buscar en ...
Nombre	Fecha de modificación	Tipo	Tamaño		
App.css	26/10/1985 10:15	Archivo CSS	1 KB		
App.js	26/10/1985 10:15	Archivo JavaScript	1 KB		
App.test.js	26/10/1985 10:15	Archivo JavaScript	1 KB		
index.css	26/10/1985 10:15	Archivo CSS	1 KB		
index.js	26/10/1985 10:15	Archivo JavaScript	1 KB		
logo.svg	26/10/1985 10:15	Documento SVG	3 KB		
serviceWorker.js	26/10/1985 10:15	Archivo JavaScript	5 KB		
setupTests.js	26/10/1985 10:15	Archivo JavaScript	1 KB		

En index.css vamos a copiar el código del framework de código abierto “Primitive CSS” para utilizar unas clases predefinidas y ahorrar trabajo, nada más. Podríamos personalizar perfectamente nuestro código CSS.

1.-React - Integración de contenido interactivo.



```
JS App.js # index.css JS index.js
primerproyettoreact > src > # index.css /
1  /*!
2   * Primitive UI | MIT License
3   *
4   * A minimalist front-end design toolkit built with Sass for developing
5   * responsive, browser-consistent web apps.
6   *
7   * Author: Tania Rascia <tania@taniarascia.com>
8   * Source: https://github.com/taniarascia/primitive
9   * Documentation: https://taniarascia.github.io/primitive
10  */
11  /**
12   * Variables
13   *
14   * The majority of the configuration for the toolkit.
15   */
16  /*! normalize.css v8.0.1 | MIT License | github.com/necolas/normalize.css */
17  /* Document
18  | ===== */
```

Cuidado, para empezar a crear el primer proyecto es mejor utilizar la estructura que se plantea en el siguiente apartado “Componentes en React”.

Ahora, en index.js debemos importar React, ReactDOM y el fichero CSS que acabamos de crear (no son necesarios los ; en esta especificación).



```
Help index.js - ProyectoReact - Visual Studio Code
JS App.js # index.css JS index.js X
primerproyettoreact > src > JS index.js
1  import React from 'react'
2  import ReactDOM from 'react-dom'
3  import './index.css'
```

A continuación, vamos a crear el componente App desde 0.

- App va a heredar de Component.
- Nos fijamos en que el div no va a usar “class”. En su lugar utiliza “classname” porque realmente no estamos escribiendo código HTML, estamos escribiendo código JavaScript.

Ahora indicamos que este componente App va a ser la raíz del proyecto:

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Lanzamos el proyecto con el comando: `npm start`

Podemos visualizar el resultado en el navegador:

1.5.- Formas de escribir código en React

Es posible escribir código en React de varias formas, pero la más recomendable es utilizar JSX (JavaScript + XML).

JSX no es HTML, está más cerca de JavaScript.

Debemos tener en cuenta varios aspectos importantes:

- Se utiliza `className` en vez de `class` para aplicar clases CSS (esto es porque `class` es una palabra reservada en JavaScript).
- Las propiedades y métodos se aplicarán con la segunda parte del nombre en mayúscula (camelCase). Por ejemplo: `onclick` se convertirá en `onClick`
- Las etiquetas que se cierran en sí mismas, por ejemplo `img`, lo harán de esta forma: ``

1.6.- Variables en React

Declarar y asignar valores a variables en React utilizando JSX es muy sencillo:

```
const nombre = 'Kosta'
```

```
const encabezado = <h1>Hola, {nombre}</h1>
```

1.7.- Componentes en React

En React prácticamente cualquier cosa es un componente, que puede ser:

- Componentes de clase
- Componentes simples

Estos componentes los explicaremos por separado.

Las aplicaciones pueden estar compuestas de muchos pequeños componentes, de hecho, podríamos cambiar la estructura de la aplicación que acabamos de crear:

En el fichero **index.js** tendríamos:

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
import './index.css'
ReactDOM.render(<App />, document.getElementById('root'))
```

Es decir, vamos a importar un componente `App` desarrollado en otro fichero `App.js`

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

class App extends Component {
  render() {
    return (
      <div className="App">
```

1.-React - Integración de contenido interactivo.

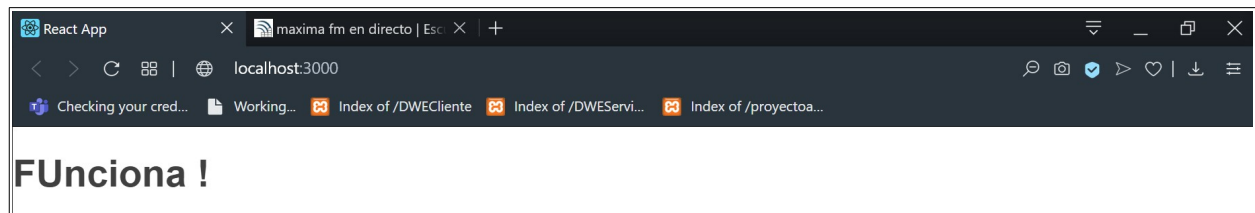
```
    <h1>¡Funciona!</h1>
  </div>
)
}
}
export default App
```

Es fundamental no olvidar que en este fichero debe ir la línea que nos permite exportar el componente “export default App”.

Puede ser interesante separar la aplicación en componentes, aunque no es necesario.

Es fundamental no olvidar que en este fichero debe ir la línea que nos permite exportar el componente “export default App”.

Puede ser interesante separar la aplicación en componentes, aunque no es necesario.



1.8.- Componentes de clase

Estos componentes serán desarrollados en componentes personalizados de clase.

Ya nos habremos fijado en que los componentes que hemos trabajado hasta la fecha están capitalizados, por ejemplo “**App**”.

Es fundamental capitalizar el nombre de los componentes para diferenciarlos de los elementos HTML.

En el siguiente ejemplo vamos a desarrollar un componente Tabla que vamos a rellenar con datos:

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

class Tabla extends Component {
  render() {
    return (
      <table>
        <thead>
          <tr>
            <th>Nombre</th>
            <th>Apellidos</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>Joel</td>
            <td>Edgerton</td>
          </tr>
          <tr>
            <td>Carmen</td>
            <td>Maura</td>
          </tr>
          <tr>
            <td>Luis</td>
            <td>Tosar</td>
          </tr>
          <tr>
            <td>Chloe</td>
          </tr>
        </tbody>
      </table>
    )
  }
}
```

```
        <td>Grace Moretz</td>
      </tr>
    </tbody>
  </table>
)
}
}
export default Tabla
```

Y en el fichero **App.js**

```
import React, { Component } from 'react'
import Tabla from './Tabla'

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>¡Funciona!</h1>
        <Tabla />
      </div>
    )
  }
}

export default App
```



<https://www.taniarascia.com/getting-started-with-react/>

1.9.- Componentes simples

Los componentes simples son básicamente funciones. No utilizan la palabra clave class.

Tomando como referencia el ejemplo anterior, vamos a crear dos componentes simples para formar una tabla:

- Una cabecera para la tabla
- Un body para la tabla

Vamos a utilizar la sintaxis de ES6 para crear estas funciones (funciones flecha, en este caso):

Esto lo vamos a implementar en un fichero **TablaComponentesSimples.js**

Primero el componente simple **TablaCabecera**

Posteriormente el componente simple **TablaCuerpo**

```
import React, { Component } from "react";

const TablaCabecera = () => {
  return (
    <thead>
      <tr>
        <th>Nombre:</th>
        <th>Apellidos:</th>
      </tr>
    </thead>
  )
}

const TablaCuerpo = () => {
  return (
    <tbody>
      <tr>
        <td>Joel</td>
        <td>Edgerton</td>
      </tr>
      <tr>
        <td>Carmen</td>
        <td>Maura</td>
      </tr>
      <tr>
        <td>Chloe</td>
        <td>Grace Moretz</td>
      </tr>
    </tbody>
  )
}
```

```
class TablaComponentesSimples extends Component{
  render(){
    return(
      <table>
        <TablaCabecera/>
        <TablaCuerpo/>
      </table>
    )
  }
}

export default TablaComponentesSimples
```

Hay que recordar que para poder utilizar este componente de clase debemos incluir la línea para exportarlo.

Cuando creamos componentes, nunca hay que olvidar esta instrucción:

Para componentes simples y componentes de clase:

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

Para componentes simples:

```
import React from 'react'
```

1.10.- Props

Son argumentos que se pasan a los componentes en React.

Podríamos decir que **Props** son argumentos que se pasan a los componentes en React. Sería algo similar a los parámetros que pasamos a una función en JavaScript.

Una característica de **Props** es que únicamente son de lectura, es decir, los pasamos como parámetros al componente, pero este componente no puede modificarlos. Si queremos modificar estos datos, tenemos que usar **State**.

Siguiendo el ejemplo de componente con cabecera y cuerpo, en este caso vamos a borrar todos los datos de esta tabla y vamos a crear una colección de datos (array de objetos) basada en JSON:

En **App.js**, concretamente dentro de **render()**, antes de **return**, por supuesto.

```
render(){
  /*Coleccion de objetos con nomenclatura JSON para utilizar con props */
  const actoresActrices = [
    {
      nombre: 'Joel',
      apellidos:'Edgerton'
    },
    {
      nombre: 'Carmen',
```

```
    apellidos: 'Maura'
  },
  {
    nombre: 'Luis',
    apellidos: 'Tosar'
  },
  {
    nombre: 'Chloe',
    apellidos: 'Grace Moretz'
  }
]
```

En **App.js** tenemos que pasar los datos a nuestro componente:

```
<TablaComponentesSimplesProps datosActoresActrices={actoresActrices}/>

return (
  <div className="App">
    <h2>Tabla creada mediante componente de clase que no
      utiliza componentes simples
    </h2>
    <Tabla />
    <h2>Tabla creada mediante un componente de clase que
      utiliza componentes simples
    </h2>
    <TablaComponentesSimples/>
    <h2>Tabla creada mediante un componente de clase que utiliza paso de parámetros con props</h2>
    /* Sesión 34 -> 30 de Enero de 2020 */
    <TablaComponentesSimplesProps datosActoresActrices={actoresActrices}/>
    <TablaComponentesSimplesState datosPersonaje={personajes} borrarPersonaje={this.borrarPersonaje}/>
    <h2>Añadir nuevo personaje</h2>
    <Formulario manejarEnvio={this.manejarEnvio} />
  </div>
)
```

Como hemos pasado los datos a ese componente, tenemos que modificar la parte del propio componente en **TablaComponentesSimplesProps.js**

```
class TablaCamponentesSimplesProps extends Component {
  render() {
    const { datosActoresActrices } = this.props
    return (
      <table>
        <TablaCabecera />
        <TablaCuerpo datosActoresActrices={datosActoresActrices} />
      </table>
    )
  }
}
```

Es importante saber que podemos acceder a todos los props desde un componente mediante **this.props**. En nuestro caso sólo estamos pasando un props, pero para acceder a uno en concreto deberíamos utilizar **props.nombreprops**

Seguimos en **TablaComponentesSimplesProps.js**.

Como podemos observar a continuación, utilizaremos el props como parámetro para el componente simple **TablaCuerpo**. Utilizamos el método map para retornar una fila de tabla por cada objeto del array **datosActoresActrices**. Con índice llevamos la cuenta del número de filas. Hay que tener en cuenta que vamos a utilizar una key para identificar el elemento dentro del array. Las keys sólo se suelen utilizar cuando el orden de estos elementos no va a cambiar.

Todos estos datos los vamos a recoger en una variable **filasDeDatos** que vamos a utilizar en el componente simple.

```
const TablaCuerpo = props => {
  const filasDeDatos = props.datosActoresActrices.map((fila, indice) => {
    return (
      <tr key={indice}>
        <td>{fila.nombre}</td>
        <td>{fila.apellidos}</td>
      </tr>
    )
  })
  return (
    <tbody>
      {filasDeDatos}
    </tbody>
  )
}
```

Seguimos en **TablaComponentesSimplesProps.js**

Nuestro componente de clase, en su parte cargada mediante el componente simple **TablaCuerpo**, va a utilizar el props que hemos pasado desde **App.js**

```
class TablaCamponentesSimplesProps extends Component {
  render() {
    const { datosActoresActrices } = this.props
    return (
      <table>
```



```
        <TablaCabecera />
        <TablaCuerpo datosActoresActrices={datosActoresActrices} />
    </table>
  )
}
}

export default TablaCamponentesSimplesProps
```

Codigo completo de los ejercicios realizados en clase:

```
import React,{ Component } from "react";

/*Sacado de TablaComponentesSimple.js */
const TablaCabecera = () => {
  return (
    <thead>
      <tr>
        <th>Nombre:</th>
        <th>Apellidos:</th>
      </tr>
    </thead>
  )
}

/*En este caso en el cuerpo de la tabla pasaremos parametros, asi se podra const
ruir la tabla con los datos
pasados */
const TablaCuerpo = props => {
  const filasDeDatos = props.datosActoresActrices.map((fila, indice) => {
    return (
      <tr key={indice}>
        <td>{fila.nombre}</td>
        <td>{fila.apellidos}</td>
      </tr>
    )
  })

  return (
    <tbody>
      {filasDeDatos}
    </tbody>
  )
}
```

```
}

class TablaComponentesSimplesProps extends Component {
  render() {
    /* <TablaComponentesSimplesProps datosActoresActrices={actoresActrices}
    /> */
    const { datosActoresActrices } = this.props/*Utilizo props pasaos al com
    ponente desde la aplicacion*/

    return (
      <table>
        <TablaCabecera />
        <TablaCuerpo datosActoresActrices={datosActoresActrices} />
      </table>
    )
  }
}

export default TablaComponentesSimplesProps
```

Es fundamental saber que el componente no puede cambiar los valores del props.

1.11.- State – Permite modificar los datos de un componente

Props tiene una utilidad muy importante, sin embargo no nos permite cambiar los datos de un componente. Para esto utilizamos **state**.

Podemos pensar en **State** como datos que pueden ser guardados y modificados sin necesidad de ser añadidos a una base de datos.

En App.js antes de render()

Vamos a crear un objeto State con los nombres de unos personajes de series.

