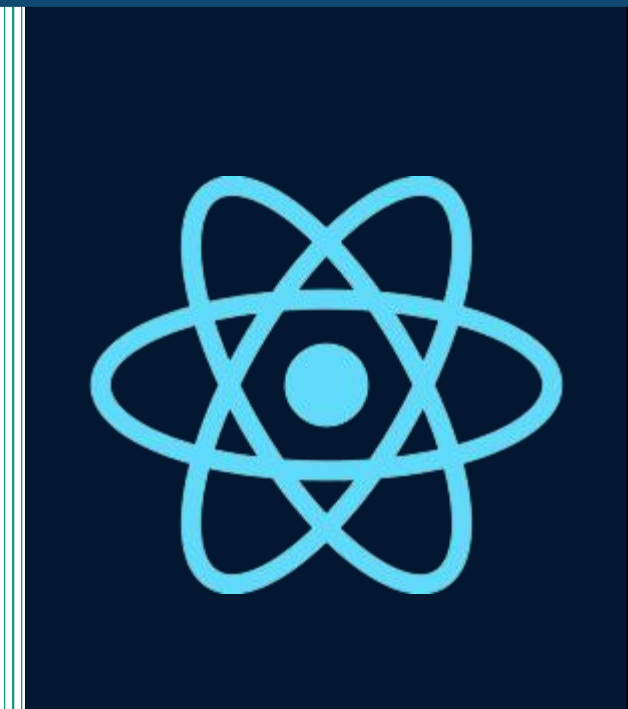
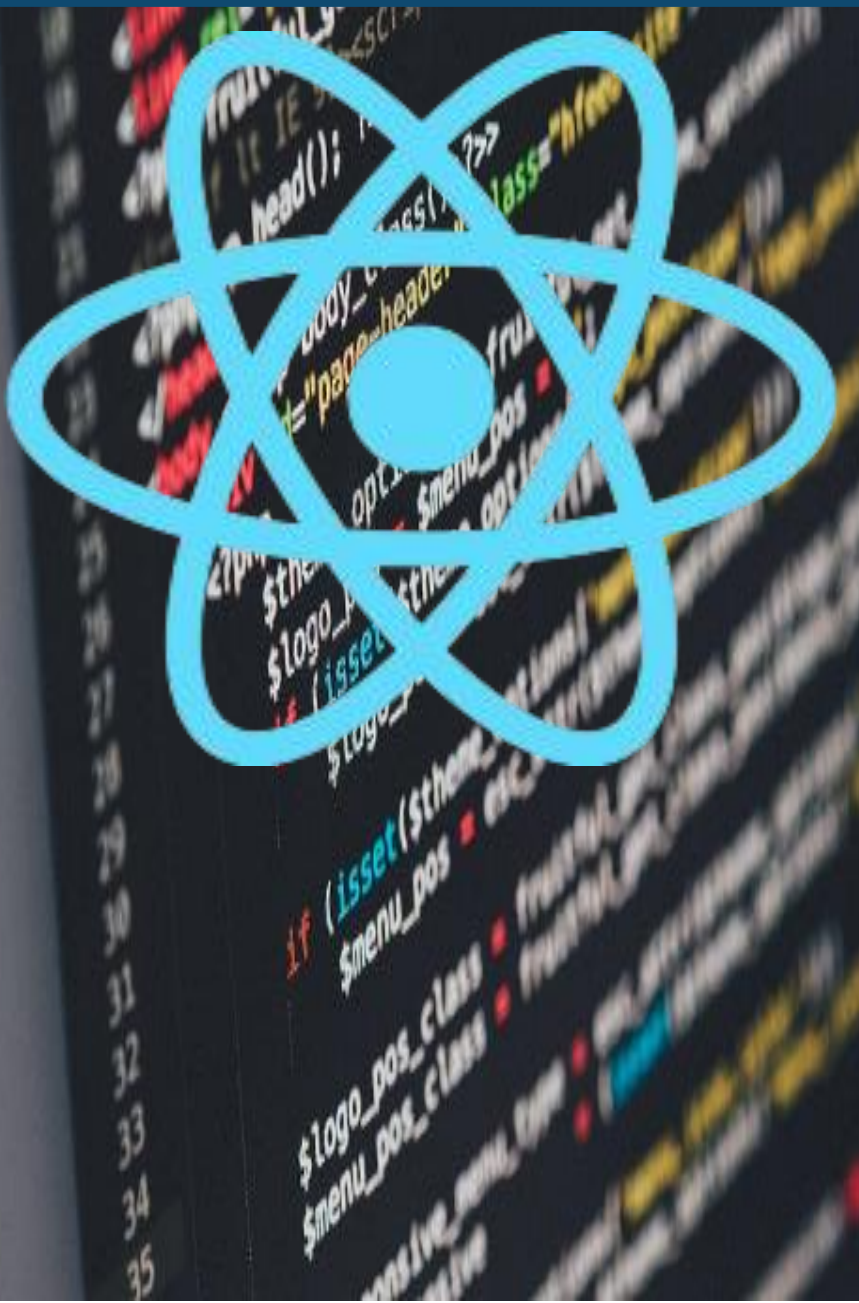


# DESARROLLO WEB ENTORNO CLIENTE

# 2° DAW CFGM

# REACT: TIENDA DE GUITARRAS



**Natalia Escrivá Núñez**

**IES Serra Perenxisa**

**n.escribanunez@edu.gva.es**

Curso 2024-2025

# CONTENIDO

<b>Contenido</b> .....	<b>2</b>
<b>1. INTRODUCCIÓN</b> .....	<b>4</b>
<b>2. TECNOLOGIAS/HERRAMIENTAS NECESARIAS</b> .....	<b>4</b>
<b>3. PRIMEROS PASOS</b> .....	<b>5</b>
3.1. Instalación herramientas .....	5
3.2. Creando el primer proyecto .....	7
3.3. Estructura de un proyecto con Vite .....	9
3.4. Primeros cambios en el proyecto .....	11
<b>4. ESCRIBIR CSS EN REACT</b> .....	<b>15</b>
<b>5. COMPONENTES EN REACT</b> .....	<b>16</b>
<b>6. JSX</b> .....	<b>19</b>
6.1. Reglas en jsx.....	19
6.2. De class a className .....	22
<b>7. PRACTICANDO CON LOS COMPONENTES</b> .....	<b>25</b>
7.1. Crear un componente .....	25
<b>8. REACT HOOKS</b> .....	<b>28</b>
8.1. State en React y Hook useState().....	29
8.2. Hook useEffect().....	31
8.3. Practicando con Hooks .....	33
<b>9. STATEMENTS AND EXPRESSIONS</b> .....	<b>37</b>
<b>10. PROPS</b> .....	<b>39</b>
<b>11. EVENTOS</b> .....	<b>44</b>
<b>12. FORMAS DE MODIFICAR UN STATE</b> .....	<b>48</b>
12.1. Pasar el State a través de Props .....	48
12.2. Usando un PrevState .....	49
<b>13. INMUTABILIDAD</b> .....	<b>52</b>
13.1. Practicamos inmutabilidad y eventos .....	52

<b>14. STATE DERIVADO.....</b>	<b>57</b>
<b>15. HOOK useMEMO .....</b>	<b>59</b>
<b>16. PRACTICANDO LO APRENDIDO .....</b>	<b>60</b>
<b>17. localStorage .....</b>	<b>62</b>
17.1. Hook useEffect .....	62
17.2. Recuperando datos de localStorage .....	63

# 1. INTRODUCCIÓN

**React es una biblioteca de JS para construir Interfaces de Usuario (UI).**

React ayuda a crear interfaces de usuario interactivas de forma sencilla.

**Se sirve de funciones y eventos** que dispararán ciertas funciones en base a las interacciones del usuario.

Se encarga de **Actualizar y Renderizar** (mostrar) de manera eficiente y automática **los componentes de la aplicación**. Por ejemplo, tras pulsar en algún lugar, mostramos una nueva sección.

**Se basa en Componentes** que no son más que funciones en JS que son utilizadas para renderizar el diseño de la aplicación o sitio web.

Está **desarrollada por Meta** (antes Facebook). Esto nos indica que hay un soporte importante detrás de React. Además, otra empresa llamada Vercel también apoya el desarrollo de esta biblioteca.

## 2. TECNOLOGIAS/HERRAMIENTAS NECESARIAS

Existen muchas formas de crear una app en React, incluso se puede crear un ambiente propio de desarrollo con herramientas como Babel, Parcel, Webpack, Vite, etc..

También existen una gran cantidad de herramientas disponibles para crear aplicaciones en React en minutos sin conocer la configuración. Es la opción recomendada ya que son estables y ahorran la preocupación de la configuración.

Necesitaremos tener instalado Node.js (npm) o yarn

Como opciones actuales encontramos Vite y Next.js, también Remix Run y Astro. En estos momentos Remix Run ha sido comprado por Shopify y están haciendo muchos cambios. También se utiliza Git.

Si se empieza un proyecto con Vite se tendrá que instalar dependencias para Routing, consultas HTTP.

Necesitaremos también la extensión del navegador React Developer Tools (disponible para Chrome y Firefox)

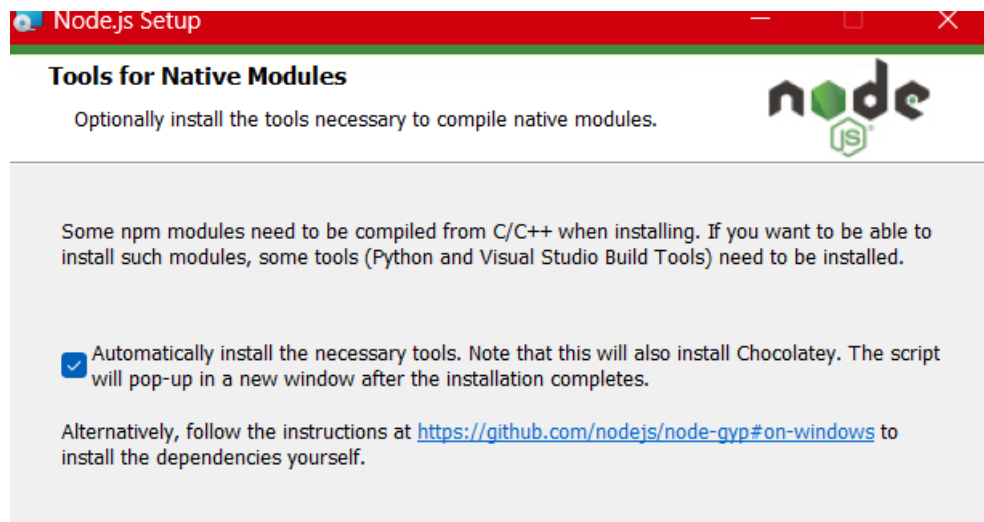
## 3. PRIMEROS PASOS

### 3.1. Instalación herramientas

Instalación de node.js → Desde la [página oficial](#), instalaremos la versión estable más reciente. Usaremos node.js para tener el entorno y generar proyectos y npm para instalar dependencias.



Para evitar instalar dependencias más tarde, clicamos la opción de instalarlas en este momento: Chocolatey, Python..



Si ya tenemos instalado node.js instalado, al instalar la nueva versión, directamente se actualiza.

```
PS C:\Users\Natalia> node -v
v20.10.0
PS C:\Users\Natalia> npm -v
9.6.2
```

```
PS C:\Users\Natalia> node -v
v20.17.0
PS C:\Users\Natalia> npm -v
9.6.2
```

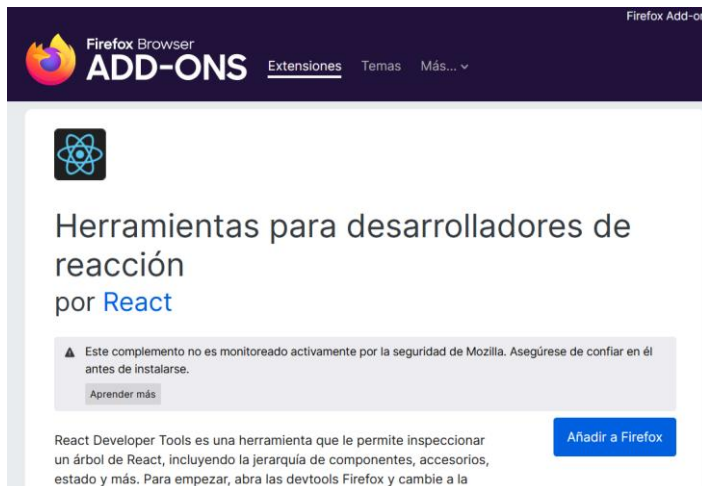
Podemos ver que node.js se ha actualizado pero npm no. Lo hacemos a mano:

```
PS C:\Users\Natalia> npm install npm -g
```

```
PS C:\Users\Natalia> node -v
v20.17.0
PS C:\Users\Natalia> npm -v
10.8.2
```

Instalamos ahora la extensión React Developer Tools en [Chrome](#) y [Firefox](#).





## 3.2. Creando el primer proyecto

Lo primero que haremos es situarnos en la ruta donde vamos a crear el proyecto. Utilizamos **Vite como herramienta de compilación**.

```
npm create vite@latest
```

Por ser la primera vez nos pide autorización para instalar una dependencia → autorizamos.

```
Need to install the following packages:
create-vite@5.5.2
Ok to proceed? (y)
```

Nos pide el nombre del proyecto → TiendaGuitarra. Nos sugiere con el mismo nombre el Package Name → aceptamos

```
✓ Project name: ... tiendaGuitarra
✓ Package name: ... tiendaguitarra
```

Seleccionamos el framework → React

```
? Select a framework: » - Use arrow-keys. Return to submit.  
  Vanilla  
  Vue  
>  React  
    Preact  
    Lit  
    Svelte  
    Solid  
    Qwik  
    Others
```

Debemos seleccionar ahora la versión a instalar → Elegimos JavaScript + SWC (nuevo compilador de Vite, muy rápido y estable).

```
? Select a variant: » - Use arrow-keys. Return to submit.  
  TypeScript  
  TypeScript + SWC  
  JavaScript  
>  JavaScript + SWC  
  Remix
```

Vemos que el proyecto se ha creado → Comprobamos que tenemos el proyecto en la ruta indicada.

```
Scaffolding project in E:\CLASE\DAW\DWEC\24-25\BLOQUE3_FRAMEWORKS\REACT\PROYECTOS\1. FUNDAMENTOS Y TYPESCRIPT\tiendaGuitarra...
```

Nos posicionamos en la ruta del proyecto y ejecutamos los comandos que se indican.

