



# Proyecto Relojes ReactJS

## Introducción

A lo largo de este proyecto vamos a crear una app que incluya un `reloj digital`, una `cuenta atrás` y un `cronómetro` a través de 3 componentes distintos haciendo uso de distintos `Hooks`.

Antes de nada, vamos a crear nuestro nuevo proyecto de React:

```
npm create vite@lates
```

Una vez arranquemos nuestro proyecto, limpiemos el esqueleto inicial y tengamos nuestra clásica estructura creada, vamos a abordar nuestro primer componente: el `reloj digital`.


## Reloj digital

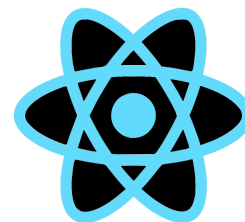
Vamos a crear nuestro componente `<DigitalClock/>` dentro de la carpeta `components` exportándolo al `index.js` y exportándolo para recogerlo en `App.jsx`.

Truco: si tenéis instalada la extensión `ES7 React/Redux/GraphQL/React-Native Snippets`, una forma muy rápida de crear componentes es introduciendo "rafce" en nuestro archivo vacío.

ES7+ React/Redux/React-Native snippets - Visual Studio Marketplace

Extension for Visual Studio Code - Extensions for React, React-Native and Redux in JS/TS with ES7+ syntax. Customizable. Built-in integration with prettier.

 <https://marketplace.visualstudio.com/items?itemName=dsznajder.es7-react-js-snippets>



Esto nos creará una `ReactArrowFunctionExportComponent`, o lo que es lo mismo, la estructura principal de un componente:

```
import React from 'react'

const DigitalClock = () => {
  return (
    <div>

    </div>
  )
}

export default DigitalClock
```

Por un lado tenemos que definir `useState` para que nos setee la hora a través de una función, y por otro definir un `useEffect` que nos recoja la `hora local` a través de una función y la ejecute tanto al cargar la página como al pasar a través del `intervalo` de un segundo.

```
import React, { useEffect, useState } from "react";

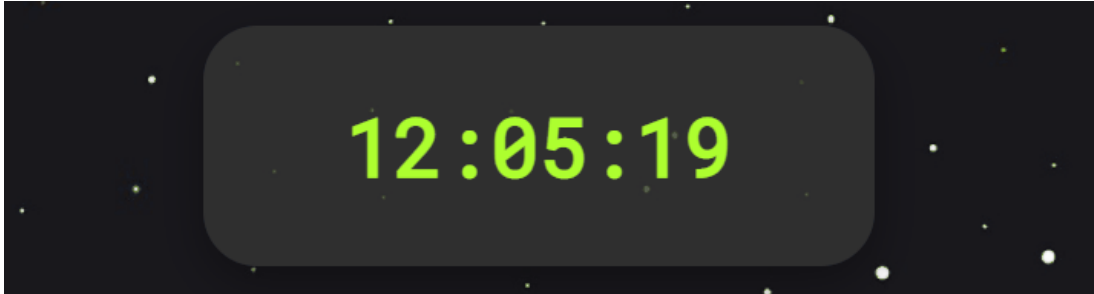
const DigitalClock = () => {
  const [clockState, setClockState] = useState();

  useEffect(() => {
    setInterval(() => {
      const date = new Date();
      setClockState(date.toLocaleTimeString());
    }, 1000);
  }, []);
  return (
    <div className="digital-clock">
      <h2>{clockState}</h2>
    </div>
  );
};

export default DigitalClock;
```

La constante `date` recoge la información de la hora local a través del método `date.toLocaleTimeString()` y el valor recogido será el que se aplique a `clockState` mediante `setClockState` cada `1000` milisegundos (un segundo).

De esta manera, con unos `estilos` de css muy básicos y retornando el valor de `clockState` tendremos nuestra hora reflejada segundo a segundo, tal y como funciona un reloj digital:



## Cuenta atrás

Le toca el turno al componente de la `cuenta atrás`, un componente al que le indicaremos una fecha y una hora determinada e irá calculando el tiempo restante hasta llegar a 0.

```
import React, { useState, useEffect } from "react";

const Countdown = () => {
  const [time, setTime] = useState("");
  useEffect(() => {
    let countDownDate = new Date("Jan 1, 2020 00:00:00").getTime();
  }, []);
  return <div className="countdown"></div>;
};

export default Countdown;
```

En nuestro componente `<Countdown/>` vamos a usar `useState` para definir el tiempo dentro de la constante `time` y `useEffect` para que la variable `countDownDate` nos recoja el `valor` del tiempo en milisegundos de un string como `"Jan 1, 2020 00:00:00"`.