```
/*Objeto para utilizar en state*/
state = {
  personajes: [
    {
      name: 'Payton Hobart',
    },
    {
      name: 'Wendy Carr',
    },
    {
      name: 'Mina',
    },
  ],
}
```

```
    },  
    {  
      name: 'Jonathan Harker',  
    },  
    {  
      name: 'Drácula',  
    },  
    {  
      name: 'Once',  
    },  
    {  
      name: 'Jim Hopper',  
    }  
  ]  
}
```

En **App.js** antes de **render()**

También vamos a implementar un método para borrar personajes:

```
/*Método para borrar personajes*/  
borrarPersonaje = indice => {  
  const { personajes } = this.state;  
  
  this.setState({  
    personajes: personajes.filter((personaje, i) => {  
      return i !== indice;  
    })  
  });  
};  
}
```

Inicializamos el objeto personajes, ahora dentro de **render()** en **App.js**

```
/*Para utilizar en state*/  
const { personajes } = this.state
```

Props tiene una utilidad muy importante, sin embargo no nos permite cambiar los datos de un componente. Para esto utilizamos state.

1.-React - Integración de contenido interactivo.

Podemos pensar en State como datos que pueden ser guardados y modificados sin necesidad de ser añadidos a una base de datos.

En App.js antes de render()

Vamos a crear un objeto State con los nombres de unos personajes de series.

```
/*Objeto para utilizar en state*/
```

```
state = {  
  personajes: [  
    {  
      name: 'Payton Hobart',  
    },  
    {  
      name: 'Wendy Carr',  
    },  
    {  
      name: 'Mina',  
    },  
    {  
      name: 'Jonathan Harker',  
    },  
    {  
      name: 'Drácula',  
    },  
    {  
      name: 'Once',  
    },  
    {  
      name: 'Jim Hopper',  
    }  
  ]  
}
```

En App.js antes de `render()`

También vamos a implementar un método para borrar personajes:

```
/*Método para borrar personajes*/
borrarPersonaje = indice => {
  const { personajes } = this.state;

  this.setState({
    personajes: personajes.filter((personaje, i) => {
      return i !== indice;
    })
  });
};
}
```

Inicializamos el objeto `personajes`, ahora dentro de **`render()`** en **App.js**

```
/*Para utilizar en state*/
const { personajes } = this.state
```

Ahora, dentro de **`return ()`**, en **App.js**, pintamos el componente simple (para trabajar con state es conveniente convertir el componente de clase en uno simple):

```
<h1>Tabla creada con un componente simple (con state y que permite eliminar elementos) que utiliza dos componentes simples</h1>
  <TablaComponentesSimplesState datosPersonaje={personajes} borrarPersonaje={this.borrarPersonaje} />
```

Ahora debemos crear un componente simple **TablaComponentesSimplesStat.js**

El inicio es idéntico a los otros componentes de Tabla que hemos creado:

```
import React from 'react'
```

```
const TablaCabecera = () => {  
  return (  
    <thead>  
      <tr>  
        <th>Nombre</th>  
        <th>Apellidos</th>  
      </tr>  
    </thead>  
  )  
}
```

Ahora debemos modificar el componente simple TablaCuerpo con el acceso a los datos state:

```
const TablaCuerpo = props => {  
  const filasDeDatos = props.datosPersonaje.map((fila, indice) => {  
    return (  
      <tr key={indice}>  
        <td>{fila.name}</td>  
        <td><button onClick={() => props.borrarPersonaje(indice)}>Borrar  
</button></td>  
      </tr>  
    )  
  }  
)  
  return (  
    <tbody>  
      {filasDeDatos}  
    </tbody>  
  )  
}
```

Convertimos el componente de clase en un componente simple para poder utilizar los datos state:

```
const TablaComponentesSimplesState = (props) => {  
  
  const { datosPersonaje, borrarPersonaje } = props;  
  return (  
    <table>  
      <TablaCabecera />  
      <TablaCuerpo datosPersonaje={datosPersonaje} borrarPersonaje={bo  
rrarPersonaje} />  
    </table>  
  )  
}  
  
export default TablaComponentesSimplesState
```

Ahora ya tenemos nuestro componente funcional en el que podemos eliminar datos:

Tabla creada con un componente simple (con state y que permite eliminar elementos) que utiliza dos componentes simples

Nombre	Apellidos
Payton Hobart	<button>Borrar</button>
Wendy Carr	<button>Borrar</button>
Mina	<button>Borrar</button>
Jonathan Harker	<button>Borrar</button>
Drácula	<button>Borrar</button>
Once	<button>Borrar</button>

1.12.- Tratamiento de informacion en ficheros JSON

En vez de crear un objeto de datos con formato JSON, tal vez necesitemos utilizar un fichero JSON local. Para realizar esto debemos guardar los datos que deseemos en un fichero con extensión “.json”. Los datos deben tener el siguiente formato en caso de que tengamos un array de objetos (si no tuviéramos un array, los [] sobrarían):

```
[
  {
    "name": "Payton Hobart"
  },
  {
    "name": "Wendy Carr"
  },
  {
    "name": "Mina"
  },
  {
    "name": "Jonathan Harker"
  },
  {
    "name": "Drácula"
  },
  {
    "name": "Once"
  },
  {
    "name": "Jim Hopper"
  }
]
```

Una vez que tenemos listo el fichero JSON, es necesario importarlo en el fichero “.js” en el que vayamos a utilizarlo en React:

```
import personajes from './personajes.json'
```

Para utilizar estos datos mediante State (admiten modificaciones) en React usaremos la expresión:

```
state = { personajes }
```


No me carga los datos del fichero .json , marca error....

1.13.- Envío de formularios de datos

En **App.js** vamos a añadir estos cambios:

Antes de render()