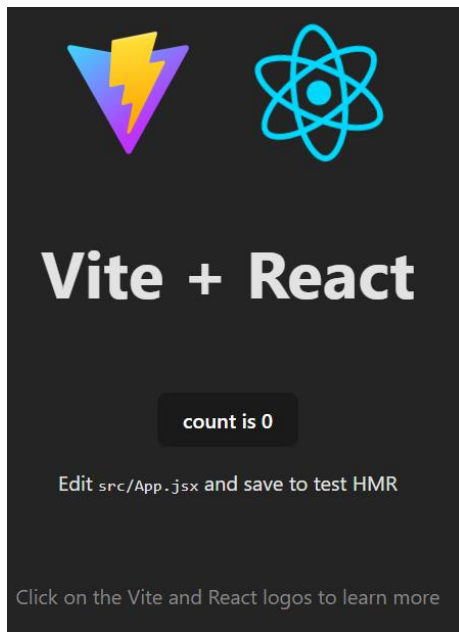
```
Done. Now run:  
  
cd tiendaGuitarra  
npm install  
npm run dev
```

Con el último comando lo que hacemos es arrancar el servidor de desarrollo, nos da esta información:

```
VITE v5.4.2 ready in 265 ms  
  
→ Local:   http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help
```



Tenemos la url que nos da. La copiamos y ponemos en un navegador. Tenemos la página principal del proyecto.

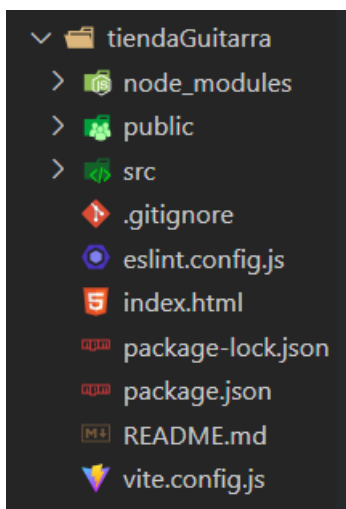


Si nos fijamos vemos un contador que, si pulsamos se va incrementando el número.

Podemos cerrarlo y abrir el proyecto en el editor Visual Studio

### 3.3. Estructura de un proyecto con Vite

Abrimos el proyecto en Visual Studio code y vemos la estructura del proyecto:



**node\_modules:** Aloja las dependencias del proyecto. Según se vayan instalando, la carpeta irá creciendo.

**public:** se suelen colocar las imágenes que serán accesibles sin más para los usuarios del sitio web.

**src:** Aloja todo el código. Donde pasaremos la mayor parte del tiempo. Dentro encontramos por ejemplo `eslint.config.js`: Nos permite definir ciertas reglas sobre cómo escribir el código de JS y encontrar posibles errores. NO lo modificaremos.

**.gitignore:** Ignora algunos archivos

**Index.html:** Inyecta React en el div del body. Es el archivo principal del proyecto. React es una biblioteca de JS. Todo el código que se genera va a ese

div. Es un DOM virtual, es decir, es código HTML que se está representando. En este archivo NO se hacen cambios, solo el título o incluir herramientas de Google.

**package.json**: archivo de configuración con información para arrancar el proyecto, construirlo, presvisualizarlo. Además, se instalan aquí las dependencias, del proyecto y del desarrollo. Cuando en la consola ponemos "**npm run dev**", va a la línea 7 de este fichero.

```
6   "scripts": {  
7     "dev": "vite",  
8     "build": "vite build",  
9     "lint": "eslint .",  
10    "preview": "vite preview"
```

**package-lock.json**: NO se modifica. Se genera directamente a partir de la información del fichero anterior.

**vite.config.js**: archivo de configuración de vite. No escribiremos código en ese fichero.

Profundizamos en **src** y **public**:

En **src** es donde haremos nuestra mayor parte del trabajo. Iremos creando distintos archivos y los componentes.

En **public** encontramos el logo de vite.svg y en **src** → **assets** encontramos el logo de react.svg

En **src** → **App.jsx** es donde está incluido el código que vemos en la pantalla principal, incluidos los logos. Cuando vemos la extensión **jsx** es porque incluye una sintaxis que nos permite combinar código HTML con código de JS, muy propia de **React**. Algo importante es ver **la forma en la que se importa**:

Si está en **public**, directamente **import nombreVariable from '/vite.svg'**

```
import viteLogo from '/vite.svg'
```

Si está en **src** será **import nombreVariable from './assets/react.svg'**

```
import reactLogo from './assets/react.svg'
```

**App.css**: archivo que se crea directamente con código css cuando se crea App.jsx (contiene su hoja de estilos).

**Index.html**: encontramos el código que inicia la aplicación. En **src** → **main.jsx** encontramos el código que selecciona el id **root** y lo renderiza. De ahí pasará al componente **App.jsx**.

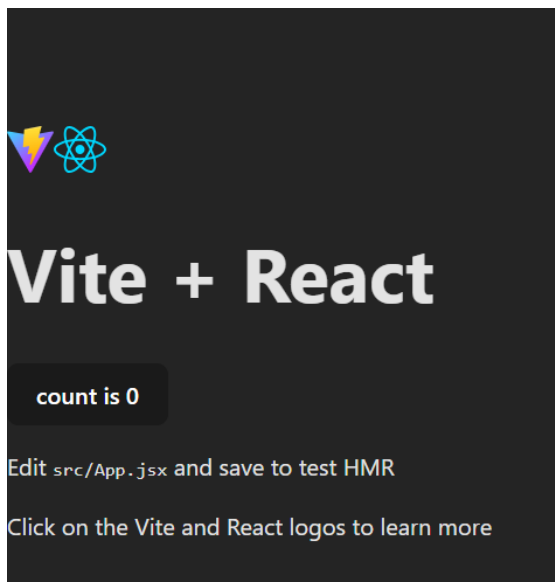
### 3.4. Primeros cambios en el proyecto

1. Eliminamos el fichero **App.css** ya que los estilos irán en cada componente que vayamos creando.

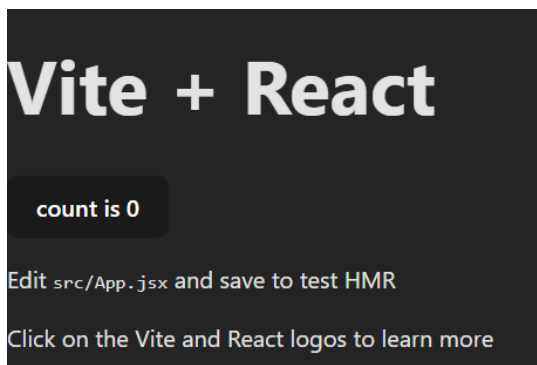
Tendremos que quitar también su vinculación en el fichero **App.jsx**

```
tiendaGuitarra > src > App.jsx > ...  
1  import { useState } from 'react'  
2  import reactLogo from './assets/react.svg'  
3  import viteLogo from '/vite.svg'  
4  import './App.css'
```

Ahora, el proyecto se ve así:



2. Eliminamos los logos. El div de App.jsx que los renderiza y también sus import. Por último, podemos eliminar los archivos svg. El sitio web se vería así:



3. Eliminamos todo el código de **App.jsx** (a partir de return), a excepción de un **h1** con el texto **"Tienda de Guitarras"**.

```
1  import { useState } from 'react'
2
3  function App() {
4    const [count, setCount] = useState(0)
5
6    return (
7      <>
8        <h1>Tienda de Guitarras</h1>
9      </>
10    )
11  }
12
13  export default App
```

Tienda de Guitarras

Vemos que el texto que hemos introducido queda a mitad de la página. Esto es porque en el fichero **main.jsx**, una vez que selecciona el elemento **'root'** de **index.html**, monta toda la aplicación de React en ese **div**.

```
tiendaGuitarra > index.html > ...
1  <!doctype html>
2  <html lang="en">
3  >   <head> ...
8     </head>
9     <body>
10      <div id="root"></div>
11      <script type="module" src="/src/main.jsx"></script>
12    </body>
13  </html>
14
```

```
App.jsx  main.jsx  X
tiendaGuitarra > src > main.jsx
1  import { StrictMode } from 'react'
2  import { createRoot } from 'react-dom/client'
3  import App from './App.jsx'
4  import './index.css'
5
6  createRoot(document.getElementById('root')).render(
7    <StrictMode>
8      <App />
9    </StrictMode>,
10 )
```

La app que monta es la que vemos en **<App />** y es el fichero **App.jsx** mostrando el contenido de ese componente. Esto será el punto inicial de la aplicación.

Seguimos en main.jsx y vemos dos imports:

- **from 'react'** → React es la biblioteca ligera.
- **from 'react-dom/client'** → React-Dom es lo que nos permite integrarlo con nuestro código HTML.

Si vemos **from 'react-Native'** estaríamos creando aplicaciones para Android e iOS.

Vemos otro **import from './index.css'** → Es la hoja de estilos que se utiliza para incluir estilos globales, comunes a todos los componentes. Vamos a eliminar el contenido de ese fichero. Ahora nuestra app queda así, sin estilos.

## Tienda de Guitarras

4. Seguimos en el fichero **App.jsx** y eliminamos el import y la variable de la función. El fichero queda así:

```
tiendaGuitarra > src > App.jsx > ...
1
2  function App() {
3      return (
4          <>
5
6              <h1>Tienda de Guitarras</h1>
7
8          </>
9      )
10 }
11
12 export default App
13
```

En **resumen**, ficheros y su contenido:

- index.html → fichero sobre el que se monta la aplicación de React.
- src → escribiremos la mayor parte del código. Encontramos aquí 3 ficheros principales:
  - App.jsx → componente principal de la aplicación
  - index.css → para código CSS común a todos los componentes.
  - main.jsx → al que vinculamos con los dos anteriores (import) para que se inyecten en el archivo index.html.

5. Vamos a **modificar nuestro nuevo proyecto**. Necesitaremos el recurso 1.HTMLCSS-Inicio. En él encontramos un fichero HTML, otro CSS e imágenes. Movemos estos ficheros a nuestro proyecto de esta forma:

- Reemplazamos la carpeta public del proyecto por la del recurso de Aules.
- Reemplazamos el fichero index.css de src del proyecto por el del recurso de Aules. Ahora debemos ver esto:

# Tienda de Guitarras

- Del fichero index.html del recurso, copiamos las líneas del <head> relacionadas con Google Fonts (de la línea 8 a la 10) y las pegaremos en el index.html de nuestro proyecto

```
8 <link rel="preconnect" href="https://fonts.googleapis.com">
9 <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
10 <link href="https://fonts.googleapis.com/css2?family=Outfit:wght@400;700;900&display=swap" rel="stylesheet">
```

Podremos ver que nuestro texto cambia

# Tienda de Guitarras

## 6. Comenzamos a introducir **código en App.jsx del index.html**

Vamos a copiar casi todo el código del fichero index.html de recurso y lo pegaremos en el fichero App.jsx. Lo que seleccionaremos será:

- <header>... </header>
- <main> ...</main>
- <footer> ... </footer>

Lo pegaremos **sustituyendo** el **<h1> Tienda de Guitarras </h1>**

Nuestra aplicación se ve así:



## Nuestra Colección

	<b>LUKATHER</b> Lorem ipsum, dolor sit amet consectetur adipiscing elit. Sit quae labore odit magnam in autem nesciunt, amet deserunt		<b>SRV</b> Lorem ipsum, dolor sit amet consectetur adipiscing elit. Sit quae labore odit magnam in autem nesciunt, amet deserunt		<b>BORLAND</b> Lorem ipsum, dolor sit amet consectetur adipiscing elit. Sit quae labore odit magnam in autem nesciunt, amet deserunt
<b>299€</b>		<b>299€</b>		<b>299€</b>	
<b>AGREGAR AL CARRITO</b>		<b>AGREGAR AL CARRITO</b>		<b>AGREGAR AL CARRITO</b>	

## 4. ESCRIBIR CSS EN REACT

**Vite** nos permite incorporar imágenes, otros archivos de JS y archivos CSS.

Existen distintas formas de incorporar estilos CSS en React, de hecho es un tema muy extendido de esta biblioteca:

- Proyecto en HTML y CSS y moverlo a React
- SASS
- Módulos CSS
- Librerías basadas en Componentes
- Frameworks como Bootstrap o Tailwind → En el caso de nuestro ejemplo, se está utilizando una versión compilada de Bootstrap (por lo que no podremos añadir todas las clases que queramos de Bootstrap al proyecto). Tenemos los estilos globales en index.css y se importa desde main.jsx.

- Styled Components → consiste en crear componentes que tienen un diseño en CSS.

## 5. COMPONENTES EN REACT

Un componente **es una función de JS**. El fichero App.jsx es un componente. Los componentes DEBEN iniciar su nombre en mayúsculas.

```
function App() {  
  return (  
    <>  
  )  
}
```

Lo que hay entre las llaves de esta función es el código del componente.

**El código de este componente le pertenece pero puede pasarlo a otros componentes** a través de los llamados **props**.

Vemos en la captura la palabra reservada **return** y un paréntesis. Lo que está dentro de ese paréntesis es **lo que se muestra por pantalla**.

Básicamente un componente es tener HTML y JS en un solo archivo.

Los componentes tendrán extensión jsx si trabajamos en JavaScript o tsx si trabajamos con TypeScript.