Una vez definida nuestra fecha "meta" necesitamos incluir un intervalo para que la función `getTime()` se ejecute segundo a segundo:

```
import React, { useState, useEffect } from "react";
```

```
const Countdown = () => {
  const [time, setTime] = useState("");
  useEffect(() => {
    let countDownDate = new Date("Jan 1, 2020 00:00:00").getTime();
    let x = setInterval(() => {
      let now = new Date().getTime();
    }, 1000);
  }, []);
  return <div className="countdown"></div>;
};

export default Countdown;
```

De esta forma tendremos actualizada la información cada 1000 milisegundos. El "problema" al que nos enfrentamos ahora es que estamos almacenando la fecha y hora actual dentro de la variable "now", por lo que siguiendo la lógica debemos calcular la distancia entre `now` y `countDownDate`.

Esta diferencia será la que nos refleje segundo a segundo la cuenta atrás, así que vamos a ello:

```
const Countdown = () => {
  const [time, setTime] = useState("");
  useEffect(() => {
    let countDownDate = new Date("Jan 1, 2020 00:00:00").getTime();
    let x = setInterval(() => {
      let now = new Date().getTime();
      let distance = countDownDate - now;
      let days = Math.floor(distance / (1000 * 60 * 60 * 24));
      let hours = Math.floor((distance % (1000 * 60 * 60 * 24)) / (1000 * 60 * 60));
      let minutes = Math.floor((distance % (1000 * 60 * 60)) / (1000 * 60));
      let seconds = Math.floor((distance % (1000 * 60)) / 1000);

      setTime(days + "d " + hours + "h " + minutes + "m " + seconds + "s ");

    }, 1000);
  }, []);
};
```

Hemos creado la variable `distance` a la cual le vamos a dar el valor de `countDownDate` menos el valor de `now`. Esto nos devolverá el tiempo que falta cada segundo para la cuenta atrás.

Al haber obtenido el valor del tiempo en milisegundos tenemos que desglosar dicho tiempo en variables `days`, `hours`, `minutes` y `seconds` realizando en cada uno de ellos el calculo pertinente.

Por ejemplo, en `minutes` estamos indicando que un minuto son `60` segundos convirtiéndolos de milisegundos a segundos.

Una vez obtenido nuestras variables desglosadas vamos a concatenarlas todas para setear el tiempo con `setTime` de la siguiente manera:

```
setTime(days + "d " + hours + "h " + minutes + "m " + seconds + "s ");
```

Solo nos faltaría un aviso o algún tipo de mensaje que nos indique que la cuenta atrás ha terminado y nos detenga el intervalo de 1000 milisegundos, por lo que vamos a realizar esta función con un simple condicional:

```
import React, { useState, useEffect } from "react";
import "./Countdown.css";

const Countdown = () => {
  const [time, setTime] = useState("");
  useEffect(() => {
    let countDownDate = new Date("August 28, 2021 13:21:00").getTime();
    let x = setInterval(() => {
      let now = new Date().getTime();

      let distance = countDownDate - now;

      let days = Math.floor(distance / (1000 * 60 * 60 * 24));
      let hours = Math.floor(
        (distance % (1000 * 60 * 60 * 24)) / (1000 * 60 * 60)
      );
      let minutes = Math.floor((distance % (1000 * 60 * 60)) / (1000 * 60));
      let seconds = Math.floor((distance % (1000 * 60)) / 1000);

      setTime(days + "d " + hours + "h " + minutes + "m " + seconds + "s ");

      if (distance < 0) {
        clearInterval(x);
        setTime("COUNTDOWN FINISHED");
      }
    }, 1000);
  }, []);
}
```

```

    return (
      <div className="countdown">
        <h2>{time}</h2>
      </div>
    );
  };

  export default Countdown;

```

De esta manera el valor de time será "COUNTDOWN FINISHED", indicando así el final de la función.

La medida que siempre utilizamos es segundos, por eso los `math.floor` se utilizan para calcularlo a través de la medida (1000) que representa 1 segundo.

Solo nos quedaría llamar al valor de `time` en nuestro `return` y tendremos la cuenta atrás funcionando:



## Cronómetro

Este ejemplo será el más complejo de los tres, ya que vamos a crear un componente capaz de inicializar un cronómetro, pausarlo, reanudarlo y reiniciarlo. Vamos allá.

Para controlar el estado de nuestro cronómetro vamos a utilizar useState de la siguiente manera:

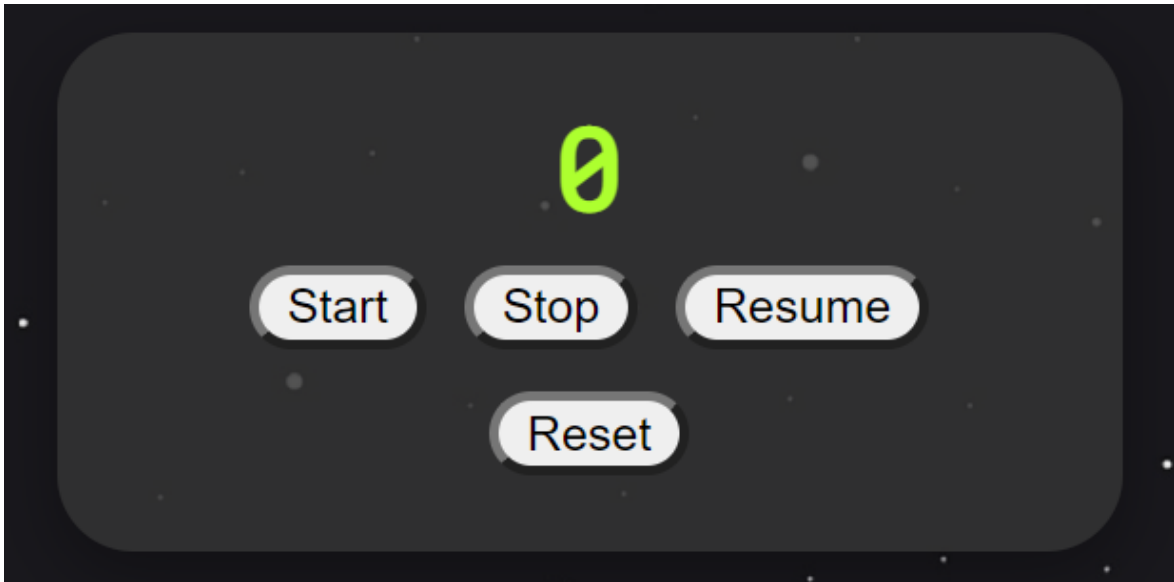
```
const Stopwatch = () => {
  const [time, setTime] = useState(0);
  const [timerOn, setTimerOn] = useState(false);
  return (
    <div className="stopwatch">

    </div>
  )
}
```

Tenemos por un lado una constante llamada `time`, la cual tendrá de valor por defecto `0` y por otro una constante llamada `timerOn` la cual tendrá de valor por defecto `false`, ya que el cronómetro no estará activado al arrancar la aplicación.

Para controlar el estado `timerOn` vamos a crear los botones del cronómetro de la siguiente manera:

```
const Stopwatch = () => {
  const [time, setTime] = useState(0);
  const [timerOn, setTimerOn] = useState(false);
  return (
    <div className="stopwatch">
      <h2>{time}</h2>
      <button onClick={() => setTimerOn(true)}>Start</button>
      <button onClick={() => setTimerOn(false)}>Stop</button>
      <button onClick={() => setTimerOn(true)}>Resume</button>
      <button onClick={() => setTime(0)}>Reset</button>
    </div>
  );
};
```



Vayamos por partes:

- El botón `Start` va a hacer que `setTimerOn` se vuelva true, por lo que arrancará a contar una vez definamos el comportamiento que le dará valor a time.
- El botón `Stop` devolverá a `setTimerOn` su estado por defecto, false.
- El botón `Resume` es idéntico a Start, más tarde definiremos la diferencia entre uno y otro.
- El botón `Reset` seteará la constante `time` a `0` de nuevo, reiniciándolo a su valor por defecto.

Vamos a definir ahora un `useEffect` que corra cada vez que el valor de `timerOn` cambie, ya que por defecto correría cada vez que se renderice la aplicación. En esta ocasión solo queremos que tenga efecto una vez activemos el cronómetro.

```
const Stopwatch = () => {
  const [time, setTime] = useState(0);
  const [timerOn, setTimerOn] = useState(false);

  useEffect(() => {
    let interval = null;

    if (timerOn) {
      interval = setInterval(() => {
        setTime((prevTime) => prevTime + 10);
      }, 10);
    } else {
```



```

    clearInterval(interval);
  }

  return () => clearInterval(interval);
}, [timerOn]);
return (
  <div className="stopwatch">
    <h2>{time}</h2>
    <button onClick={() => setTimerOn(true)}>Start</button>
    <button onClick={() => setTimerOn(false)}>Stop</button>
    <button onClick={() => setTimerOn(true)}>Resume</button>
    <button onClick={() => setTime(0)}>Reset</button>
  </div>
);
};

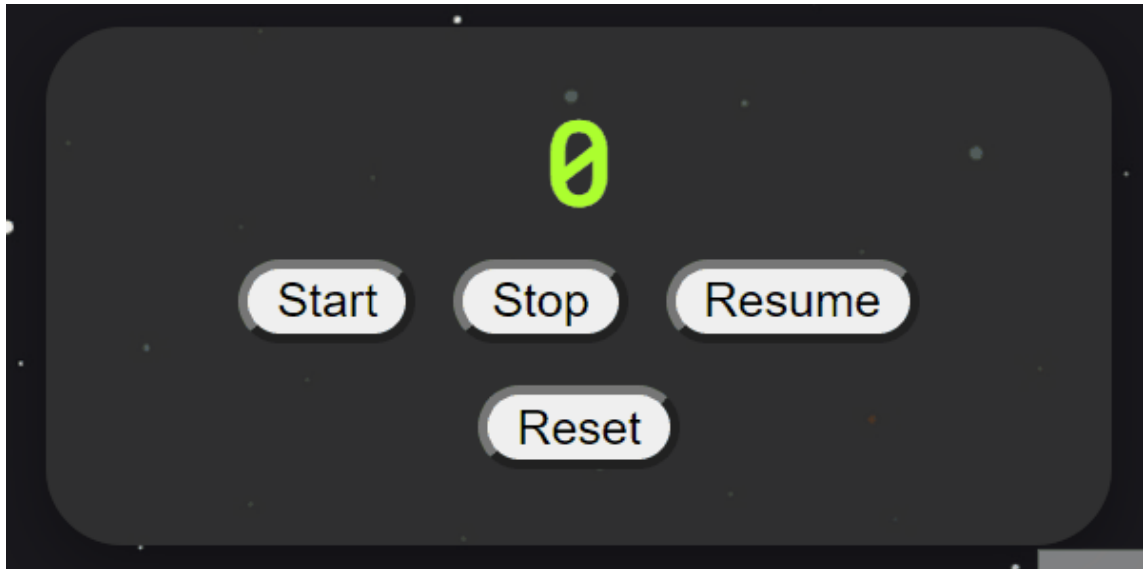
```

Hemos configurado nuestro `useEffect` con una variable `interval` inicializada en `null`. Gracias a un condicional vamos a hacer que, si el `timerOn` está corriendo (`true`), nos almacene el valor del tiempo cada `10 milisegundos`. Con el método de JavaScript `setInterval()` nos va a ejecutar este proceso cada 10 milisegundos y va a almacenar el tiempo obtenido en la variable `time` + 10 milisegundos de diferencia.

Esta operación nos permitirá igualar el resultado obtenido con el "delay" del intervalo.

Por otra parte, en el condicional tenemos el `else` para que si `timerOn` no está corriendo nos iguale de nuevo `interval` a `null`. También es conveniente despejar esta operación a través del método `clearInterval()` dentro del `return`, así nos aseguraremos de que esta operación se detiene por completo una vez dejemos de utilizar la aplicación.

Una vez realizados estos pasos podremos ver que nuestro cronómetro arranca, se detiene y se reinicia:



Nos faltaría formatear el tiempo en minutos, segundos y centésimas (de manera similar a como lo hicimos en la cuenta atrás), y ocultar o mostrar algunos de los botones según la acción que estemos realizando.

De momento estábamos retornando el valor de `{time}` sin más, por lo que vamos a dividirlo en las 3 secciones que suele mostrar un cronómetro estándar:

```
<span>{"0" + Math.floor((time / 60000) % 60).slice(-2)}</span>  
<span>{"0" + Math.floor((time / 1000) % 60).slice(-2)}</span>  
<span>{"0" + ((time / 10) % 100).slice(-2)}</span>
```

Dentro de estos 3 `<span>` hemos incluido un `"0"` al principio para mostrar correctamente los valores iniciales, calculado tanto los minutos, como los segundos y las centésimas.

Por ejemplo, para calcular los `minutos` hemos dividido el tiempo entre `60000`, ya que hay 60 segundos en un minuto, por lo tanto 60000 milisegundos.

Usando el módulo sobrante `%` retornaremos el restante después de dividir el número en proporciones enteras de, en nuestro caso, 60. Gracias a esto podemos encapsular nuestro

valor en 60, los segundos de un minuto.

También hemos hecho uso de `.slice` para, una vez tengamos el valor máximo de las dos cifras, oculta el "exceso" creando el efecto de pasar los valores de un span a otro de derecha a izquierda.

De esta forma los marcadores siempre tendrán dos dígitos pese a que el valor acumulado en cada ellos vaya aumentando. Con esto y utilizando `":"` como separador ya tendremos nuestro display funcionando perfectamente:



Lo único que nos queda es ocultar y mostrar los botones de nuestro cronómetro según necesitemos. Para ello vamos a utilizar condicionales:

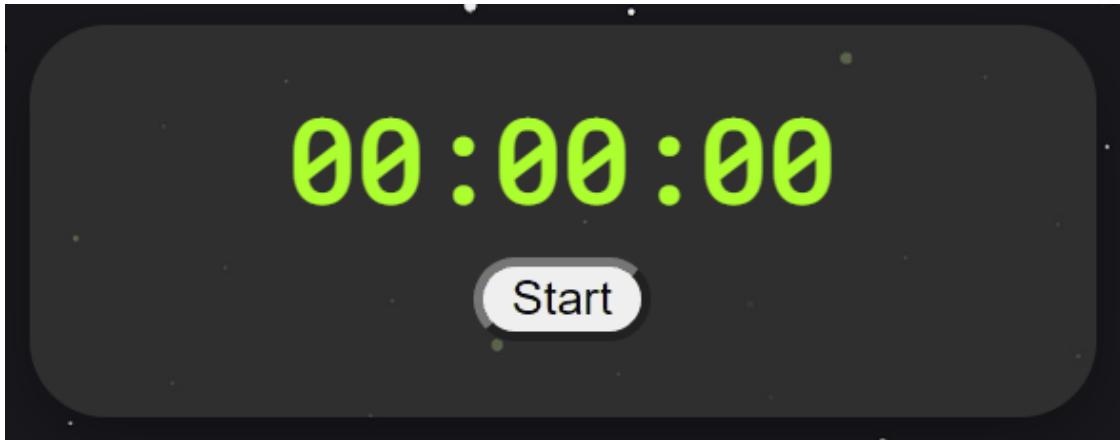
```
<div id="buttons">
  {!timerOn && time === 0 && (
    <button onClick={() => setTimerOn(true)}>Start</button>
  )}
  {timerOn && <button onClick={() => setTimerOn(false)}>Stop</button>}
  {!timerOn && time > 0 && (
    <button onClick={() => setTime(0)}>Reset</button>
  )}
  {timerOn && time > 0 && (
    <button onClick={() => setTimerOn(true)}>Resume</button>
  )}
</div>
```

De esta manera vamos a mostrar el botón `Start` si el `timerOn` es false y `time === 0`, es decir, cuando no haya arrancado el cronómetro y el valor de time esté todavía en 0.

El botón `Stop` aparecerá una vez `timerOn` sea true y el valor de `time` sea mayor que 0.

`Resume` y `Reset` solo aparecerán cuando el `timerOn` esté en `false` y `time` sea mayor que cero, es decir, aparecerán una vez haya arrancado por primera vez el cronómetro., lo hayamos parado mediante `Stop` y `time` haya acumulado algún valor por encima de 0.

Y con este condicional tendremos nuestro cronómetro completamente funcional y listo para batir las mejores marcas:



Si invocamos nuestros 3 componentes en `App.jsx` tendremos una aplicación completa y preparada para controlar el tiempo de 3 formas distintas.