```
/*Voy a utilizar una colección vacía para luego ir rellenándola mediante un formulario*/  
  
state = { personajes: [] }
```

Un método para manejar los datos del Formulario:

```
/*Para manejar los datos del formulario*/  
manejarEnvio = personaje => {  
  this.setState({personajes: [...this.state.personajes, personaje]});  
}
```

El propio componente dentro de return ()

```
<h2>Añadir nuevo</h2>  
  <Formulario manejarEnvio={this.manejarEnvio} />
```

Importamos el componente al principio del fichero:

```
import Formulario from './Formulario'
```

A continuación vamos a crear un **componente “Formulario.js”**:

```
constructor(props) {  
  super(props);
```

```
    this.initialState = {
      name: ''
    };

    this.state = this.initialState;
  }

  manejarCambio = evento => {
    const { name, value } = evento.target;

    this.setState({
      [name] : value
    });
  }

  enEnvioFormulario = (evento) => {
    evento.preventDefault();

    this.props.manejarEnvio(this.state);
    this.setState(this.initialState);
  }

  render() {
    const { name } = this.state;

    return (
      <form onSubmit={this.enEnvioFormulario}>
        <label for="nombreid">Nombre</label>
        <input
          type="text"
          name="name"
          id="nombreid"
          value={name}
          onChange={this.manejarCambio} />
      </form>
    );
  }
}
```

```
        <button type="submit">
            Enviar Datos
        </button>
    </form>
);
}
}

export default Formulario
```

Tabla creada con un componente simple (con state y que permite eliminar elementos) que utiliza dos componentes simples

Nombre	Apellidos
--------	-----------

Tabla creada con Componente Simple State y datos exportados de un fichero .json

Añadir nuevo

Nombre

Kostadin

Enviar Datos

Tabla creada con un componente simple (con state y que permite eliminar elementos) que utiliza dos componentes simples

Nombre	Apellidos
--------	-----------

Kostadin

Borrar

Tabla creada con Componente Simple State y datos exportados de un fichero .json

Añadir nuevo

Nombre

Gonzalo

Enviar Datos

1.14.- Obtener datos de una API

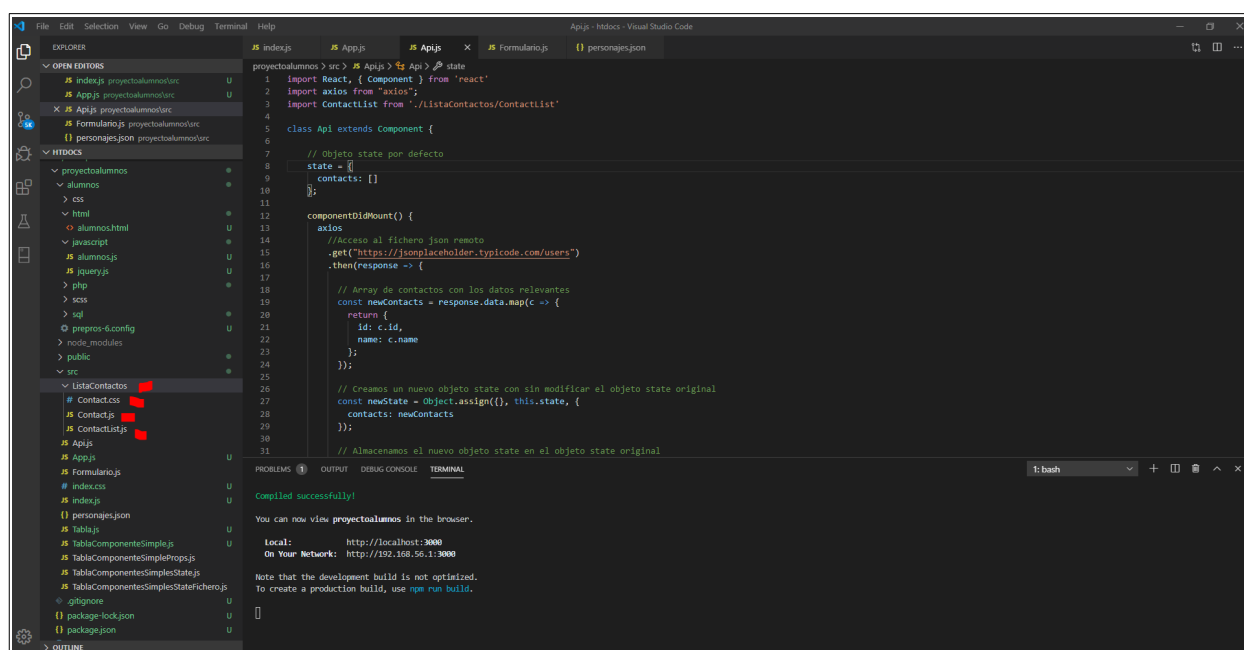
Se trata de servicios de otras aplicaciones que podemos integrar en nuestro software.

En este caso, vamos a obtener datos de la API de Andreas Reiterer. Tendremos una lista de clientes almacenados en la siguiente dirección: [enlace](#).

Es fundamental realizar las importaciones necesarias:

- Las que acostumbramos.
- **Axios** para tratar peticiones AJAX (necesitamos instalar el paquete **npm i -S axios**)
- Unos componentes que simplemente formatean una lista de contactos (tenemos estos ficheros en la carpeta de la UT5 en Teams).

Una vez descargados los ficheros facilitados en la UT5 se incorporan en el proyecto **React** en la carpeta **“src”**.



Documentos descargados.

1. Contact.css:

```
.contact {  
  margin: 10px;  
  padding: 10px;  
  border: 1px solid #bbb;  
  background-color: #eee;  
}  
  
.contact span {  
  font-size: 1.2em;  
  text-decoration: none;  
  color: #333;  
}
```

2. Contact.js

```
import React from "react";  
import PropTypes from "prop-types";  
import "./Contact.css";  
  
function Contact(props) {  
  return (  
    <div className="contact">  
      <span>{props.name}</span>  
    </div>  
  );  
}  
  
Contact.propTypes = {  
  name: PropTypes.string.isRequired  
};  
  
export default Contact;
```

3. ContactList.js

```
import React from "react";
import PropTypes from "prop-types";

// import the Contact component
import Contact from "../Contact";

function ContactList(props) {
  return (
    <div>{props.contacts.map(c => <Contact key={c.id} name={c.name} />)}</div>
  );
}

ContactList.propTypes = {
  contacts: PropTypes.array.isRequired
};

export default ContactList;
```

Desarrollamos un componente Api.js

```
import React, { Component } from 'react'
import axios from "axios";
import ContactList from '../ListaContactos/ContactList'
class Api extends Component {

  // Objeto state por defecto
  state = {
    contacts: []
  };

  componentDidMount() {
    axios
      //Acceso al fichero json remoto
      .get("https://jsonplaceholder.typicode.com/users")
      .then(response => {

        // Array de contactos con los datos relevantes
```

```
const newContacts = response.data.map(c => {
  return {
    id: c.id,
    name: c.name
  };
});

// Creamos un nuevo objeto state con sin modificar el objeto state original
const newState = Object.assign({}, this.state, {
  contacts: newContacts
});

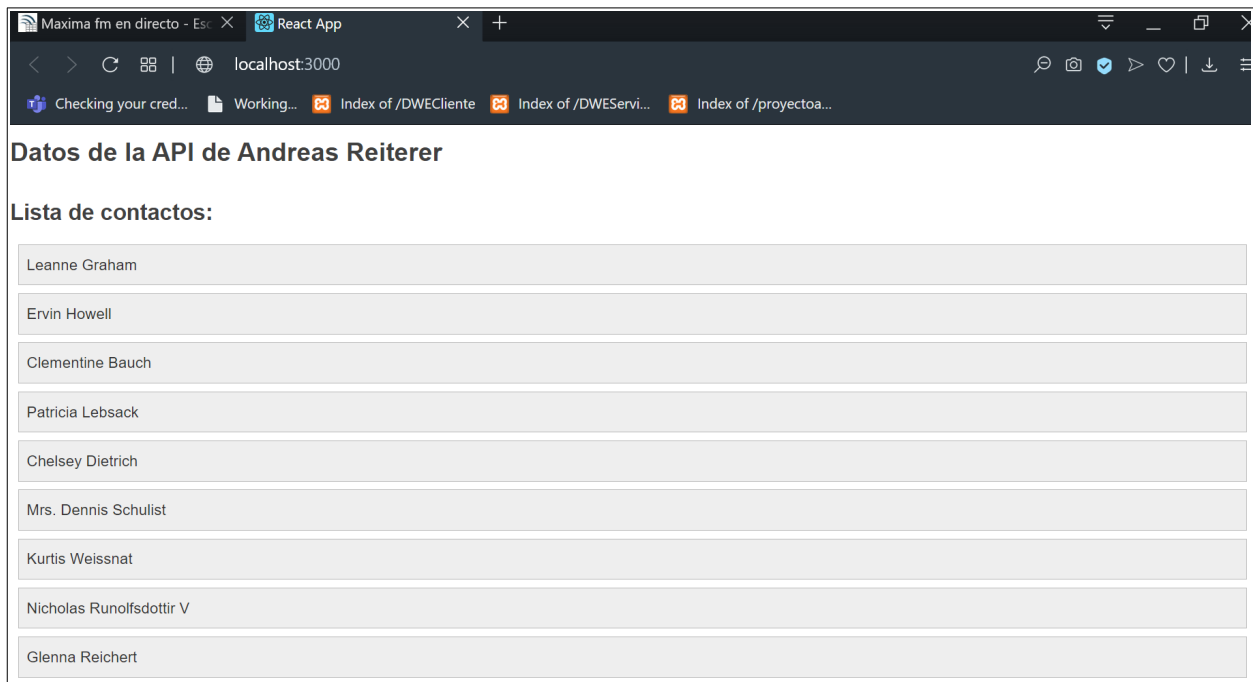
// Almacenamos el nuevo objeto state en el objeto state original
this.setState(newState);
})
.catch(error => console.log(error));
}

render() {
  return (
    <div className="App">
      <ContactList contacts={this.state.contacts} />
    </div>
  );
}

export default Api
```

Utilizamos el componente en App.js

```
<h2>Datos de la API de Andreas Reiterer</h2>
<h3>Lista de contactos:</h3>
<Api />
```

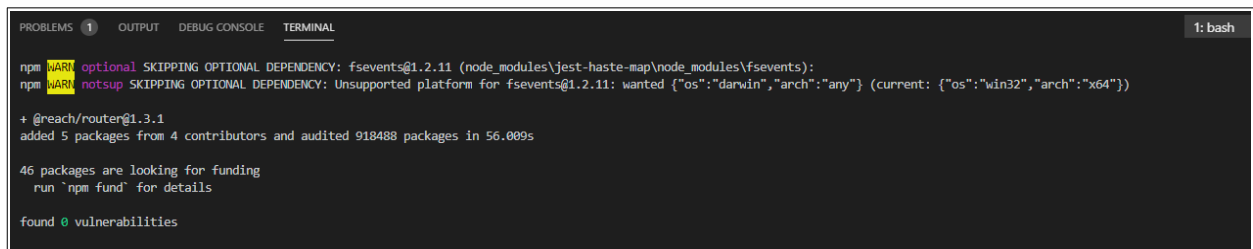


1.15.- Manejo de rutas en React

Para manejar rutas en React podemos utilizar el módulo “Reach Router”. Tenemos mucha información [aquí](#).

Para instalar este módulo debemos lanzar este comando:

- **npm install @reach/router**



Es fundamental realizar las siguientes importaciones:

```
import React from "react";
import { render } from "react-dom";
import { Router, Link } from "@reach/router";
```


En este caso vamos a realizar la aplicación en un solo fichero index.js en el que vamos a renderizar el componente simple principal “Aplicacion”

Componente principal “Aplicacion”. Lo principal que debemos reseñar es que las rutas las tenemos que definir en <Router>, para lo que necesitamos el módulo “Reach Router” instalado. También podemos observar que hemos desarrollado un menú muy simple en <nav> y que hemos definido cada uno de los enlaces a las rutas mediante <link>.

```
const Aplicacion = () => (  
  <div>  
    <h1>Manejo de rutas en React</h1>  
    <nav>  
      <Link to="/">Inicio</Link>{" "  
      <Link to="SobreNosotros">Sobre nosotros</Link>  
    </nav>  
  
    <Router>  
      <Inicio path="/" />  
      <SobreNosotros path="/SobreNosotros" />  
    </Router>  
  </div>  

```

Luego tendremos otros dos componentes simples “Inicio” y “SobreNosotros” que serán invocados desde “Aplicación”

```
const Inicio = () => (  
  <div>  
    <h2>Estás en inicio</h2>  
  </div>  
  
const SobreNosotros = () => (  
  <div>  
    <h2>Estás en Sobre Nosotros</h2>  
  </div>  

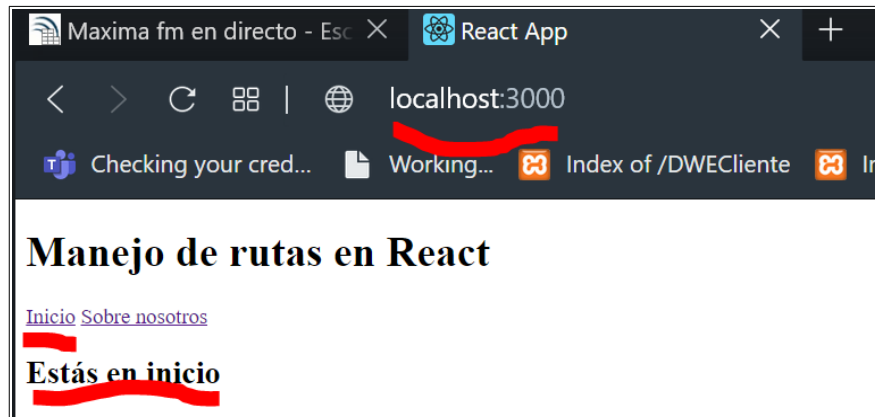
```

1.-React - Integración de contenido interactivo.

Por último, vamos a renderizar el componente “Aplicacion” en root, como estamos acostumbrados:

```
render(<Aplicacion />, document.getElementById("root"));
```

Inicio:



Sobre nosotros:



1.16.- Construir y desplegar una aplicación en React

Una vez que hemos realizado las tareas requeridas en el entorno de desarrollo, es hora de aplicar lo necesario para obtener nuestra aplicación para utilizar en producción, es decir, lo que vamos a alojar en nuestro servidor web. Es posible que queramos realizar modificaciones en el fichero index.html (el title, por ejemplo).

Para realizar la optimización de cara a producción utilizaremos el comando:

- **npm run build**

El contenido estático listo para mover a cualquier localización o para alojar en nuestra web estará en el directorio “build”.

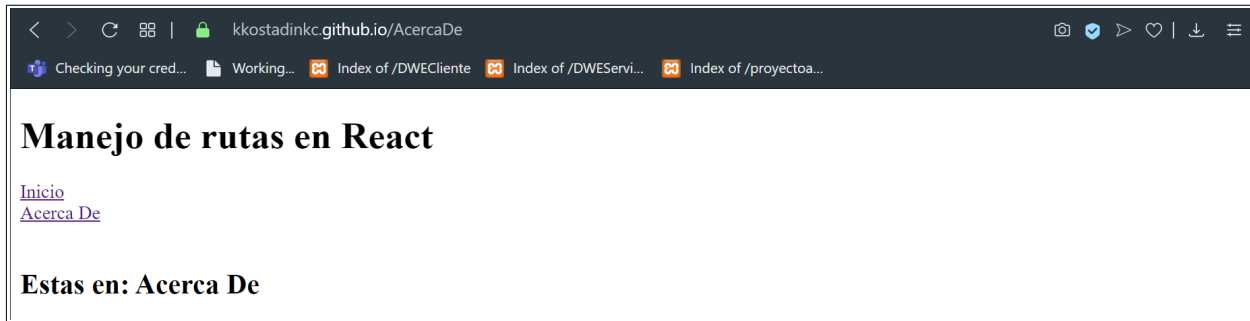
Es posible que al obtener el paquete con todos los contenidos estáticos no consigamos ningún resultado al abrir la página index.html

Esto se debe que es necesario especificar que todos los recursos dependientes del proyecto van a tomar como ruta relativa la localización de index.html

Para llevar a cabo esto debemos añadir esto en el fichero package.json

"homepage": ".",

Más información [aquí](#).



1.17.- Ejercicios

Ejercicio 1

-> Nuevo proyecto React: proyectocomponenteslayout

-> Crear varios componentes de clase o simples para utilizar en la Aplicación Principal que permita

realizar una distribución mediante Grid:

-> Header

-> Main

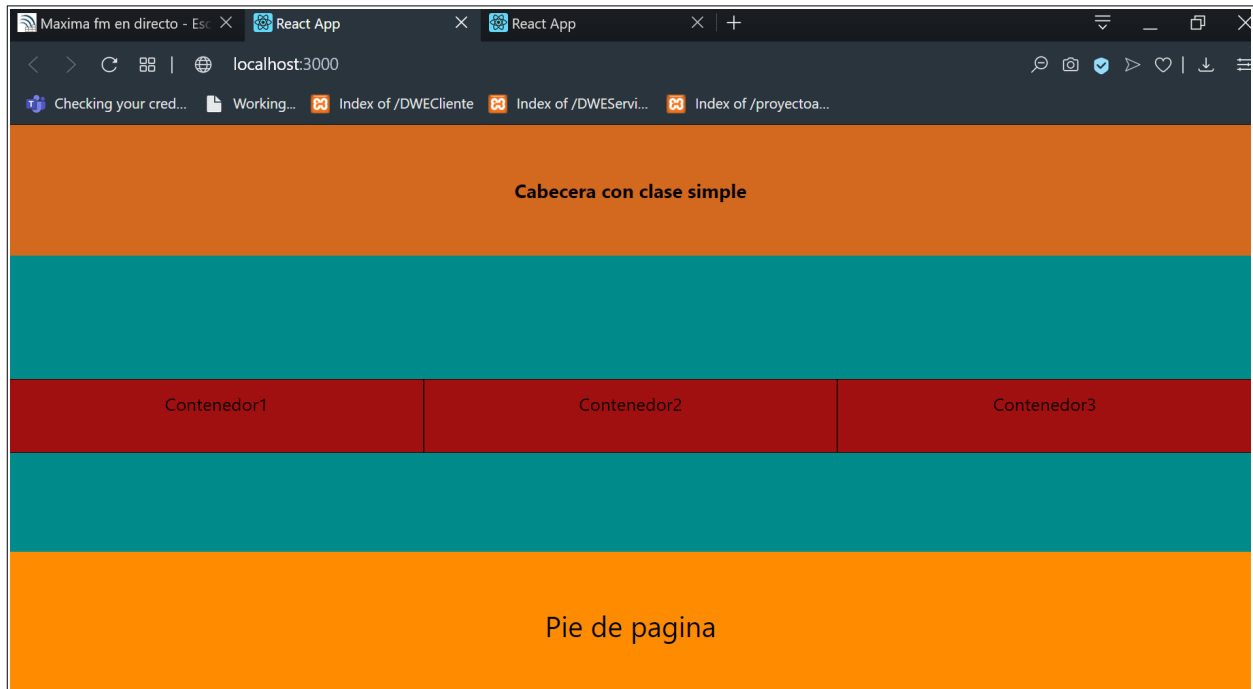
- 3 capas que entre todas ocupen el 100% del ancho del navegador

-> Footer

-> Realizar las hojas de estilo utilizando Sass

-> Gestionar las versiones de Software con Git

1.-React - Integración de contenido interactivo.



2.- UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

2.1.- Selectores de atributos

Los selectores de atributos se usan para dar estilos a elementos que tengan un atributo específico.

Primero creamos la clase **SelectorAtributo.js** que tendrá la siguiente estructura

```
import React, {Component} from 'react'
/* import Imagen from 'react-image' */

class SelectorAtributo extends Component{

  render(){
    return(
      <div>
        <a href="https://www.google.com" rel="noopener noreferrer"
">Enlace a Google SIN target</a>
        <br />
        <a href="https://www.google.com" target="_blank" rel="noopener n
orereferrer">Enlace a Google CON target</a>
        <br />
        <a href="https://www.google.com" target="_self" rel="noopener no
referrer">Enlace a Google CON target="_self"</a>
        <br />
        {/* <Imagen alt="Foto del K2" src={require('./img/
k2invernal.jpg')} /> */}

        <p className="claseparrafo-1">Este párrafo tiene la clase="clase
parrafo1"</p>
        <p className="claseselectorUno">Este párrafo tiene la clase="cla
seselectorUno"</p>
      </div>
    )
  }
}

export default SelectorAtributo
```

2.-UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

Ejemplo:

- Los elementos `<a>` que tengan el atributo **target** llevarán asociado un color de fondo rojo.

```
a[target] {  
  background-color: red;  
}
```

Selector de Atributo

[Enlace a Google SIN target](#)

[Enlace a Google CON target](#)

[Enlace a Google CON target="_self"](#)

Este párrafo tiene la clase="claseparrafo1"

Este párrafo tiene la clase="claseselectorUno"

Se puede ir aún más allá y especificar los elementos que cumplan un determinado **valor de atributo**:

Ejemplo:

- Los enlaces que tengan el valor **_blank** en el atributo **target** llevarán asociado un color de fondo naranja.

```
a[target="_self"] {  
  background-color: royalblue;  
}
```

Selector de Atributo

[Enlace a Google SIN target](#)

[Enlace a Google CON target](#)

[Enlace a Google CON target="_self"](#)

Este párrafo tiene la clase="claseparrafo1"

Este párrafo tiene la clase="claseselectorUno"

También se pueden seleccionar **los elementos que tengan una palabra determinada en el valor de un atributo.**

Ejemplo:

- Hacemos un ejemplo con varias imágenes que tengan Boneflower en su atributo title.
- El selector del ejemplo seleccionaría todos los elementos que tengan la palabra Boneflower en el título. La palabra debe aparecer independiente de otro texto.

```
[title~="Boneflower"] {  
    border: 5px solid yellow;  
}
```

Otra de las posibilidades que ofrece el **selector de atributos es incluso seleccionar los elementos que tengan una determinada palabra en el inicio del valor que toma el atributo.** La palabra debe ser única y puede estar seguida de un separador como un guion (-). Es sensible a mayúsculas y minúsculas.

Ejemplo:

- Seleccionaría todos los elementos cuyo atributo class empiece por la palabra claseimagen

```
[class|=claseimagen] {  
    background: yellow;  
}
```

Otra opción es que nos ofrecen los selectores de atributos es poder seleccionar **los elementos cuyo valor de atributo comience por una palabra, pero hace falta que sea única y puede estar seguida de cualquier conjunto de caracteres.**

Ejemplo:

```
.claseparrafo-1 {  
    font: 30px monospace;  
}
```

También es posible seleccionar los **elementos con un atributo cuyo valor termine con un valor específico.** No hace falta que sea una palabra única o precedida de un separador como guion (-).

Ejemplo:

```
[class$="gonsansal"] {  
    background: yellow;  
}
```

2.-UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

Otra opción es seleccionar los **elementos** cuyo **valor de atributo** contenga una **determinada palabra** que puede estar situada en cualquier parte del valor del atributo.

Ejemplo:

```
[class*="te"] {  
    background: yellow;  
}
```

2.2.- Box-sizing

La propiedad box-sizing se utiliza para incluir el padding y el border de un elemento en su anchura y altura total.

Valores que puede tomar:

- **Content-box:** es el valor por defecto. El padding y el border no están incluidos en el ancho y el alto del elemento.
- **Border-box:** la anchura y altura del elemento contienen el padding y el border.

Ejemplos:

1. Capa sin aplicar box-sizing

```
.capa1sinboxs {  
    width: 300px;  
    height: 100px;  
    padding: 50px;  
    border: 1px solid red;  
}
```

2. Capa con box-sizing aplicado.

Vemos cómo se activa el padding pero no se añade a su altura ni a su anchura.

```
.capa2conboxs {  
    width: 300px;  
    height: 100px;  
    padding: 50px;  
    border: 1px solid yellow;  
    box-sizing: border-box;  
}
```

Podemos aplicarlo a todos los elementos:

```
* {  
    box-sizing: border-box;
```



```
}
```

2.3.- Formularios– Estilos en formularios

Mediante CSS se puede formatear al detalle los formularios y hacerlos mucho más atractivos.

Clase de formulario

Ejemplos con la propiedad width:

Todos los campos de tipo input tendrán anchura del 100%

```
input {  
    width: 100%;  
}
```

Todos los campos de tipo input text tendrán una anchura del 60%

```
input[type=text]{  
    width: 60%;  
}
```

Se puede utilizar padding para añadir espacio interior a los campos:

```
input[type=text] {  
    width: 100%;  
    padding: 12px 20px;  
    margin: 8px 0;  
    box-sizing: border-box;  
}
```

También se pueden personalizar los bordes:

Podemos incluso sólo mostrar el borde inferior

```
input[type=text] {  
    border: none;  
    border-bottom: 2px solid red;  
}
```

2.-UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

```
import React, {Component} from 'react'
class TrabajoConFormulario extends Component {
  render() {
    return (
      <div>
        <form name="formulario1" id="formulario1id" action="">
          <fieldset>
            <legend>Formulario en React</legend>
            <label for="campotexto1id">Nombre:</label>
            <input type="text" name="campotexto1" id="campotexto1id"
placeholder="Introduce tu nombre"></input><br />
            <label for="campotexto2id">Busqueda:</label>
            <input type="text" className="claseinput1" name="campotex
to2" id="campotexto2id" placeholder="Término de busqueda"></input>
          </fieldset>
        </form>
      </div>
    )
  }
}
export default TrabajoConFormulario
```

Un aspecto muy interesante es cambiar el aspecto cuando hacemos **foco sobre el elemento** (utilizamos la pseudoclase **:focus**), incluso podemos añadir una animación con la propiedad **transition** (necesitamos añadir **webkit** para navegadores antiguos):

```
input[type="text"] {
  width: 60%;
  padding: 12px 20px;
  margin: 8px 0;
  box-sizing: border-box;
  border: 3px solid #ccc;
  -webkit-transition: 1s;
  transition: 1s; //Añade un efecto de animación que se prolonga durante x seg
undos
  outline: none;
}
```

```
input[type="text"]:focus {
  width: 100%;
  border: 10px solid greenyellow;
}
```

Trabajo con Formularios

Formulario en React

Nombre:

Busqueda:

Otra opción muy interesante es situar imágenes de fondo en los campos, por ejemplo, para crear un campo de búsqueda:

Realizar ejemplo con Icono de Búsqueda – Situado en recursos.

La base para llevar a cabo esto será utilizar la propiedad `background-image` para poner la imagen de fondo y usar la propiedad `background-position` para situar la imagen donde queramos. También vamos a utilizar otra propiedad (`border-radius`) para dar curvatura a los bordes.

```
<form>
  <input type="text" name="busqueda" placeholder="Buscar..">
</form>
```

También se pueden crear cuadros de búsqueda con animaciones:

```
input[type="text"].claseinput1 {
  box-sizing: border-box;
  border: 2px solid #ccc;
  border-radius: 4px;
  font-size: 16px;
  background-color: white;
  background-image: url('../img/busqueda.png');
  background-position: 10px 10px;
  background-repeat: no-repeat;
  padding: 12px 20px 12px 40px;
  -webkit-transition: width 0.4s ease-in-out;
  transition: width 0.4s ease-in-out;
}
```

Trabajo con Formularios

Formulario en React

Nombre:

Busqueda:

2.-UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

Puede resultar interesante formatear los campos textarea. Mediante la propiedad `resize` podemos impedir que el área de texto sea modificada en anchura y altura por el usuario.

Ejemplo:

```
select {  
  width: 100%;  
  padding: 16px 20px;  
  border: none;  
  border-radius: 4px;  
  background-color: #f1f1f1;  
}
```

```
<form>  
  <select id="disco" name="selectdiscos">  
    <option value="hellfire">Edguy - Hellfire Club</option>  
    <option value="ecliptica">Sonata Arctica - Ecliptica</option>  
    <option value="karma">Kamelot -Karma</option>  
  </select>  
</form>
```

Trabajo con Formularios

Edguy - Hellfire Club

Edguy - Hellfire Club

Sonata Arctica - Ecliptica

Kamelot -Karma

2.4.- Etiquetado campo radio

<!--Ejemplo de etiquetado de un campo radio-->

<fieldset>

<legend>Género:</legend>

<input type="radio" name="sexo" id="masculino" value="M">

<label for="masculino">Masculino</label><!--Permite hacer clic en la etiqueta para activar el radio. Es importante de cara a la accesibilidad-->

<input type="radio" name="sexo" id="femenino" value="F">

<label for="femenino">Femenino</label><!--Permite hacer clic en la etiqueta para activar el radio. Es importante de cara a la accesibilidad-->

</fieldset>

2.5.- Transition

Esta propiedad permite que se muestre el cambio de otra propiedad de manera progresiva, de tal forma que tenga un efecto de animación.

Se pueden aplicar diferentes formas de implementar una transición

- **ease** – especifica un efecto de transición con un inicio lento, posteriormente rápido, y que **finaliza** de nuevo de forma lenta. Es la opción por defecto.
- **linear** – especifica un efecto de transición con la misma velocidad de principio a fin.
- **ease-in** – especifica una transición con un inicio lento
- **ease-out** – especifica una transición con una finalización lenta.
- **ease-in-out** – especifica una transición con un inicio y una finalización lentas.

Ejemplo (se puede aplicar animación tanto en altura como en anchura):

```
.animacionAltura {
  width: 100px;
  height: 100px;
  background: rgb(47, 0, 255);
  -webkit-transition: width 2s, height 4s;
  /* Para Safari 3.1 a 6.0 */
  transition: width 2s, height 4s;
}
.animacionAltura:hover {
  width: 300px;
  height: 300px;
}
```

Posición inicial:

2.-UT6 - Diseño de webs accesibles e implementación de usabilidad en la web

Animaciones de altura

Hola soy
un div
animado

Posicion final al posicionar el raton encima:

Animaciones de altura

Hola soy un div animado

2.6.- Distribución mediante grid

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

class Distribucion extends Component{

  render(){
    return(
      <div className="contenedorGrid">
        <header className="contenedorGrid__cabecera">
          <nav className="contenedorGrid__cabecera__menu">{/* nav es una etiqueta semántica HTML 5 para hacer referencia a un menú de navegación */}
            <div>Item 1</div>
            <div>Item 2</div>
            <div>Item 3</div>
          </nav>
        </header>
        <aside className="contenedorGrid__aside1">

        </aside>
        <main className="contenedorGrid__main">

        </main>
        <aside className="contenedorGrid__aside2"></aside>
        <footer class="contenedorGrid__pie"></footer>
      </div>
    )
  }
}

export default Distribucion
```

```
//scss relativo a la distribucion con header, aside, main y footer
.contenedorGrid{

    grid-template-columns: 20% 60% 20%;
    display: grid;
    border:1px solid black;
    box-sizing: border-box;
    /*      width:100%; */

    &__cabecera, &__pie{

        grid-column: 1/4;
        border:1px solid black;
        height: 200px;
        min-height: 200px;
        padding: 20px 0;
        display: flex;
        justify-content: center;

        &__menu{
            display: flex;
            border: 2px solid black;
            width: 80%;
            >div{
                width: 1/3*100%;
                border: 1px black solid;
            }
        }
    }
    &__aside1{
        grid-column: 1/2;
        border:1px solid black;
        height: 600px;
    }
    &__main{
        grid-column: 2/3;
        border:1px solid black;
        height: 600px;
    }

    &__aside2{
        grid-column: 3/4;
        border:1px solid black;
        height: 600px;
    }
}
```


Distribucion de mediante grid-template-column

Item 1			Item 2			Item 3		

2.7.- Medida FR

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

class Medidafr extends Component{
  render(){
    return(
      <div className="grid">
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
        <div className="grid__column"></div>
      </div>
    )
  }
}

export default Medidafr
```

```
/*Medida FR*/
html, body {
  height: 100%;
  margin: 0; padding: 0;
}
.grid {
  display: grid;
  grid-template-columns: 400px repeat(12, 1fr);
  grid-column-gap: 10px;
  background-color: rgba(255, 0, 0, 0.7);
  height: 400px;
  &__column {
    font-family: "Output Sans", -apple-system, BlinkMacSystemFont, "Segoe UI",
"Roboto", "Oxygen", "Ubuntu", "Cantarell",
    "Fira Sans", "Droid Sans", "Helvetica Neue", sans-serif;
    color: white;
    text-transform: uppercase;
    font-size: 9px;
    background-color: rgba(0, 0, 0, 0.2);
    justify-content: center;
    align-items: center;
    display: flex;
  }
}
```