Los **propósitos** de los componentes son:

- Ser reutilizable
- Separar la funcionalidad

No se tienen porqué cumplir los dos.

Un componente **siempre tendrá** un **return()** que es lo que se mostrará en pantalla. **Lo que se mostrará en pantalla será código HTML**.

Se recomienda crear los componentes dentro de una carpeta en **src**. Incluso se pueden tener distintas carpetas para separar, aún más, los componentes según su funcionalidad o uso.





Creamos una carpeta en **src**, llamada **components** y dentro de ella creamos el **componente Header.jsx**. Así sería su estructura básica:

```
guitarra > tiendaGuitarra > src > components > Header.jsx > ..
1
2 function Header(){
3
4
5   return(
6
7   )
8 }
9
```

Hemos dicho que un componente es una función y que siempre debe incluir `return()`, para mostrarlo en pantalla.

Una característica de los ficheros `.jsx` es que en el mismo fichero se puede mezclar código JS y HTML.



Escribimos un `h1` en `return` que ponga "Desde Header". Pero si vemos la página web, no se muestra y esto es porque el componente principal, **App.jsx NO sabe que este componente Header existe**. Lo que haremos será **exportar** el componente (en este caso `export default`, aunque no es necesario) y luego lo debemos importar desde el fichero que lo va a mostrar (`App.jsx`):

```
1
2 export default function Header(){
3
4
5   return(
6     <h1>Desde Header</h1>
7   )
8 }
9
```

```
TOS Y TYPESCRIPT > 01_guitarra > tiendaGuitarra > src > App.jsx > ..
1
2 import Header from "./components/Header"
3
4
5 function App() {
6   return (
```

Pero tampoco lo vemos reflejado en nuestro proyecto.

Esto sucede porque hemos importado el componente pero **tenemos que renderizarlo** (lo debemos incluir dentro del return del componente que lo va a mostrar).

Se renderiza a través de etiquetas HTML.



Vamos a **renderizar** nuestro nuevo componente justo al inicio de **return** (aparecerá en la parte superior de la página). Podemos ver que se diferencia de la sintaxis de HTML sobre todo por la inicial en mayúscula.

```
5  function App() {  
6    return (  
7      <>  
8      <Header/>  
9      <header class="py-5 header">  
10     <div class="container-xl">
```

Ahora sí lo vemos en nuestro proyecto:



Los componentes se pueden llamar múltiples veces sin problema, desde distintos componentes. Cada vez que se actualice un componente, se actualizará allí donde se le haya llamado.

## 6. JSX

JSX son las iniciales de JavaScript Syntax Extension.

Esta extensión fue desarrollada por Meta (Facebook) para React (los mismos creadores de la biblioteca).

Es una **sintaxis especial para poder agregar HTML y JavaScript en un solo fichero**.

Es un lenguaje de Templates/Vistas que muestra el HTML pero tiene todas las funciones de JS.

Todo lo que hay en JS puede ser utilizado en los ficheros jsx.

Una vez compilado, son archivos JS con funciones y objetos.

El código JS se escribirá en la parte superior de `return()` y lo mostraremos dentro de `return()`. Veamos un ejemplo:



Vemos que en el momento en que, en la parte de HTML, ponemos llaves, irá a buscar variables fuera de `return()`.

### 6.1. Reglas en jsx

Si un elemento HTML tiene una etiqueta de apertura, deberá tener también la de cierre. En el caso de etiquetas HTML que solo tienen apertura y no cierre (`link`, `img`, `input`) deberán finalizar con `/>`

```
return(
  <h2>Hola: {nombre}
)
```

```
[plugin:vite:react-swc] × Unexpected token. Did you mean `{'}` or `&rbrace;`?
└─[E:/CLASE/DAW/DWEC/24-25/BLOQUE3_FRAMEWORKS/REACT/PROYECTOS/1. FUNDAMENTOS Y
  TYPESCRIPT/01_guitarra/tiendaGuitarra/src/components/Header.jsx:9:1]
   6 |     return(
   7 |         <h2>Hola: {nombre}
   8 |     )
   9 | }
     | ^
```

Cada componente debe tener un return y en ese return solo un elemento debe estar en el nivel máximo.

```
return(
  <h2>Hola: {nombre}</h2>
  <p>Hola otra vez</p>
)
```

```
[plugin:vite:react-swc] × Expected ',', got 'otra'
└─[E:/CLASE/DAW/DWEC/24-25/BLOQUE3_FRAMEWORKS/REACT/PROYECTOS/1. FUNDAMENTOS Y
  TYPESCRIPT/01_guitarra/tiendaGuitarra/src/components/Header.jsx:8:1]
   5 |
   6 |     return(
   7 |         <h2>Hola: {nombre}</h2>
   8 |         <p>Hola otra vez</p>
```

La solución más rápida aparentemente es poner el código dentro de un elemento padre, a través de un div. Pero estamos añadiendo elementos que no necesitamos.

```
return(
  <div>
    <h2>Hola: {nombre}</h2>
    <p>Hola otra vez</p>
  </div>
)
```

# Hola: Natalia

Hola otra vez

Lo podemos solucionar con el **componente Fragment**.

Con **Fragment**, lo que conseguimos es poder devolver varios elementos en la misma línea jerárquica del código sin crear divs innecesarios.

Vemos **3 formas de incluir *Fragment***:

Ejemplo 1 → Importando Fragment directamente. Lo incluimos entre etiquetas de apertura y cierre.

```
1  import { Fragment } from "react"
2
3  export default function Header(){
4
5      const nombre = "Natalia"
6
7      return(
8          <Fragment>
9              <h2>Hola: {nombre}</h2>
10             <p>Usando Fragment: Ejemplo 1</p>
11          </Fragment>
12      )
13 }
```

---

# Hola: Natalia

Usando Fragment: Ejemplo 1

---

Ejemplo 2 → A través de React: lo importamos. Lo incluimos entre entre etiquetas de apertura y cierre.

```
1  import React from "react"
2
3  export default function Header(){
4      const nombre = "Natalia"
5
6      return(
7          <React.Fragment>
8              <h2>Hola: {nombre}</h2>
9              <p>Usando Fragment: Ejemplo 2</p>
10          </React.Fragment>
11      )
12 }
```

---

# Hola: Natalia

Usando Fragment: Ejemplo 2

---

Ejemplo 3 (favorita) → No importamos nada y sintaxis <> </>

```
2  export default function Header(){
3      const nombre = "Natalia"
4
5      return(
6          <>
7              <h2>Hola: {nombre}</h2>
8              <p>Usando Fragment: Ejemplo 3</p>
9          </>
10      )
11 }
```

---

# Hola: Natalia

Usando Fragment: Ejemplo 3

---

## 6.2. De class a className

Sabemos que jsx es una extensión de JavaScript que nos permite añadir código HTML en ficheros JS.

Si inspeccionamos nuestro proyecto, veremos que tenemos un warning:

```
Warning: Invalid DOM property `class`. Did you mean `className`?  
    at img  
    at a  
    at div  
    at div  
    at div  
    at header  
    at App
```

Esto es porque lo que hemos hecho es trasladar código HTML a un fichero jsx.

En HTML las clases se asignan con "class", y en JS "class" es una palabra reservada. Para poder darle el sentido que tiene debemos hacer un cambio:



Vamos a seleccionar del fichero "App.jsx" todos los "class" y lo cambiamos a "className" que es la palabra reservada de JS para referirse a las clases de HTML.

Antes:

```
2  import Header from "../components/Header"  
3  
4  
5  function App() {  
6    return (  
7      <>  
8        <Header/>  
9        <header class="py-5 header">  
10         <div class="container-xl">  
11           <div class="row justify-content-center justify-content-md-between">  
12             <div class="col-8 col-md-3">  
13               <a href="index.html">  
14                   
15               </a>  
16             </div>  
17             <nav class="col-md-6 a mt-5 d-flex align-items-start justify-content-end">  
18               <div
```

Después:

```
2 import Header from "../components/Header"
3
4
5 function App() {
6   return (
7     <>
8       <Header/>
9       <header className="py-5 header">
10         <div className="container-xl">
11           <div className="row justify-content-center justify-content-md-between">
12             <div className="col-8 col-md-3">
13               <a href="index.html">
14                 
15               </a>
16             </div>
17             <nav className="col-md-6 a mt-5 d-flex align-items-start justify-content-end">
18               <div
```

El warning habrá desaparecido.



Siguiendo con el proyecto de la tienda de guitarras, seleccionamos todo el **<Header></Header>** del fichero **App.jsx** y lo pegamos en nuestro nuevo componente **Header.jsx** (reemplazando todo lo que tenía).

```
1. FUNDAMENTOS Y TYPESCRIPT > 01_guitarra > tiendaGuitarra > src > App.jsx > App
1
2 import Header from "../components/Header"
3
4
5 function App() {
6   return (
7     <>
8       <Header/>
9
10      <main className="container-xl mt-5">
11        <h2 className="text-center">Nuestra Colección</h2>
12      </main>
13    </>
14  )
15}
```

```

1. FUNDAMENTOS Y TYPESCRIPT > 01_guitarra > tiendaGuitarra > src > components > Header.jsx
1
2  export default function Header(){
3
4
5      return(
6
7          <header className="py-5 header">
8              <div className="container-xl">
9                  <div className="row justify-content-center justify-content-between">
10                     <div className="col-8 col-md-3">

```

Así es como se ve nuestro proyecto:





## 7. PRACTICANDO CON LOS COMPONENTES

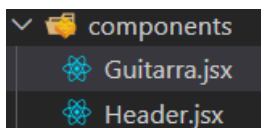
Este punto va a servir para practicar todo lo aprendido.

Partiendo del código HTML que tenemos en el componente App.jsx, vamos a crear otro para las guitarras.

### 7.1. Crear un componente



Vamos a crear un nuevo Componente (dentro de la carpeta components)



Dentro del componente crearemos la función (que se exportará) y pondremos el return necesario en todos los componentes.

```
1
2  export default function Guitarra() {
3
4      return(
5
6
7      )
8
9  }
```

En el fichero App.jsx tenemos un **<main>** que recoge todas las guitarras. El primer **<div>** de ese **<main>** contiene `<div className="row mt-5">` y es lo que nos permite (gracias a Bootstrap) ver el contenido en modo tabla.



Esto lo queremos seguir manteniendo, por lo que seleccionamos y cortamos el primer div hijo del citado anteriormente. Lo pegaremos dentro del return del nuevo Componente creado.

Así debe quedar nuestro nuevo componente **Guitarra.jsx**

```
1. FUNDAMENTOS Y TYPESCRIPT > 01_guitarra > tiendaGuitarra > src > components > Guitarra.jsx > Guitarra
1
2 export default function Guitarra() {
3
4   return(
5     <div className="col-md-6 col-lg-4 my-4 row align-items-center">
6       <div className="col-4">
7         
9       <div className="col-8">
10        <h3 className="text-black fs-4 fw-bold text-uppercase">Lukather</h3>
11        <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Sit qua
12        <p className="fw-black text-primary fs-3">299€</p>
13        <button
14          type="button"
15          className="btn btn-dark w-100"
16        >Agregar al Carrito</button>
17      </div>
18    </div>
19  )
20 }
21
22 }
```

Ahora borraremos la información del resto de guitarras del fichero **App.jsx**. El componente App.jsx (hasta el cierre de </main>, debe quedar así:

```
2 import Header from "./components/Header"
3
4
5 function App() {
6   return (
7     <>
8       <Header/>
9
10      <main className="container-xl mt-5">
11        <h2 className="text-center">Nuestra Colección</h2>
12
13        <div className="row mt-5">
14
15        </div>
16      </main>
17    </>
18  )
19 }
```



Donde antes teníamos TODAS las guitarras ahora vamos a llamar al Componente Guitarra. Debemos:

- Importar el componente (al inicio del fichero)
- Renderizarlo

Puesto que React está perfectamente integrado en JS, si llamamos al componente, directamente se importa.

El fichero **App.jsx** quedaría así (hasta `</main>`):

```
import Guitarra from "../components/Guitarra"
import Header from "../components/Header"

function App() {
  return (
    <>
      <Header/>

      <main className="container-xl mt-5">
        <h2 className="text-center">Nuestra Colección</h2>

        <div className="row mt-5">
          <Guitarra/>
        </div>
      </main>
    </>
  )
}
```

Y la aplicación se vería así (porque nuestro componente solo tiene información de una guitarra, es estático).



## Nuestra Colección



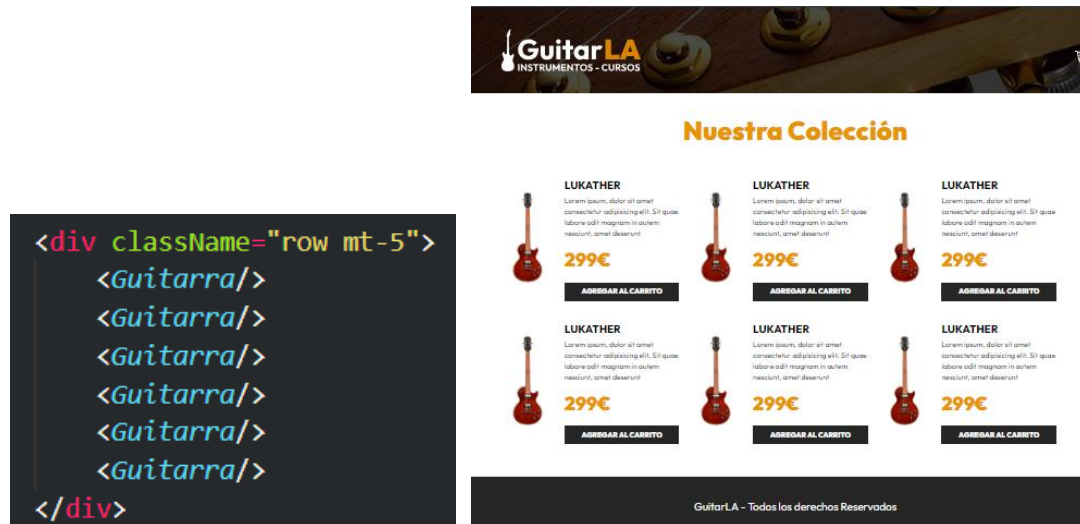
**LUKATHER**

Lorem ipsum, dolor sit  
amet consectetur  
adipiscing elit. Sit quae  
labore odit magnam in  
autem nesciunt, amet  
deserunt

**299€**

**AGREGAR AL CARRITO**

Para verlo más veces, podríamos renderizar muchas veces el componente, pero siempre se vería la misma información. NO he ganado nada.



