Hook useEffect

Actualmente, se están obteniendo los datos de la factura mediante la función **getInvoice** de forma local, leyendo un archivo que contiene todos los datos. Sin embargo, en otro contexto, estos datos podrían provenir de forma remota desde una API de backend.

La carga y el retraso de conectarse a otro servidor pueden ser significativos. En la aplicación web, no se está manejando ningún ciclo de vida del componente. Es por ello que el objetivo de esta sección es que los datos se carguen una sola vez durante todo el ciclo de vida del componente.

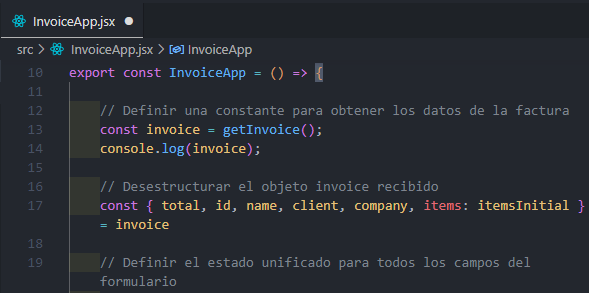
Pasos 85 al 91

Sigue estos pasos:

Paso 85: El problema actual de la función getInvoice

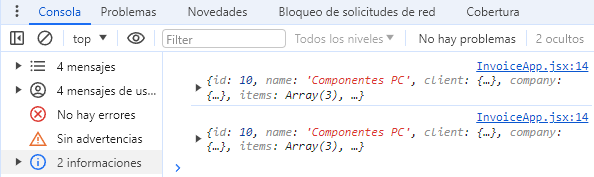
Actualmente, la función **getInvoice** se invoca cada vez que el componente se renderiza o redibuja, lo cual ocurre cada vez que se cambia el estado, como al modificar el contador o agregar un nuevo ítem mediante el formulario. Esto provoca múltiples invocaciones a **getInvoice**, lo cual no es eficiente, especialmente si la función realiza una solicitud a un servidor remoto.

Para aclarar el problema, procede a definir una constante llamada **invoice** que almacenará el objeto **invoice** obtenido de la función **getInvoice**. Luego, imprime la constante con **console.log** para observar cuántas veces se invoca. También desestructura los datos de la constante **invoice**.

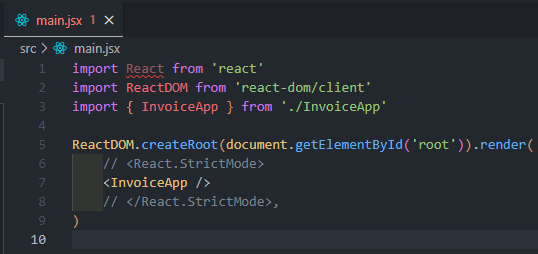


Paso 86: Visualizar el problema de ciclo de vida del componente

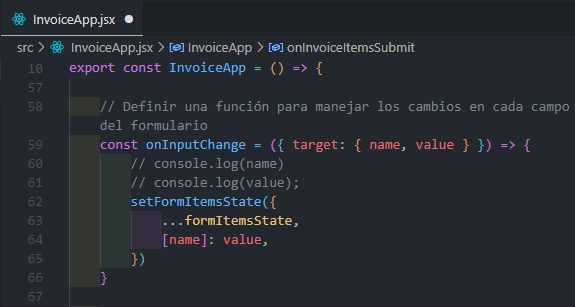
Guarda los cambios y en el navegador se puede ver que el objeto **invoice** se imprime dos veces al recargar la página. Esto se debe a que la función **getInvoice** se invoca cada vez que el componente se renderiza, lo cual no es eficiente.

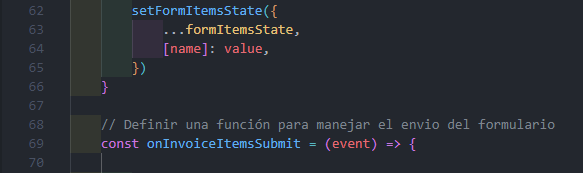


Para mejorar la visibilidad del problema, deshabilita temporalmente el **React.StrictMode** en **main.jsx**, esto hará que **invoice** se imprima solo una vez.

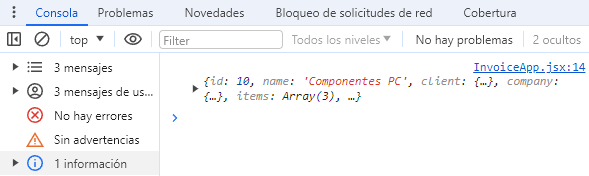


También comenta el **console.log** que se encuentra en la función **onInputChange**.



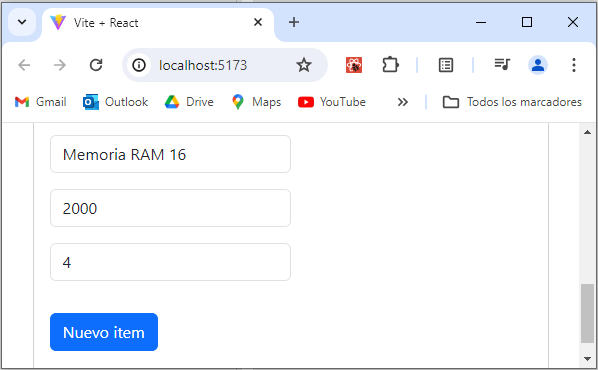


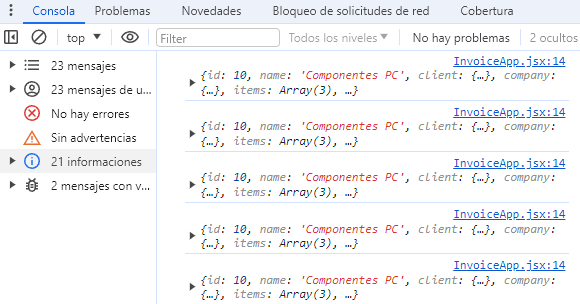
Guarda los cambios y el objeto **invoice** se imprimira solo una vez en la consola.



Al hacer clic en un cuadro de texto y modificar el formulario, el estado **formItemsState** se actualiza a través de **onInputChange**. Esto provoca que el componente se redibuje, lo que a su vez invoca **getInvoice** nuevamente. Esto no es deseable, ya que **getInvoice** debería ejecutarse sólo una vez cuando el componente se carga en el navegador.

En el navegador, al seleccionar un campo de texto y modificar el contenido (**product**, **price** o **quantity**), se observa que en la consola se hacen varias llamadas a **getInvoice**. Luego, al guardar, se hace otra llamada adicional. Esto es ineficiente y podría impactar negativamente el rendimiento, especialmente si **getInvoice** obtiene datos de forma remota desde un backend.





Cada vez que se actualiza el estado del formulario, los ítems o el contador (**counter**), el componente se redibuja. Este redibujo afecta al rendimiento de la aplicación. Es crucial optimizar este proceso, especialmente si **getInvoice** implica una solicitud a un backend, lo cual es común en aplicaciones reales donde se interactúa con servicios como Node.js, Spring o Python.

Paso 87: Definir el estado inicial de invoice

El objetivo es cargar los datos de la factura una sola vez durante todo el ciclo de vida del componente, independientemente de si se modifica el estado, se agregan nuevos ítems o se cambia el contador. Esto se logrará utilizando el hook **useEffect**.

Para manejar la factura con el estado de React, define un nuevo estado **useState** para establecer los valores iniciales de la factura (se establece la constante **invoice**, por lo cual se elimina la constante definida anteriormente), el cual se basa en la estructura del JSON del objeto **invoice.js** pero con datos vacíos (no nulos); en el caso de **items** se establece un arreglo vacío.



Paso 88: Optimizar el Ciclo de Vida del Componente con useEffect

El hook **useEffect** permite ejecutar efectos secundarios en componentes funcionales de React. Se utiliza para manejar el ciclo de vida del componente, como la ejecución de código después del renderizado inicial. Se utiliza junto con **useState** para manejar el estado y guardar datos.

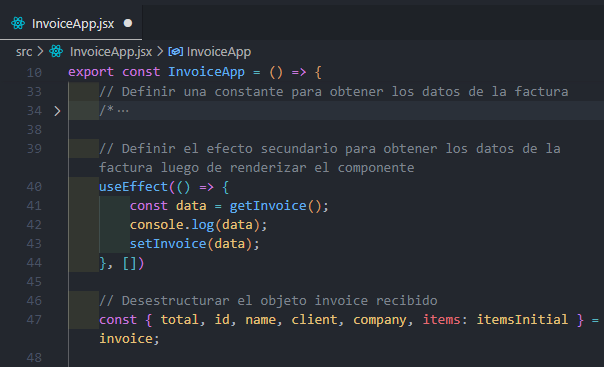
Este hook se utiliza pasando una función de flecha como primer argumento. Se define un segundo argumento que es un arreglo de dependencias. Cuando este arreglo está vacío, el efecto se ejecuta solo una vez después del renderizado inicial.

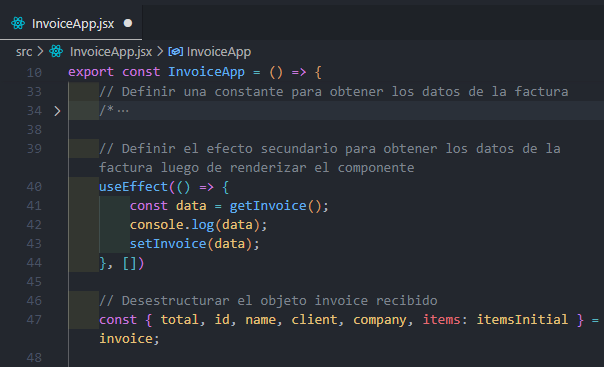
Para implementarlo, procede a importar el hook **useEffect** de React. Luego, en la función de flecha (primer argumento), define el llamado a la función **getInvoice** mediante una constante **data** para que se ejecute solo una vez cuando se renderiza el componente (el segundo argumento se deja como un arreglo vacío).

El objeto **invoice** obtenido en la constante **data** se guarda en el estado **invoice** usando **setInvoice**. Este enfoque garantiza que **getInvoice** se llame solo una vez, mejorando la eficiencia de la aplicación.

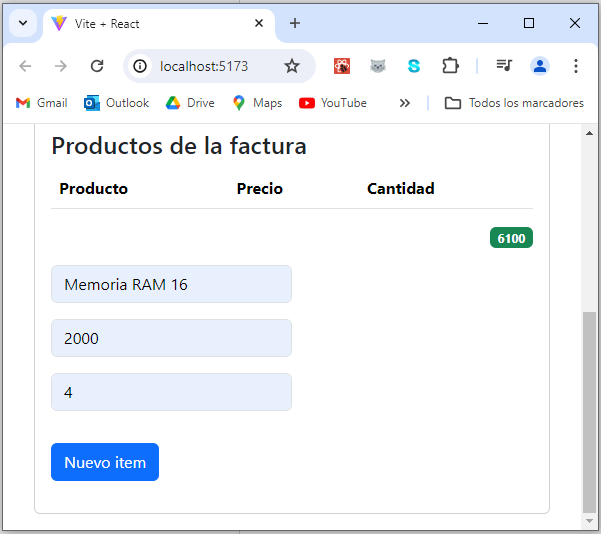
Coloca un **console.log(invoice)** dentro de **useEffect** para verificar que los datos se impriman solo una vez. Al desestructurar los datos de la factura (estado inicial de **invoice**) fuera de **useEffect**, se asegura que los datos estén disponibles para su uso en el componente.



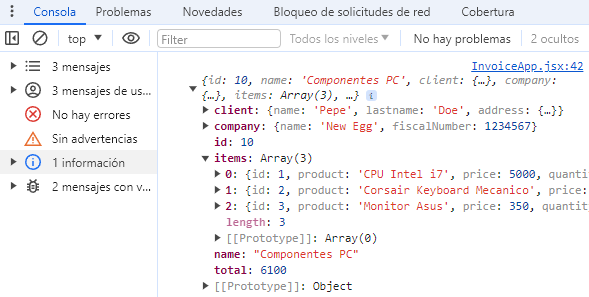




Ejecuta la aplicación y lo primero que se puede notar es que no existen productos por defecto en la tabla.



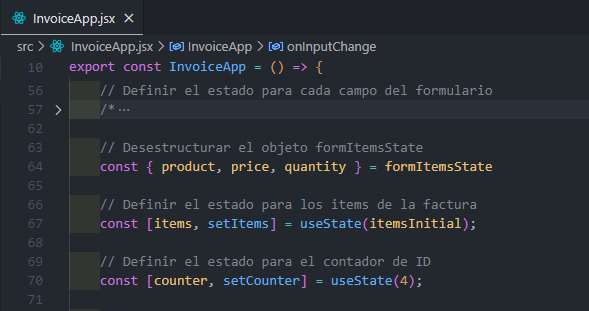
La factura (obtenida desde la constante **data**) se imprime solo una vez en la consola, en contraste con múltiples impresiones antes de la optimización. