

Unidad 2: TUTORIAL WORDLE

## Introduccióm

Este tutorial no asume ningún conocimiento previo en React, pero si en HTML y CSS. Los pasos que aprenderás en este tutorial son fundamentales para entender cómo funciona React.

Este tutorial está dividido en distintas secciones:

1. Instalación del entorno de trabajo, te guiará en la preparación de tu ordenador para completar este tutorial
2. Resumen, te enseñará los fundamentos de **React** : componentes, propiedades y estados
3. Desarrollando el Juego, te enseñará técnicas comunes en la programación con React
4. Mejoras, te propondrá pequeños cambios que te permitirá poner en práctica lo aprendido.

### ¿Qué estamos construyendo?

En este tutorial vamos a crear un juego WORDLE, en el que se le ofrece al jugador 6 intentos para adivinar una palabra de 5 caracteres. En cada intento introducido se le revela los caracteres del intento que también están en la palabra (amarillo) y los que están en la palabra y en la posición correcta (verde). <https://www.nytimes.com/games/wordle/index.html>

Puedes ver como este funciona aquí:

## PREPARACIÖN ENTORNO DE TRABAJO

1. Comprueba si tienes Node.js instalado en tu ordenador abriendo un terminal y escribiendo

>> node -v

18.14.00

1. En caso de que no tengas instalada una instancia de la versión 18 , instálala y actívala para que sea la versión que usemos

>> nvm install 18.14.0

nvm use 18.14.0

1. Crearemos nuestro proyecto react wordle en la unidad de almacenamiento elegida

(/Users/irenecid/practicas)>> npx create-react-app wordle

1. Abriremos *Visual Studio Code* en nuestro nuevo proyecto creado, para ello sobre *Visual Studio Code* iremos al menú  *Archivo -> Abrir carpeta.* y debemos ver algo parecido a esto.

Captura de pantalla de un teléfono celular

Descripción generada automáticamente

El comando ha creado la estructura típica de un proyecto react. El código de la aplicación está en el directorio src, los módulos que necesita la aplicación se instalan en *node-modules*, los recursos gráficos o videos etc se colocarán en la carpeta public.

El fichero *package.json* sobre el que profundizaremos más adelante es el punto de arranque de nuestro proyecto, y contiene en la sección de scripts los comando para arrancar la aplicación:

Texto

Descripción generada automáticamente

Sobre el menú de *Visual Studio Code* pulsaremos *Terminal -> Nuevo Terminal* y escribiremos el comando que arranca nuestra aplicación *npm start*

Texto

Descripción generada automáticamente

Veremos nuestra aplicación sobre la que crearemos nuestro juego en el navegador. Fíjate que la URL es <http://localhost:3000>. Para pararla pulsaremos Ctrl + C

Imagen que contiene Interfaz de usuario gráfica

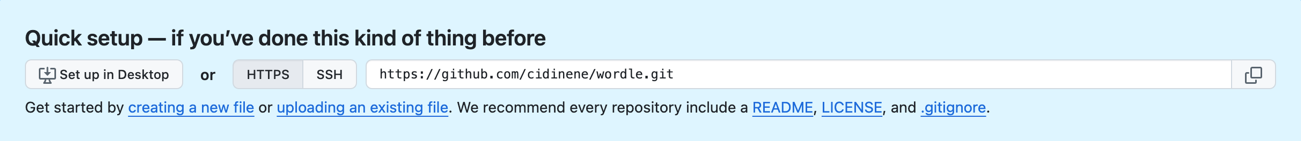
Descripción generada automáticamente

1. Antes de empezar con la programación de nuestra aplicación vamos a añadir a nuestro proyecto un sistema de control de versiones remoto , git. Para ello, Crearemos un repositorio en **github** denominado wordle. Para ello, en la página de github de nuestra cuenta, en la pestaña repositorios, pulsaremos sobre el botón **New** e introduciendo un nombre como wordle a nuestro repositorio.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

1. Copiaremos la ruta del repositorio, en mi caso [*https://github.com/cidinene/wordle.git*](https://github.com/cidinene/wordle.git)



En el terminal , escribiremos de Visual Studio Code introducirmos los siguientes comandos, sustituyendo la ruta de mi repositorio por el tuyo (en rojo) que acabade copia en el paso anterior

git init

git add .

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/cidinene/wordle.git

git push -u origin main

Veremos que nuestro proyecto ya está en github con su rama main

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

## Resumen

En nuestro editor , en la sección de ficheros, en la carpeta src, podemos ver que se han creado un fichero App.js

import logo from './logo.svg';

import './App.css';

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.js</code> and save to reload.

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

</a>

</header>

</div>

);

}

export default App;

Como vemos, este documento devuelve la renderización html de la página principal que se muestra al ejecutar nuestra aplicación y cuyo estilo se encuentra referenciado en el fichero de estilos App.css

Para nosotros , un componente en React es una función que devu

elve el renderizado del componente.

### Dibujando nuestra pantalla a través de componentes

Vamos a crear un componente Board cuyo renderizado nos muestre el tablero principal ,

Para ello vamos a crear por cada cuadrado un componente Square y cada fila una consecución de 5 Square.

Square.js

import './Square.css';

function Square() {

return (

<input className="cuadrado"></input>

);

}

export default Square;

Square.css

.cuadrado

{

color:#ffff;

width: 40px;

height: 50px;

background-color: #666;

display: inline-block;

border:1px solid white;

text-align: center;

font-size: 2.5em;

margin: 4px;

}

.cuadrado:focus

{

border:1px solid yellow;

background-color: #ffffff;

}

Vamos a cambiar la página principal para que contenga una región para el tablero con 5 filas de 5 componentes Square ( cada fila un div row) . Cambia el código por el siguiente:

App.js

import './App.css';

import Square from './Square';

function App() {

return (

<div className="App">

<header className="App-header">

<p>

WORDLE

</p>

</header>

<p>

<div className='board'>

<div id="row" className='row' >

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

</div>

<div id="row" className='row' >

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

</div>

<div id="row" className='row' >

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

</div>

<div id="row" className='row' >

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

</div>

<div id="row" className='row' >

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

<Square></Square>

</div>

</div>

</p>

</div>

);

}

export default App;

Vamos a modificar un par de estilos de la clase principal para que el header no ocupe toda la pantalla y para que el fondo de la aplicación sea negro

App.css

App.css

.App {

text-align: center;

background-color: black;

}

.App-logo {

height: 40vmin;

pointer-events: none;

}

.App-header {

background-color: #282c34;

display: flex;

borramos la altura

flex-direction: column;

align-items: center;

justify-content: center;

font-size: calc(10px + 2vmin);

color: white;

}

.App-link {

color: #61dafb;

}

.row

{

text-align: center;

display: flex;

justify-content: center;

}

Si ejecutas la aplicación deberías poder ver algo como esto:

Tabla

Descripción generada automáticamente con confianza media

### Propiedades del componente

En wordle solo están activas las celdas de la fila que corresponde con el intento actual. En caso de que el intento actual sea el primero, solo estará activa esta fila. Cuando se rellenan los 5 dígitos de la fila, se procede a comprobar si la letra de la celda coincide con el de la palabra buscada, pintándose en verde en caso de que coincida el carácter y la posició , en amarillo si el carácter está incluido en la solución pero en distinta posición.

Para que el cuadrado pueda saber cómo pintarse necesita saber la fila en la que está.

Vamos para ello a incluir dos propiedades a cada cuadrado **row** y **column**, que se pasarán como parámetros del componente.

Ejemplo de código para las celdas de la primera fila:

App.js

<div id="row" className='row' >

<Square row="1" column="1"></Square>

<Square row="1" column="2"></Square>

<Square row="1" column="3"></Square>

<Square row="1" column="4"></Square>

<Square row="1" column="5"></Square>

</div>

A su vez, el componente cuadrado se modificara para recibir las propiedades

Square.js

function Square({row, column})

Además, debemos indicar en que intento estamos actualmente, para habilitar solamente aquellos cuadrados que estén en la fila que corresponde al intento. Para ello vamos a crear una variable en la clase principal que será accesible para los hijos. Como el estado del cuadrado dependerá de esta variable, la definiremos como un estado.

App.js

const [currentAttempt, setCurrentAttempt] = useState(1);

…..

<Square row="1" column="1" attempt={currentAttempt}></Square>

….

Square.js

function Square({row, column, attempt}) {

const [letter, setLetter] = useState(' ')

return (

<input className="cuadrado" disabled={row != attempt ? "true" : ""} type="text" value={letter} onChange={(e)=> setLetter(e.target.value)}></input>

);

}

Ahora vamos a añadir en la clase padre la solución y guardar las letras introducidas en cada intento.

Para ello vamos a crear una variable solución y una lista de 5 cadenas con 5 caracteres vacíos para los intentos.

App.js

const solution ='CLASE';

function App() {

const [currentAttempt, setCurrentAttempt] = useState(1);

const [messageWinner, setMessageWinner] = useState("");

const guesses = [" ", " "," " , " ", " "];

La función que guardará los caracteres en la cadena correspondiente al número de intento activo en este momento se llamará **handleSquareChange**. Además, en dicha función, si el carácter introducido completa las 5 letras del intento se comprobará si coincide con la solución y en caso contrario avanzamos al siguiente intento siempre que no hayamos superado los 5 intentos.

App.js

const handleSquareChange=(parameters) =>

{

guesses[parameters.row-1]= replaceAt(guesses[parameters.row-1], parameters.column-1,parameters.letter);

let isAttempComplete=true;

for(let i=0;i<5;i++)

{

if(guesses[parameters.row-1].charAt(i)==" ")

{

isAttempComplete=false;

break;

}

}

if (isAttempComplete)

{

if(guesses[parameters.row-1] == solution)

{

setMessageWinner("CONGRATULATIONS");

setCurrentAttempt(10);

}

else

{

if (currentAttempt<5)

{

setCurrentAttempt(currentAttempt+1);

}

else

{

setMessageWinner("Game Over");

setCurrentAttempt(10);

}

}

}

Ahora vamos a pasar la referencia a los cuadrados para que llamen a esta función cuando se introduzca el carácter.

App.js

<div className='board'>

<div id="row" className='row' >

<Square row="1" column="1" attempt={currentAttempt} textUpdate={(e) =>handleSquareChange(e)}></Square>

<Square row="1" column="2" attempt={currentAttempt} textUpdate={(e) =>handleSquareChange(e)}></Square>

<Square row="1" column="3" attempt={currentAttempt} textUpdate={(e) =>handleSquareChange(e)}></Square>

<Square row="1" column="4" attempt={currentAttempt} textUpdate={(e) =>handleSquareChange(e)}></Square>

<Square row="1" column="5" attempt={currentAttempt} textUpdate={(e) =>handleSquareChange(e)}></Square>

</div>

…

Cambiaremos el Hook Square para recibir este parámetro, y llamarlo cuando el usuario introduzca el valor

Square.js

function Square({row, column, attempt, textUpdate}) {

const [letter, setLetter] = useState(' ');

const handleInputLetter=(e) =>

{

let letter=e.target.value.trim()[0].toUpperCase();

setLetter(letter);

textUpdate({row, column, letter});

}

return (

<input className="cuadrado" disabled={row != attempt ? true : false} type="text" value={letter} onChange={handleInputLetter}></input>

);

}

Comprobaremos que ahora mismo la lógica funciona correctamente , pero sin pintar las letras en verde o amarillo.

### Coloreando las letras.

Ahora nos toca dar las pistas de las letras acertadas en cada intento.

Para ello tenemos que tener claro que solo pintamos los cuadrados con letras de un intento anterior al actual.

Si la letra coincide y además está en la posición adecuada , la pintaremos en verde y si no en amarillo.

Para ello vamos a añadir dos estilos nuevos en nuestro css

Square.css

.cuadradoverde

{

color:#ffff;

width: 55px;

height: 55px;

background-color: #29ca3f;

display: inline-block;

border:1px solid white;

text-align: center;

font-size: 2.5em;

margin: 4px;

}

.cuadradoamarillo

{

color:#ffff;

width: 55px;

height: 55px;

background-color: rgb(209, 209, 90);

display: inline-block;

border:1px solid white;

text-align: center;

font-size: 2.5em;

margin: 4px;

}

El proceso solo tiene que aplicarse cuando se haya completado un intento, es decir, cuando se actualice el estado de la variable attempts.

Una vez que se actualice el número de intentos , queremos forzar que se vuelva a pintar los colores de los cuadrados teniendo en cuenta la lógica aplicada, para ello usaremos el Hook useEffect() asociado al estado cuyo cambio queremos ser notificados.

Square.js

function Square({ row, column, attempt, textUpdate}) {

const [letter, setLetter] = useState(' ');

const [backgroundCellColor, setBackgroundCellColor] = useState('cuadrado');

const handleInputLetter=(e) =>

{

let letter=e.target.value.trim()[0].toUpperCase();

setLetter(letter);

textUpdate({row, column, letter});

}

useEffect(() => {

// Update the document title using the browser API

if (row < attempt && solution.charAt(column-1)==letter)

{ setBackgroundCellColor('cuadradoverde');}

else {

if(row<attempt && solution.indexOf(letter)>=0)

{

setBackgroundCellColor('cuadradoamarillo');

}

}

}, [attempt]);

return (

<input className={backgroundCellColor} disabled={row != attempt ? true : false} type="text" value={letter} onChange={handleInputLetter} ></input>

);

}

Ya tenemos nuestra primera versión de juego, ahora toca mejorarlo.