## 8. REACT HOOKS

Los Hooks, al igual que los componentes es de lo más importante en React. Se pueden considerar la base de React.

**Permiten utilizar las diferentes funciones de React en los componentes.**

React tiene una serie de Hooks pero también podemos crear los nuestros.

Los Hooks están disponibles desde la versión 16.8 (actualmente 19), y antes de que apareciesen, se tenían que crear Clases para crear y modificar el State (característica de React), con los Hooks ya no es necesario, usaremos funciones.

Los Hooks se pueden dividir entre básicos y adicionales: Los básicos los vemos en **negrita** (estarán en todos los proyectos) y el resto son los adicionales.

Estos son básicamente los Hooks que existen en React.

<b>useState</b>	useCallback	useTransition	useImperativeHandle
<b>useEffect</b>	useMemo	useDeferredValue	useId
<b>useContext</b>	useRef	useLayoutEffect	useReducer
	useInsertionEffect	useSyncExternalStore	

## 8.1. State en React y Hook useState()

La pieza central o lo más importante en React es el State.

El State es una **variable con información relevante en nuestra aplicación** de React.

**Algunas veces el State pertenece a un componente específico u otros querremos compartirlo a lo largo de diferentes componentes.**

Lo podemos ver como el resultado de alguna interacción en el sitio o aplicación web: el listado de clientes, la imagen a mostrar en una galería, si un usuario está autenticado o no.

---

### 8.1.1. EJEMPLOS DE USESTATE()

Lo debemos importar:

```
Import {useState} from "react"
```

#### Ejemplo 1 →

```
const [ cliente, setCliente ] = useState( { } ) //Este será el valor inicial
```

**\*\*vemos xxx, setxxx → es una convención de React.**

#### Ejemplo 2 →

```
const [ total, setTotal ] = useState(0)
```

### Ejemplo 3 →

```
const [ productos, setProductos ] = useState([ ])
```

### Ejemplo 4 →

```
const [ modal, setModal ] = useState(false)
```

Dependiendo de qué tipos sean nuestros datos, será su valor inicial

---

#### 8.1.2. ANALIZANDO USESTATE()

Utilizamos el primer ejemplo:

```
const [ cliente, setCliente ] = useState({ });
```

es el State, la  
**variable** que tiene  
toda la información

**función que modifica**  
el State, se usará cuando  
queramos hacer cambios  
en el State (cliente)

**valor inicial** del State

**React reacciona ante cada State:** cada vez que un State cambie, la aplicación de React va a renderizar y actualizarse con esos cambios, no hay que hacer más y la Interfaz siempre estará sincronizada con los State → NO recargamos la página, NO actualizamos el DOM...

Para modificar el State utilizamos la función que extraemos cuando declaramos el State, en el caso de ejemplo **setCliente**, NO haremos nunca:

**cliente = xxx**

### 8.1.3. IMPLEMENTANDO USESTATE()

Después de ver ejemplos de un Hook concreto de React (`useState()`), para implementarlos, debemos conocer las **reglas** de los Hooks:

1. Los Hooks se colocan en la parte superior de los componentes.
2. NO se deben colocar dentro de condicionales (para no variar el número de hooks registrados), ni bucles, tampoco después de un `return`.
3. NO se deben colocar dentro de funciones.

**Ejemplo** → error por colocar Hooks dentro de condicional

```
7
8 function App() {
9
10   //State
11   const [auth, setAuth] = useState(true);
12   const [total, setTotal] = useState(0);
13
14   if (auth){
15     const [cart, setCart] = useState([]);
16     console.log(cart);
17   }
18
19   setTimeout(() => {
20     setAuth(false)
21   }, 3000);
22
23
24   return (
```

La primera vez que se ejecute, aparecerá el array vacío por consola.

Se han contabilizado 3 Hooks.

Después del `setTimeout()`, dará error porque se ha cambiado el State y se contabilizan solo 2 Hooks.

## 8.2. Hook `useEffect()`

Después de `useState()`, es el Hook más utilizado.

Es un Hook que se utiliza en distintos escenarios.

Siempre es un **callback**, que, dependiendo de como se declare, realizará distintas acciones.

Es el sustituto de lo que era `componentDidMount()` y `componentDidUpdate()` de versiones previas de React donde había clases y lo que se conocían como métodos de ciclo de vida que hoy en día ya no se utilizan.

Vemos la sintaxis:

`Import {useEffect} from "react"`

```
useEffect( () => {  
    console.log("El componente está listo");  
}, []);
```

Los corchetes del final es el array de dependencias.

Los **usos de useEffect** son:

Se ejecuta automáticamente cuando el componente está listo, es un buen lugar para colocar código para consultar una API o para obtener datos de LocalStorage.

Debido a que le podemos pasar una dependencia (y esta dependencia va a ser un State) y estar escuchando los cambios que sucedan en una variable (o en ese State), puede actualizar el componente (o ejecutar ciertas funciones) cuando ese cambio suceda.

Dependiendo del valor que pasemos en el array de dependencias (o no pasemos nada), el hook de useEffect hará algo diferente.

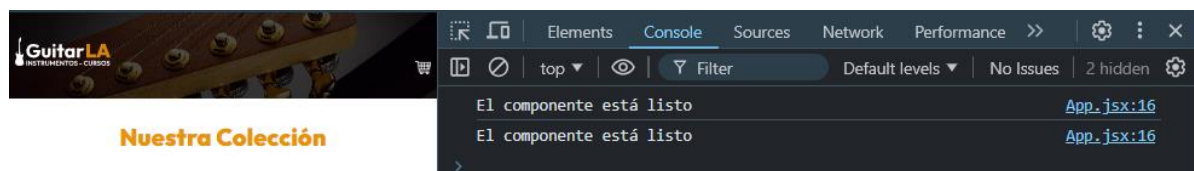
Ejemplos → sin dependencias (izq) y con dependencias (dcha)

```
2 import { useState, useEffect } from "react"  
3 import Guitarra from "../components/Guitarra"  
4 import Header from "../components/Header"  
5  
6  
7  
8 function App() {  
9  
10    //State  
11    const [auth, setAuth] = useState(true);  
12    const [total, setTotal] = useState(0);  
13    const [cart, setCart] = useState([]);  
14  
15    useEffect(() => {  
16        console.log('El componente está listo');  
17    }, []);
```

```
2 import { useState, useEffect } from "react"  
3 import Guitarra from "../components/Guitarra"  
4 import Header from "../components/Header"  
5  
6  
7  
8 function App() {  
9  
10    //State  
11    const [auth, setAuth] = useState(true);  
12    const [total, setTotal] = useState(0);  
13    const [cart, setCart] = useState([]);  
14  
15    useEffect(() => {  
16  
17    }, [cart, auth, total]);
```

Las dependencias pueden ser States o variables. Cada vez que cambie alguno de ellos, se ejecutará el hook useEffect.

Si es un array vacío, NO hay dependencias, por lo que solo se ejecutará una vez, cuando el componente esté listo.



Vemos dos líneas en consola (por la doble renderización de React). Lo podemos eliminar en el fichero main.jsx eliminando el modo estricto (pero no se recomienda)



```

1. FUNDAMENTOS Y TYPESCRIPT > 01_guitarra > tiendaGuitarra > src > main.jsx
1  import { StrictMode } from 'react'
2  import { createRoot } from 'react-dom/client'
3  import App from './App.jsx'
4  import './index.css'
5
6  createRoot(document.getElementById('root')).render(
7    <StrictMode>
8      <App />
9    </StrictMode>,
10 )

```

Puede ser que queramos que `useEffect` se ejecute si el State tiene un valor determinado. Podemos utilizar dentro del hook condicionales.

Ejemplo → condicional dentro de `useEffect()`

```

8  function App() {
9
10     //State
11     const [auth, setAuth] = useState(false);
12
13     useEffect(() => {
14       if(auth)
15         console.log('Usuario autenticado correctamente');
16     }, [auth]);
17
18     setTimeout(() => {
19       setAuth(true)
20     }, 3000);
21
22
23     return (

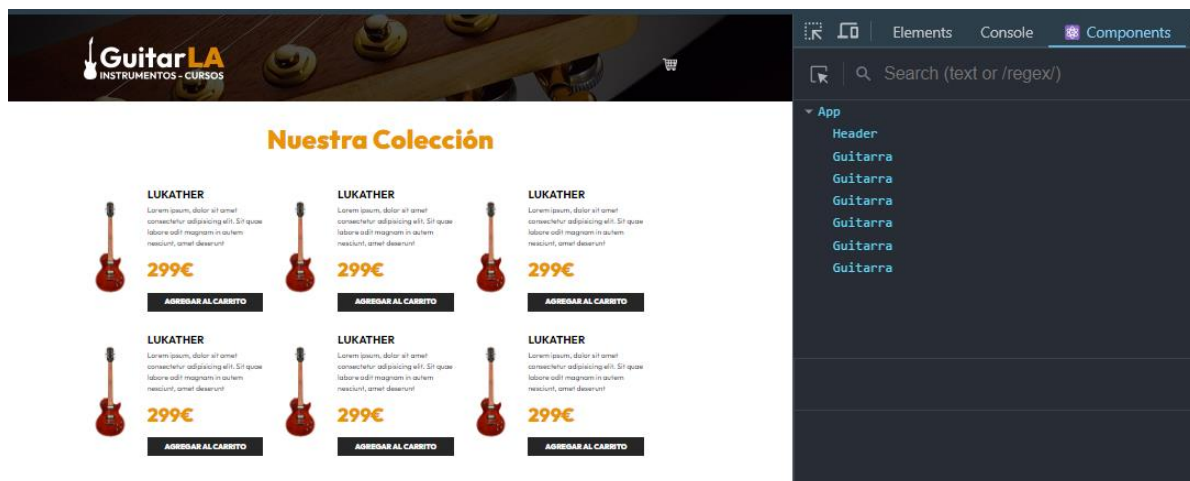
```

Si ejecutamos, al cargarse el componente NO vemos nada en consola (`auth = false`). Cuando pasan 3 segundos y el State cambia, se ve el mensaje en consola.

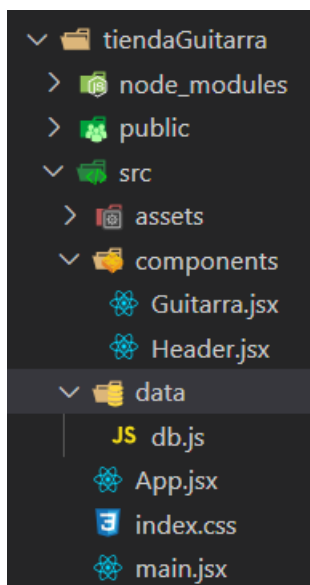
## 8.3. Practicando con Hooks

Si ejecutamos nuestra aplicación, vemos varias guitarras, todas con la misma información. Esto es porque estamos renderizando el componente guitarra varias veces.

Vemos el árbol de componentes de React:



Lo que vamos a hacer ahora es incorporar información de nuestra “base de datos” para que nuestra aplicación muestre los datos de forma dinámica.



Para ello, crearemos en nuestro proyecto una carpeta llamada “data” dentro de “src” y añadiremos el fichero “db.js” que contiene un array de objetos. Será nuestra base de datos.

El contenido de db.js es de un array con 12 objetos con id, name, image (las de la carpeta public), description y price)



En App.jsx, importaremos useState y useEffect from ‘react’ y también vamos a importar nuestra BBDD

```

2 import { useState, useEffect } from "react"
3 import Guitarra from "../components/Guitarra"
4 import Header from "../components/Header"
5 import {db} from "../data/db"

```

Recordamos cómo debemos tener nuestro componente App.jsx (hasta el footer)

```

7 function App() {
8
9
10 return (
11   <>
12     <Header/>
13
14     <main className="container-xl mt-5">
15       <h2 className="text-center">Nuestra Colección</h2>
16
17       <div className="row mt-5">
18         <Guitarra/>
19         <Guitarra/>
20         <Guitarra/>
21         <Guitarra/>
22         <Guitarra/>
23         <Guitarra/>
24       </div>
25     </main>

```

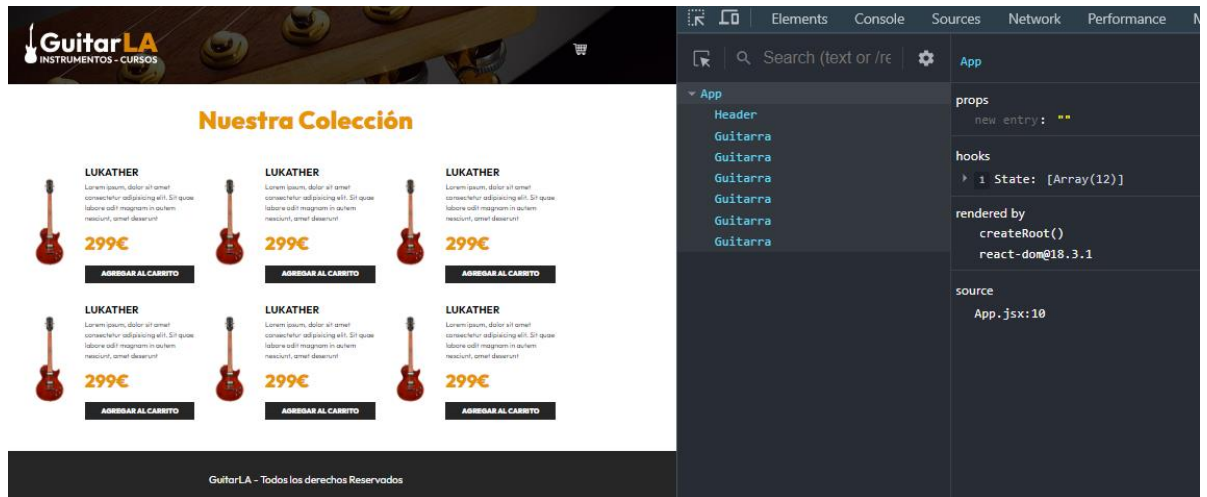


En **App.jsx**, creamos el State de nuestro array de guitarras. Para cargar nuestros datos lo podemos hacer de dos formas:

1. Lo inicializamos con la variable que acabamos de importar (db)

```
const [data, setData] = useState(db);
```

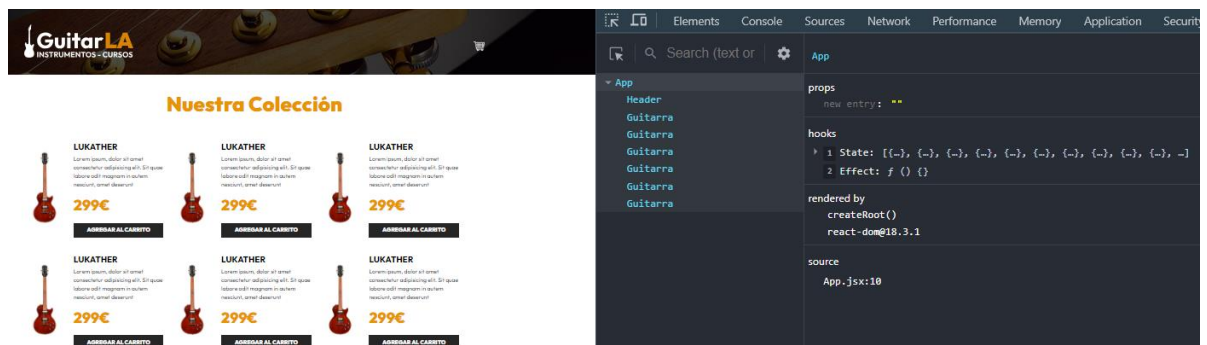
Al ser un archivo local, lo carga y la información la tenemos disponible. Si inspeccionamos la página y vemos el componente con React Developer Tools, vemos el apartado hooks en el componente App.



2. Inicializamos el State como array vacío y utilizamos `useEffect()`. Cuando el componente cargue, se modifica el valor del State data.

```
10 const [data, setData] = useState([])
11
12 useEffect(()=>{
13   |   setData(db)
14 }, []);
```

Si inspeccionamos la página y vemos el componente con React Developer Tools, vemos el apartado hooks en el componente App.



Tenemos dos opciones, al ser un archivo local, podemos usar la primera, si fuera una consulta a una API, la segunda es más recomendable porque se llamará cuando el componente esté listo.

En nuestro caso, usaremos la primera opción (porque es un archivo local).

## 9. STATEMENTS AND EXPRESSIONS

En JS debemos diferenciar los Statements de las Expressions

Una app de JS tiene una serie de Statements, cada **Statement** es una instrucción para hacer algo (creación de variables, códigos condicionales, lanzamiento de errores con *throw new Error()*, iterar con *while* o *for...*)

Una **Expression** es algo que produce un valor. Una vez utilizada va a generarnos un valor nuevo (ternarios, usar un *array method* que genere un nuevo array, usar *map* que genera un nuevo array en lugar de *forEach*).

**Los Statements los colocamos ANTES del return**  
**Las Expressions las colocamos DESPUÉS del return**

Vamos a nuestro proyecto:

Tenemos un State (data) que lo hemos inicializado a un array de objetos.

Queremos recorrer ese array para que se muestren las guitarras en la pantalla.

Recordamos nuestro código (no vemos los imports y el footer) y lo que se ve:

```

7  function App() {
8
9      //State
10     const [data, setData] = useState(db);
11
12     return (
13         <>
14             <Header/>
15
16             <main className="container-xl mt-5">
17                 <h2 className="text-center">Nuestra Colección</h2>
18
19                 <div className="row mt-5">
20                     <Guitarra/>
21                     <Guitarra/>
22                     <Guitarra/>
23                     <Guitarra/>
24                     <Guitarra/>
25                 </div>
26             </main>
27         </>
28     );
29 }

```



Vamos a iterar con **map**. Se ejecutará una vez por cada elemento del array.

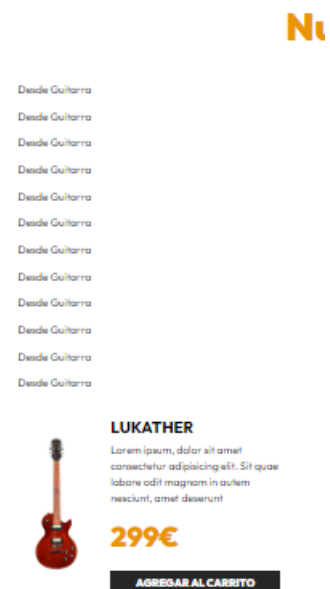
En **App.jsx** dejaremos una única referencia al componente `<Guitarra/>`

Justo encima del componente `<Guitarra/>`, pondremos el código entre llaves (para indicar que es código JS).

```

<main className="container-xl mt-5">
  <h2 className="text-center">Nuestra Colección</h2>
  <div className="row mt-5">
    {data.map(() => (
      <p>Desde Guitarra</p>
    ))}
    <Guitarra/>
  </div>
</main>

```



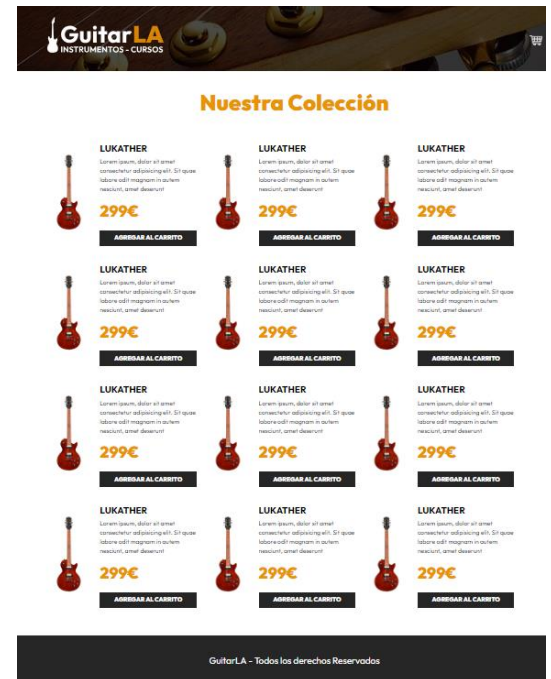
Lo que vamos a hacer es mostrar todas las guitarras. Sustituimos el párrafo por el componente.

```
<main className="container-xl mt-5">
  <h2 className="text-center">Nuestra Colección</h2>
  <div className="row mt-5">
    {data.map(() => (
      <Guitarra/>
    ))}
  </div>
</main>
```

Vemos que se muestra el componente tantas guitarras como elementos tiene el array.

La información siempre es la misma porque en el componente **Guitarra.jsx** la información es estática.

Para hacerlo dinámico usaremos **Props**.



## 10. PROPS

Es una forma de compartir información entre componentes. Se puede pasar información de un componente padre al hijo por medio de los Props. Se pasan como **objetos**.

Los Props son muy similares a los atributos en HTML, pero podemos pasarles arrays, objetos o funciones.

La sintaxis es:

```
<Header
  nombreProp = {datos/State/funciones}
/>
```

Podemos tener distintos Props:

```
<Users
  users = {users}
  setUsers = {setUsers}
  titulo = "Listado de Usuarios"
/>
```

- El nombre, a través del cual accederemos a él en el otro Componentes (izq)
- El valor que se le quiere pasar (dcha)

**Los Props se pasan de padres a hijos, NUNCA al contrario.**

**Si tenemos un State que se va a pasar por diferentes componentes, lo colocaremos en el archivo principal.**



Vamos a implementar Props en nuestro proyecto.

Recordamos que estamos iterando un array de objetos.

Debajo del componente `<Guitarra/>` vamos a ir añadiendo los Props.

```
<div className="row mt-5">
  {data.map(element => (
    <Guitarra
      guitarraObj = {element}
    />
  )
  )}
</div>
```

Nuestro Prop se llama  
"guitarraObj" y le pasamos el objeto

En el Componente hijo (en nuestro caso `Guitarra.jsx`) le pasaremos el Prop como argumento a la función (es un objeto). Hacemos un `console.log` para ver lo que recibe.

```
export default function Guitarra({guitarraObj}) {
  console.log(guitarraObj)
```



<pre> {id: 1, name: 'Lukather', image: 'guitarra_01', description: 'Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.', price: 299} </pre>	Guitarra.jsx:4
[object Object]	Guitarra.jsx:4
<pre> {id: 2, name: 'SRV', image: 'guitarra_02', description: 'Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.', price: 349} </pre>	Guitarra.jsx:4
[object Object]	Guitarra.jsx:4
<pre> {id: 3, name: 'Borland', image: 'guitarra_03', description: 'Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.', price: 329} </pre>	Guitarra.jsx:4
[object Object]	Guitarra.jsx:4
<pre> {id: 4, name: 'VAI', image: 'guitarra_04', description: 'Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.', price: 299} </pre>	Guitarra.jsx:4
[object Object]	Guitarra.jsx:4

En la página web aún no se ve nada porque los datos son estáticos. Lo que haremos será renderizar los datos del Props. Para tener un código limpio podemos usar destructuring.

Lo que haremos será sustituir en cada dato estático su correspondiente valor de los Props.

Partimos de esto 8 (datos estáticos)

```

1
2 export default function Guitarra({guitarraObj}) {
3
4   const {id, name, image, description, price} = guitarraObj;
5
6   console.log(guitarraObj)
7
8   return(
9     <div className="col-md-6 col-lg-4 my-4 row align-items-center">
10       <div className="col-4">
11         
12       </div>
13       <div className="col-8">
14         <h3 className="text-black fs-4 fw-bold text-uppercase">Lukather</h3>
15         <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Sit quae labore odit magna
16         <p className="fw-black text-primary fs-3">299€</p>
17         <button
18           type="button"
19           className="btn btn-dark w-100"
20         >Agregar al Carrito</button>
21       </div>
22     </div>
23

```

Y acabamos con esto:

```

1
2 export default function Guitarra({guitarraObj}) {
3
4   const {id, name, image, description, price } = guitarraObj;
5
6   return(
7     <div className="col-md-6 col-lg-4 my-4 row align-items-center">
8       <div className="col-4">
9         <img className="img-fluid" src={` /img/${image}.jpg`} alt="imagen guitarra"
10        </div>
11       <div className="col-8">
12         <h3 className="text-black fs-4 fw-bold text-uppercase">{name}</h3>
13         <p>{description}</p>
14         <p className="fw-black text-primary fs-3">{price}€</p>
15         <button
16           type="button"
17           className="btn btn-dark w-100"
18         >Agregar al Carrito</button>
19       </div>
20     </div>
21   )
22 }

```

Para las imágenes debemos tener en cuenta que están en la carpeta public (todo el proyecto tiene acceso a public).

NO hemos usado de momento el elemento id.

Recordamos que ponemos llaves para indicar que el código que va dentro es JS.



Si inspeccionamos, podemos ver un Warning:

```
✖ ▶ Warning: Each child in a list should have a unique "key" prop. App.jsx:18
Check the render method of `App`. See https://reactjs.org/link/warning-keys for more
information.
    at Guitarra (http://localhost:5173/src/components/Guitarra.jsx?t=1727896455385:10:36)
    at App (http://localhost:5173/src/App.jsx?t=1727896455385:18:29)
```

Esto es porque se está creando una lista y hay que pasar un Prop especial llamado **key** (siempre tendrá ese nombre). **Esto ocurrirá cada vez que iteremos un State.**

Le tenemos que **pasar a ese Componente un identificador único** para que React identifique cada componente que ha de crear (porque aplica algunas mejoras al performance (calidad de las salidas del sistema en respuesta a las entradas del usuario). En nuestro caso, le pasaremos el ID.

```
{data.map(element => (  
  <Guitarra  
    key = {element.id}  
    guitarraObj = {element}  
  />  
)  
)}
```

## 11. EVENTOS

La forma en que React trata los eventos es muy similar a como lo hace JS, aunque hay algunos cambios.

Los eventos son camelCase. Si pensamos en los eventos en línea, en JS es onclick y en React onClick.

A diferencia de JS y HTML, donde se coloca el nombre de la función en un String, en React (JSX), se utiliza la función entre llaves.

Comparamos la sintaxis:

HTML:

```
<button onclick = "nomFuncion()">
</button>
```

JSX:

```
<button onClick = {nomFuncion()}>
</button>
```

HTML:

```
<form onsubmit = "agregarCliente(); return false">
  <button type="submit">Submit</button>
</form>
```

JSX:

```
<form onSubmit = {handleSubmit}>
  <button type="submit">Añadir</button>
</form>
```

La función la podemos colocar antes del return o en otro fichero, aunque se recomienda que esté en un hook personalizado o en el mismo componente.

El nombre **“handle+Evento” es una convención de React.**



Vamos a 'jugar' con los eventos. Estos pequeños ejercicios no forman el código final, pero se recomienda su práctica.

Vamos al componente Guitarra.jsx y detectamos el botón. Le pondremos el nombre del evento entre llaves (así indicamos que va código JS)

```
<button
  type="button"
  className="btn btn-dark w-100"
>Agregar al Carrito</button>
```

```
<button
  type="button"
  className="btn btn-dark w-100"
  onClick={handleClick}
>Agregar al Carrito</button>
```

En la parte superior de return, justo debajo del destructuring del objeto, desarrollamos la función del evento, por ahora un mensaje en consola. Nos quedaría así:

```
export default function Guitarra({guitarraObj}) {

  const {id, name, image, description, price } = guitarraObj;

  const handleClick = () => {
    console.log('Desde click....')
  }

  return(
```

En este momento, independientemente del botón de la guitarra que presionemos, el mensaje es el mismo.



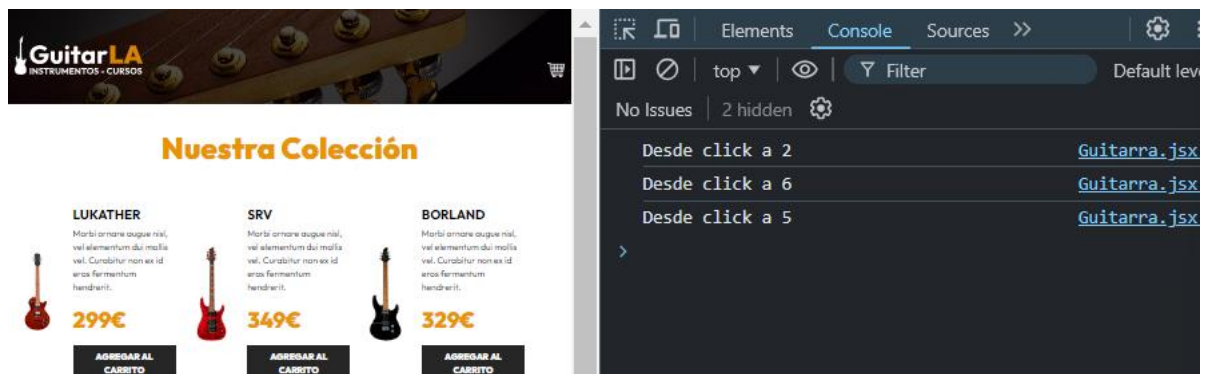
Queremos detectar la guitarra concreta cuando se clicca sobre el botón "agregar al carrito". Para ello, utilizaremos el id:

En el **evento**, como ahora tenemos que llamar a una función con argumentos (antes llamábamos a una constante ya ahora le vamos a pasar el id), lo convertiremos en una arrow function (para evitar que, cuando cargue el componente llame directamente a la función sin esperar el evento).

```
<button
  type="button"
  className="btn btn-dark w-100"
  onClick={() => handleClick(id)}
>Agregar al Carrito</button>
```

En la función, pasaremos el id y lo vamos a mostrar

```
const handleClick = (id) => {
  console.log(`Desde click a ${id} `);
}
```



El objetivo de este evento es añadir nuestra guitarra al carrito. En el carrito se ve la información completa (excepto descripción) de cada guitarra

En ese carrito se tendrán que listar las guitarras, se podrán añadir o eliminar unidades de una guitarra y se podrá vaciar.



Debemos hacer un **State** para **carrito**, ya que será una variable que irá sufriendo cambios a lo largo de la interacción con el usuario.

El **State** lo vamos a crear en el componente principal, **App.jsx** (no podemos crearlo en el componente Guitarra.jsx porque ese componente tiene 12 guitarras, por lo que tendríamos 12 carritos).

Lo vamos a crear debajo del State de datos y lo inicializamos a un array vacío (puesto que es donde vamos a ir incorporando las guitarras del pedido)

```
7  function App() {  
8  
9      //State  
10     const [data, setData] = useState(db);  
11     const [carrito, setCarrito] = useState([]);  
12  
13     return (  

```

Si vemos **Inspeccionar** en la consola de la página web, podremos ver los dos State del componente App.jsx



Para poder modificar el **State carrito** necesitaremos tener acceso a la función del evento que hemos creado en el componente Guitarra.jsx.

Para ello le podemos pasar al componente hijo Guitarra.jsx la función que edita el carrito, y lo haremos a través de **Props** (vimos que podíamos pasar strings, arrays, objetos o funciones).

```
{data.map(element => (  
  <Guitarra  
    key = {element.id}  
    guitarraObj = {element}  
    setCarrito = {setCarrito}  
  />  
  )  
)}  
})
```

En el componente Guitarra.jsx, además de extraer la guitarra, también extraemos la función que le hemos pasado (así la podremos utilizar)

```
export default function Guitarra({guitarraObj, setCarrito}) {
```

## 12. FORMAS DE MODIFICAR UN STATE

Cuando definimos un State y un setState debemos saber que estos dos elementos están conectados.

Encontramos varias formas de modificar un State. Lo vamos a ver con nuestro ejemplo de guitarras.

El State carrito es un array por lo que, cada vez que se clique sobre una guitarra ("agregar al carrito") se debería añadir ese objeto al array del State.

### 12.1. Pasar el State a través de Props

Podemos pasar el State a través de Props al componente Guitarra. En el evento del botón le pasaremos el objeto. Ya en la función, añadimos el objeto a una copia del State.

```
<Guitarra
  key = {element.id}
  guitarraObj = {element}
  carrito = {carrito}
  setCarrito = {setCarrito}
/>
```

```
<button
  type="button"
  className="btn btn-dark w-100"
  onClick={() => handleClick(guitarraObj)}
>Agregar al Carrito</button>
```

```
export default function Guitarra({guitarraObj, carrito, setCarrito}) {

  const {id, name, image, description, price } = guitarraObj;

  const handleClick = (guitarraObj) => {
    setCarrito([...carrito, guitarraObj ]);
  }
}
```

#### 12.1.1. SIMPLIFICANDO CÓDIGO



Sin salir de este ejemplo, podemos, cuando el código de la función de un evento es muy escueto, lo podemos incluir en el código de la propia arrow function del botón.

El código del botón quedaría así:

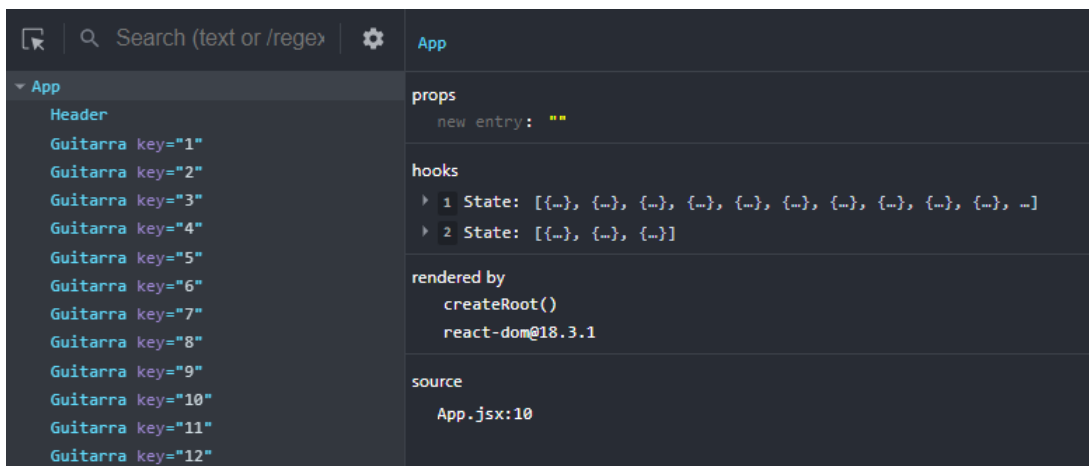
```
<button
  type="button"
  className="btn btn-dark w-100"
  onClick={() => setCarrito([...carrito, guitarraObj])}
>Agregar al Carrito</button>
```

Y la función del componente así:

```
export default function Guitarra({guitarraObj, carrito, setCarrito}) {
  const {id, name, image, description, price } = guitarraObj;

  return(
```

Si inspeccionamos con React Developer Tools, vemos que, si vamos añadiendo guitarras al carrito, el State de carrito se va modificando con éxito.



## 12.2. Usando un PrevState

Ya sabemos que una función `setXXX` está conectado al State `XXX`, esto implica que sabe en todo momento qué valor tiene el State para poder modificarlo.

Si esto es así, NO es necesario pasar por Props el State.

React aporta la opción de pasar directamente a `setXXX` el valor previo del State `XXX`. Como convención se utiliza **`prevXXX`**, aunque podemos usar el nombre que queramos.



Vamos a probar este método.

NO le pasaremos el State, por lo que los Props siguen siendo estos:

```
<Guitarra
  key = {element.id}
  guitarraObj = {element}
  setCarrito = {setCarrito}
/>
```

En el componente `Guitarra.jsx`, el código del evento quedaría así (directamente en el botón):

```
<button
  type="button"
  className="btn btn-dark w-100"
  onClick={() => setCarrito(prevCarrito => [...prevCarrito, guitarraObj])}
>Agregar al Carrito</button>
```

Utilizando este método, conseguimos modificar el State pero **si la lógica de la función va creciendo NO es conveniente que esté en el template del componente** (que debe servir para mostrar por pantalla información de los componentes).

Podemos hacer dos cosas:

1. Poner la función en la parte de la lógica del componente hijo y desde el botón únicamente llamar a la función o
2. Crear una función en el componente padre (`App.jsx`) y pasarla por Props al componente hijo (`Guitarra.jsx`)



Vamos a utilizar la **segunda** opción. Será una forma también de asegurar que podemos reutilizar código.

Creamos una función en la parte de lógica del componente padre (**App.jsx**) que utilizará el **setState** con la información previa del propio **State** (se utiliza *articulo* para generalizar el producto de un carrito):

```
function App() {  
  
  //State  
  const [data, setData] = useState(db);  
  const [carrito, setCarrito] = useState([]);  
  
  function anyadirAlCarrito(articulo){  
    setCarrito(prevCarrito => [...prevCarrito, articulo])  
  }  
  
  return (  

```

También tenemos la opción de utilizar directamente el **State** puesto que estamos en el mismo componente donde se declaró. Recordamos que al usar **Spread Operator** NO estamos modificando el **State** original.

```
function anyadirAlCarrito(articulo){  
  setCarrito(carrito => [...carrito, articulo])  
}
```

Por otro lado, en los **Props**, ya NO necesitamos pasar el **useState**, ahora pasaremos la función:

```
<Guitarra  
  key = {element.id}  
  guitarraObj = {element}  
  anyadirAlCarrito = {anyadirAlCarrito}  
>
```

En el componente hijo (Guitarra.jsx), en la definición de la función encontramos:

```
export default function Guitarra({guitarraObj, anyadirAlCarrito}) {
```

y la lógica del evento del botón será muy simple:

```
<button  
  type="button"  
  className="btn btn-dark w-100"  
  onClick={() => anyadirAlCarrito(guitarraObj)}  
>Agregar al Carrito</button>
```

## 13. INMUTABILIDAD

Seguimos con el proyecto y, una vez organizado el código, la lógica sigue siendo la misma. Si clicamos varias veces sobre la misma guitarra, se añade al carrito el objeto entero.

Queremos que el campo cantidad, se pueda modificar.

Para ello trabajaremos sobre la función del componente padre (App.jsx) que permite añadir las guitarras al carrito.

Estamos utilizando Spread Operator, hacemos una copia de nuestro State y le añadimos la guitarra seleccionada.

Si recordamos los Array Methods, sabremos que utilizar Spread Operator es lo mismo que utilizar push(). La diferencia es que el método propio de Array está modificando el array original.

**Los States en React son INMUTABLES** → no los podemos modificar, por eso tenemos una función para hacerlo, para cambiar su valor.

Pero aunque usemos la función setState para modificarlo, NO podremos utilizar métodos que lo muten o cambien su valor original.

Podemos consultar esta [guía](#) para verificar qué métodos modifican o no el array original.

### 13.1. Practicamos inmutabilidad y eventos



Volvemos al proyecto, **queremos saber si en nuestro carrito se encuentra ya la guitarra que pulsamos para no repetir registros.**

Vamos a utilizar un método de array que NO modifique el original → `findIndex()`

Si NO encuentra la guitarra, la tendrá que añadir.




Si clicamos sobre una guitarra que ya existe, lo que queremos es que se incremente una unidad. En el array de objetos que nos dan (simulando la base de datos), NO existe la propiedad cantidad, por lo que se lo vamos a añadir.

Antes de incorporar la guitarra al carrito, crearemos la propiedad y le daremos valor 1 (ya que la primera vez que clicamos, no existe la guitarra en el carrito).

Nuestra función quedaría así:

```
function anyadirAlCarrito(articulo){  
  
  const articuloExiste = carrito.findIndex(element => articulo.id === element.id);  
  if (articuloExiste >= 0){  
    console.log('ya existe...')  
  }else{  
    articulo.cantidad = 1;  
    setCarrito(carrito => [...carrito, articulo])  
  }  
}
```


Si ejecutamos, podemos ver como se añade la guitarra al State con su nuevo atributo:



Curabitur non ex id eros fermentum hendrerit.

**299€**


**AGREGAR AL CARRITO**



**SRV**  
Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.

**349€**

**AGREGAR AL CARRITO**



**BORLAND**  
Morbi ornare augue nisl, vel elementum dui mollis vel. Curabitur non ex id eros fermentum hendrerit.

**329€**

**AGREGAR AL CARRITO**

Elements Console Components >>

Search (text or /regex/)

App

- Header
  - Guitarra key="1"
  - Guitarra key="2"
  - Guitarra key="3"
  - Guitarra key="4"
  - Guitarra key="5"
  - Guitarra key="6"
  - Guitarra key="7"
  - Guitarra key="8"
  - Guitarra key="9"
  - Guitarra key="10"
- App

props

new entry: ""

hooks

- 1 State: [{-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}]
- 2 State: [{-}]
  - 0: {cantidad: 1, description: "Morbi ornare augue nisl, vel elementum dui mollis vel.", id: 2, image: "guitarra\_02", name: "SRV", price: 349, cantidad: 1, new\_entry: ""}

rendered by

createRoot()

react-dom@18.3.1



Si clicamos otra vez sobre la misma guitarra, la cantidad se debería modificar e incrementarse en una unidad.

Debemos saber que NO podemos modificar el State, por lo que algo así NO sería válido:

```
const articuloExiste = carrito.findIndex(element => articulo.id === element.id);
if (articuloExiste >= 0){//existe en el carrito
  carrito[articuloExiste].cantidad++;
```

Si ejecutamos, vemos que funciona pero no se actualizan directamente.

Lo que debemos hacer para no incumplir las reglas de la inmutabilidad de React es **crear una copia del State y settearla o modificarla más tarde**.

Nuestra función quedaría así:

```
function anyadirAlCarrito(articulo){

  const articuloExiste = carrito.findIndex(element => articulo.id === element.id)
  if (articuloExiste >= 0){//existe en el carrito
    const copiaCarrito = [...carrito]
    copiaCarrito[articuloExiste].cantidad++
    setCarrito(copiaCarrito)
  }else{
    articulo.cantidad = 1
    setCarrito(carrito => [...carrito, articulo])
  }
}
```

Una vez que ya se nos actualiza la cantidad de las guitarras evitando duplicar registros, **vamos a mostrar en el carrito de la página web su contenido**.

Esto es lo que tenemos que mostrar en el carrito (la información que se ve en la página web en este momento es estática).





El código de esa parte lo encontramos en el componente **Header.jsx**. En ese fichero, el carrito se encuentra en una tabla de Bootstrap.

Lo que vamos a hacer es sincronizar esa parte de código con nuestro State carrito.

Para ello **necesitaremos pasarle a Header el carrito**. Lo tenemos que hacer desde el componente padre o principal (App.jsx)

```
return (  
  <>  
    <Header  
      carrito = {carrito}  
    />  
  </>  
)
```

Por otro lado, en el componente **Header.jsx**, debemos extraer el State que acabamos de pasar por Props:

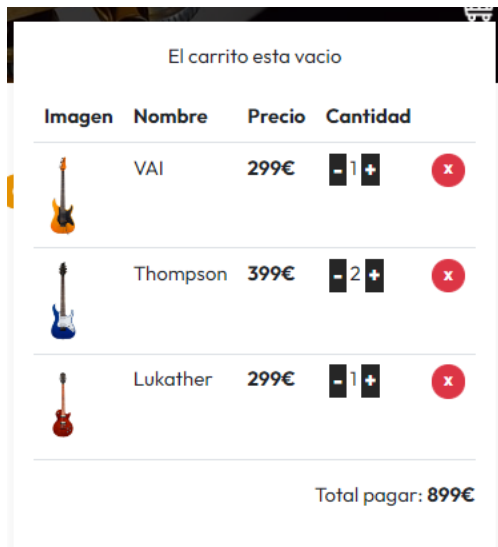
```
export default function Header({carrito}){
```

Lo que debemos hacer es iterar sobre el carrito de compras para mostrarlo. Lo haremos debajo <tbody>. La idea es que muestre la tabla que ya tenemos pero con los elementos del array de carrito (lo convertimos en dinámico).

```
    <tbody>  
      {carrito.map(element =>  
        <tr>  
          <td>  
              
          <td>SRV</td>  
          <td className="fw-bold">  
            299€  
          </td>  
        </tr>  
      )
```

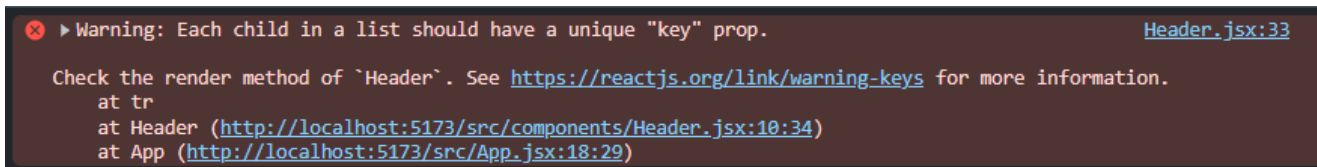
**Este pequeño ejercicio se deja para la práctica del alumnado.**

Recordemos que, para la imagen, la carpeta public NO la necesitamos (esto viene de HTML), en React, el contenido de la carpeta public es accesible desde cualquier punto.



Nuestro carrito ya se llena correctamente, con los datos de cada guitarra.

Si ejecutamos, vemos que nos sale el mismo Warning que antes: siempre que iteremos (por ejemplo con map), debemos tener una clave única.



Si nos remontamos a la vez anterior que iteramos era sobre un componente. La clave única la reflejamos así (izq). Ahora iteramos sobre HTML, lo haremos así (dcha):

```
{data.map(element => (
  <Guitarra
    key = {element.id}
    guitarraObj = {element}
    anyadirAlCarrito = {anyadirAlCarrito}
  />
)}
)
```

```
{carrito.map(element => (
  <tr key={element.id}>
    <td>
      <img className="img-fluid"
        src={`~/img/${element.image}.jpg`}
        alt="imagen guitarra"
      />
    </td>
    <td>{element.name}</td>
    <td className="fw-bold">
      {element.price}€
    </td>
    <td className="flex align-items-start gap-4">
      <button
        type="button"
        className="btn btn-dark"
      >
        -
      </button>
      {element.cantidad}
      <button
```

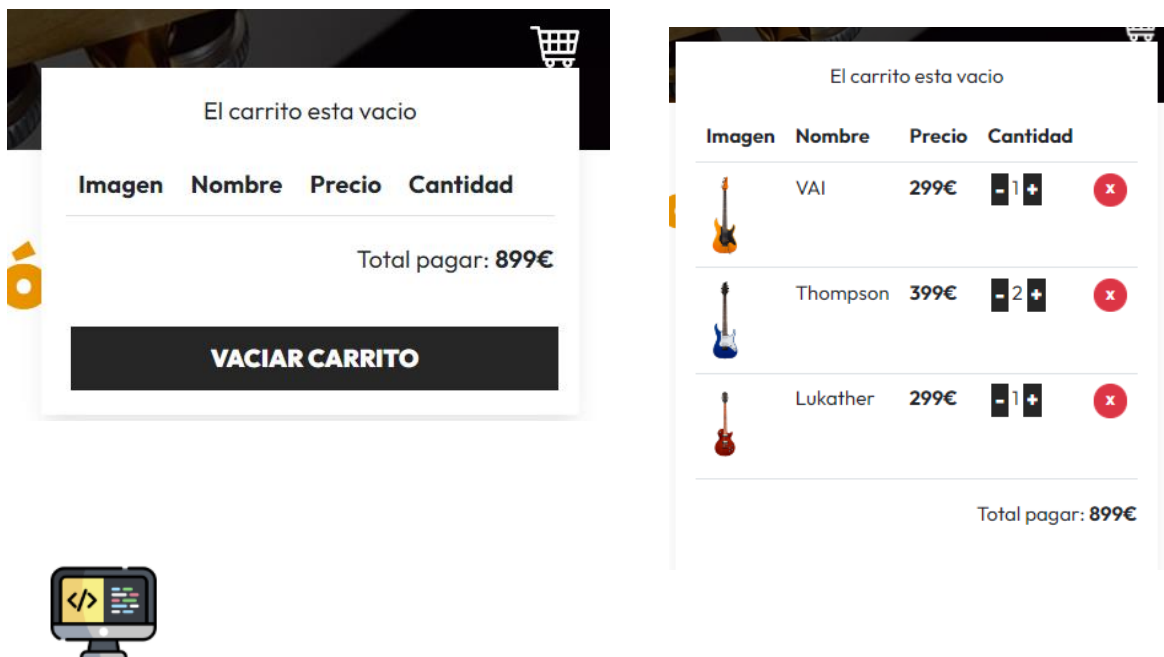


## 14. STATE DERIVADO

Se trata de una **variable** cuyo valor está estrechamente relacionado con un **State** o depende de él.

Se crea fuera del Template para separar la lógica de la vista.

Volviendo al carrito, vemos que si está vacío, se muestra el mensaje "El carrito está vacío" y, cuando está lleno, sigue mostrándose → NO tiene sentido.



Lo que queremos es que, cuando en el carrito haya guitarras, el mensaje NO se muestre.

Podríamos crear un State para ver si el carrito está vacío o no, pero no lo vamos a hacer porque debemos evitar crear una alta cantidad de States.

Vamos a ver dos opciones:

La **primera** opción es añadir código directament. En el componente Header, vamos a utilizar en la parte del template una ternaria(al ser un Expression y no un Statement, lo podemos utilizar después del return).

Quedaría así; Si no hay artículos muestra el párrafo y si los hay incluimos todo lo que había después en la etiqueta `<table>` e incluimos el total a pagar y el importe. De esta forma si no hay guitarras en el carrito, tampoco se mostrará el total a pagar.

Nos dará error porque no podemos devolver varios elementos de primer nivel: Recurrimos a un **fragment**

```
{carrito.length === 0 ? (
  <p className="text-center">El carrito esta vacio</p>
) : (
  <>
    <table className="w-100 table">
      <thead>
```

La segundo opción es utilizar un **State derivado**.

Creamos la variable en nuestro componente. Por ejemplo creamos una variable y la igualamos a una arrow function que corresponda al ternario anterior.

```
export default function Header({carrito}){

  //State derivado
  const carritoVacio = () => carrito.length === 0

  {carritoVacio() ? (
    <p className="text-center">El carrito esta vacio</p>
  ) : (
    <>
      <table className="w-100 table">
```



Vamos a utilizar otro **State derivado** para dinamizar el 'Total a pagar' del carrito de compras.

Para ello podemos utilizar una constante que la igualamos a una arrow function que irá recorriendo el carrito y calculando el total (podemos usar **reduce**, un método de los array)

```
const carritoTotal = () => carrito.reduce((total, element) => total + (element.cantidad * element.price), 0)
```

Sustituimos la llamada a la función por el importe total del carrito.



## 15. HOOK USEMEMO

Estos hooks sirven para simplificar los Templates y además está enfocado al performance porque evita que el código se ejecute si alguna de las dependencias que definiremos en este hook no ha cambiado.

Para entender este hook debemos saber que React renderiza todo cada vez que haya una interacción, es decir, cada State es renderizado al completo cada vez que, simplemente se clique en una parte. Esto puede hacer que se vea lenta la carga.

Gracias a este Hook, podremos realizar un cálculo concreto que depende de un State, es decir, le indicamos las dependencias para evitar renders completos.



Lo primero que debemos hacer es importar **useMemo**

Vamos a utilizarlo en el State Derivado de antes:

```
const carritoVacio = useMemo (() => carrito.length === 0, [carrito])
```

Vemos un segundo parámetro que es el array de dependencias.

Como cambio en la sintaxis, cuando queramos invocarlo, eliminamos los paréntesis.

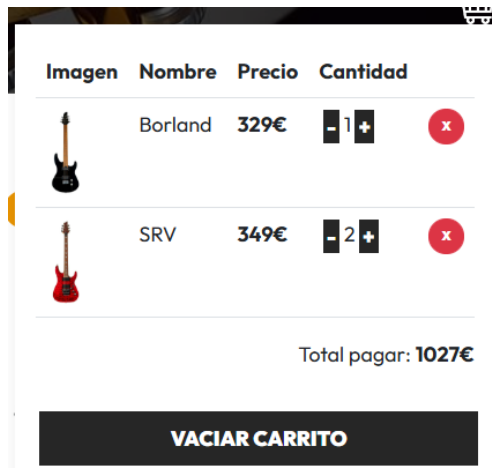
```
{carritoVacio ? (
  |      <p className="text-center"
) : (
```

Hacemos lo mismo con el State Derivado que calcula el total del carrito.

## 16. PRACTICANDO LO APRENDIDO



Vamos a trabajar con el botón de **eliminar elementos del carrito**.



Cada vez que el usuario de click sobre el círculo rojo con el aspa, se deberá borrar la fila del carrito.

Si nos damos cuenta tenemos el State en **App.jsx** y el código del carrito en **Header.jsx**.

En estos casos es mejor hacer la función en el componente padre y pasársela al hijo a través de Props.

Recordamos pasos:

1. Creamos función

```
function eliminarDelCarrito(id){  
  const nuevoCarrito = () => carrito.filter(element => element.id !== id)  
  setCarrito(nuevoCarrito)  
}
```

2. Pasamos función al componente hijo a través de props

```
<Header  
  carrito = {carrito}  
  eliminarDelCarrito = {eliminarDelCarrito}  
>
```

3. Recogemos la función en el componente hijo.

```
export default function Header({carrito, eliminarDelCarrito}){
```

4. Relacionamos el elemento del componente con la función (a través de un evento). Recordemos que si hay que pasar parámetros, debemos usar una arrow function.

```

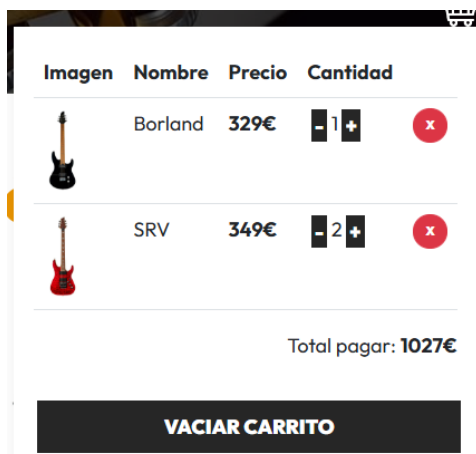
<td>
  <button
    className="btn btn-danger"
    type="button"
    onClick={() => eliminarDelCarrito(element.id)}
  >
    X
  </button>
</td>

```



Ya podemos añadir guitarras al carrito, eliminarlas y el total se autocalcula.

Ahora vamos a poder **aumentar o disminuir cantidades** directamente desde el carrito.



Podemos ver a la derecha del precio, unos botones con un signo de menos y más.

Esto será lo que utilizaremos para aumentar/disminuir cantidades.

Vamos a proceder como en el caso anterior (seguiremos los mismos pasos). Debemos poder incrementar cantidades y decrementarlas. En este último caso, si la cantidad es 1, ya no podremos decrementar más. Si queremos eliminar la guitarra, clicamos sobre el botón rojo circular.

0



Ahora vamos a **vaciar completamente el carrito** desde el botón 'VACIAR CARRITO'

Vamos a proceder como en el caso anterior (seguiremos los mismos pasos).

## 17. LOCALSTORAGE

Vamos a hacer nuestro carrito persistente de forma que si abandono la página y había incluido algunas guitarras, cuando vuelva, las encuentre.

Debemos saber que el **State en React es asíncrono**, esto nos indica que cuando vayamos a guardar los datos de nuestro State en localStorage puede que se ejecute ese almacenamiento antes de que nuestro State haya acabado de setearse.



Se puede hacer la prueba (no es necesario) generando una función que almacene en localStorage cada vez que se añada al carrito y ver su comportamiento: la primera vez que se añade, se guarda un array vacío (pero en el carrito ya hay guitarras)

```
function almacenarEnlocalStorage(){  
  localStorage.setItem('carrito', JSON.stringify(carrito));  
}
```

Imagen	Nombre	Precio	Cantidad
	SRV	349€	- 1 +

Total pagar: 349€

VACIAR CARRITO

Storage	Origin	Key	Value
Local ...	http://localhost:5173	carrito	[]

Esta asincronía permite la renderización a una velocidad adecuada para garantizar el éxito de la experiencia de usuario pero como desarrolladores, debemos tenerlo en cuenta para almacenar los datos con precisión.

### 17.1. Hook useEffect

Este Hook es muy útil para manejar los efectos secundarios de un cambio en nuestro State.

**Se ejecuta cuando el componente esté listo y cuando el State que pongamos como dependencia cambie.**

Lo tenemos que importar.

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



Vamos a quitar nuestra función y su llamada al añadir al carrito.

Añadimos, en la parte de la lógica del componente 'App.jsx', el hook. Su sintaxis será con una arrow function y un array de dependencias.

Cada vez que el State carrito cambie, se ejecutará lo que hay en la función. De este modo evitamos la creación de la función, sus múltiples llamadas y queda controlada la asincronía del State.

```
//useEffect
useEffect(()=>{
  localStorage.setItem('carrito', JSON.stringify(carrito))
}, [carrito])
```

## 17.2. Recuperando datos de localStorage

Si nos damos cuenta, al inicio del componente 'App.jsx' tenemos los dos State: los datos y el carrito.

En el caso del carrito inicia sin datos. Esto provocará que, aunque useEffect haya guardado en localStorage guitarras del carrito, cada vez que reiniciemos la página, localStorage se modifique con [] (valor inicial del State).



Tenemos que valorar si hay algo en localStorage (y pasarlo al State como valor inicial)0 y si no lo hay, el State de Carrito iniciará con [].

Para ello, en la parte de lógica de 'App.jsx', creamos una función anónima que recoja lo que hay en localStorage y, si no hay nada, devolverá []

```
function App() {

  //Valores iniciales de State
  const carritoInicial = () =>{
    const localStorageCarrito = localStorage.getItem('carrito')
    return (localStorageCarrito !== null) ? JSON.parse(localStorageCarrito) : []
  }

  //State
  const [data, setData] = useState(db)
  const [carrito, setCarrito] = useState(carritoInicial)

  //useEffect
```

## 18. ULTIMOS AJUSTES

Si vemos la consola, podemos ver este mensaje:

```
Files in the public directory are served at the root path.
Instead of /public/img/header.jpg, use /img/header.jpg.
Files in the public directory are served at the root path.
Instead of /public/img/logo.svg, use /img/logo.svg.
Files in the public directory are served at the root path.
Instead of /public/img/carrito.png, use /img/carrito.png.
```

Debemos ir imagen por **imagen y modificar su url**.

Estas url con 'public' están porque este proyecto se inició con código HTML y se ha ido migrando a React.

Tendremos que modificar **'Header.jsx'** e **'index.css'**. En este último fichero se elimina la fila que se indica:

```
758  /** TODO: Eliminar la sig línea */
759  background-image: linear-gradient(to right, rgba(0, 0, 0, 0.7), rgba(0, 0, 0, 0.7)), url("../public/img/header.jpg");
```

Por otro lado, vamos a eliminar aquello que hemos creado y no hemos utilizado.

App.jsx → podemos eliminar el setter de data:

```
const [data, setData] = useState(db)
```

## 19. DESPLIEGUE PROYECTO



Estos proyectos (Angular, Vue o React) se tienen que construir.

Si abrimos el fichero 'package.json', podemos ver el comando:

**"build": "vite build",** , que se encargará de hacer mejoras de performance, de subir una versión ligera.

Lo que debemos hacer es abrir un nuevo terminal y escribir el comando para construirlo:

```
PS E:\CLASE\DAW\DWEC\24-25\BLOQUE3_FRAMEWORKS\REACT\PROYECTOS\1. FUNDAMENTOS Y TYPESCRIPT\01_guitarra\tiendaGuitarra> npm run build

> tiendaguitarra@0.0.0 build
> vite build

vite v5.4.2 building for production...
transforming (17) src\components\Guitarra.jsx
../img/header.jpg referenced in ../img/header.jpg didn't resolve at build time, it will remain unchanged to be resolved at runtime
✓ 34 modules transformed.
dist/index.html          0.72 kB | gzip: 0.40 kB
dist/assets/index-D5aM7PDf.css 12.84 kB | gzip: 3.67 kB
dist/assets/index-DGKmos11.js 148.33 kB | gzip: 47.32 kB
✓ built in 1.01s
```

Nos da un resumen de lo que ha hecho: Ha creado varios archivos dentro de la carpeta '**dist**'.

Un detalle que podemos ver es que '**header**' hace referencia a otro lugar. Vamos a solucionarlo.

Buscamos la imagen en el fichero '**index.css**' y le quitamos los dos puntos de la ruta relativa.

Vamos a volver a lanzar el comando **npm run build**

```
PS E:\CLASE\DAW\DWEC\24-25\BLOQUE3_FRAMEWORKS\REACT\PROYECTOS\1. FUNDAMENTOS Y TYPESCRIPT\01_guitarra\tiendaGuitarra> npm run build

> tiendaguitarra@0.0.0 build
> vite build

vite v5.4.2 building for production...
✓ 34 modules transformed.
dist/index.html          0.72 kB | gzip: 0.41 kB
dist/assets/index-BHUzy7s-.css 12.84 kB | gzip: 3.67 kB
dist/assets/index-CjHALP2L.js 148.33 kB | gzip: 47.32 kB
✓ built in 924ms
```

Ya no hay advertencias y de nuevo tenemos el resumen.

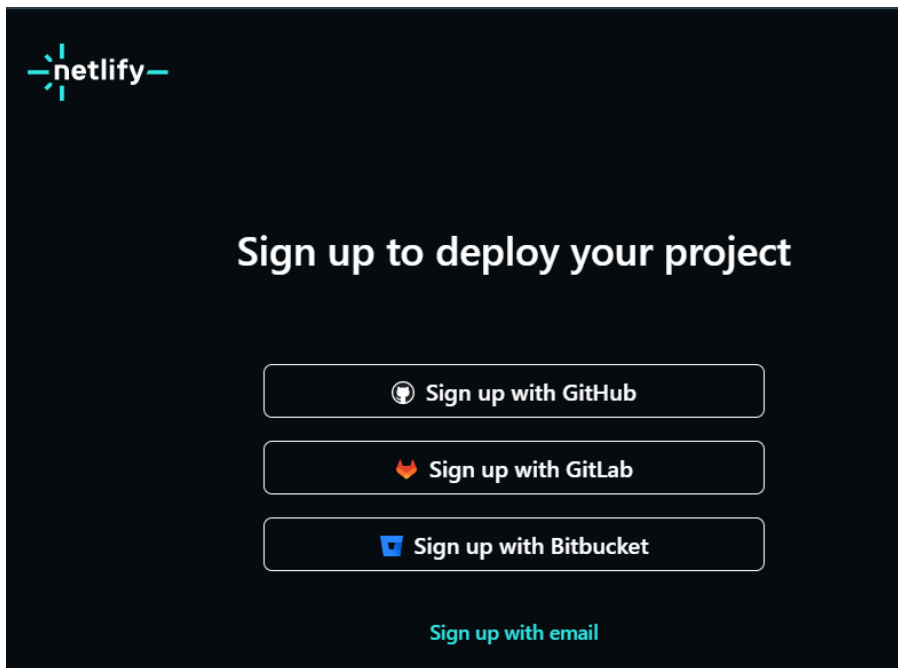
Ha transformado 34 módulos. Esto incluye componentes, archivos y dependencias que vamos importando. De todo esto se encarga **Vite**.

Vemos que **genera el archivo de index.html, el de CSS y uno de JS que contiene una versión de React.**

El archivo index.html tiene que correr en un hosting que lo soporte y una buena opción gratuita es [Netlify](#).

Netlify nos permite subir los proyectos en HTML, CSS, JS, React, Vue.js

Como no tenemos cuenta, la vamos a crear (Sign up)

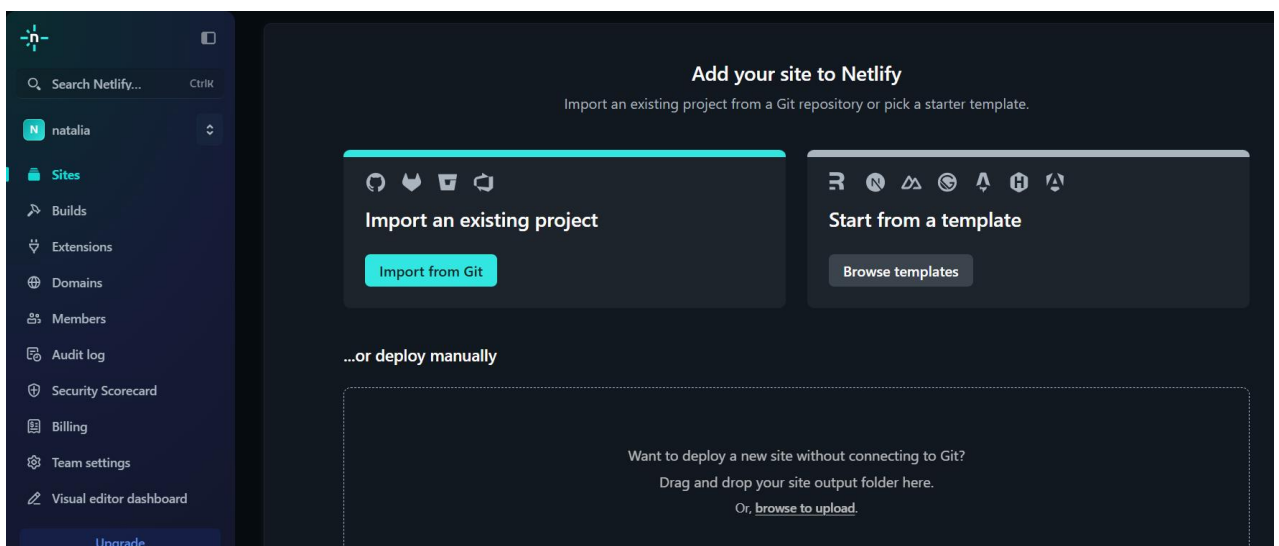


Si tu cuenta de GitHub es muy nueva puede que de problemas, no permite subir los proyectos para evitar spam. Nos registramos a través de un mail.

Indicamos mail, contraseña y la confirmamos.

Después de contestar a algunas preguntas, ya podemos seguir con el despliegue.

Una vez estamos logueados, vamos a nuestro panel, en **Sites**



Arrastramos nuestra carpeta **dist** y cliclamos sobre **Open production deploy**

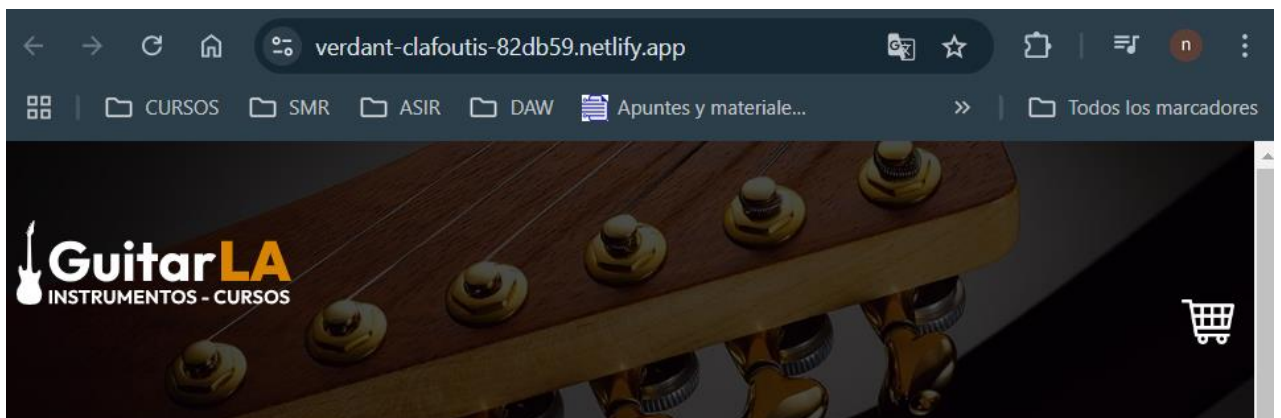
## Published deploy for verdant-clafoutis-82db59

Today at 11:18 PM

Production [Download](#)

[Open production deploy](#)[Lock to stop auto publishing](#)[Options](#)

Podemos ver nuestro proyecto publicado en Internet.



## Nuestra Colección

### LUKATHER



Morbi ornare augue nisl, vel  
elementum dui mollis vel.  
Curabitur non ex id eros  
fermentum hendrerit.

**299€**

### SRV



Morbi ornare augue nisl, vel  
elementum dui mollis vel.  
Curabitur non ex id eros  
fermentum hendrerit.

**349€**

## 20. CREAR HOOKS PROPIOS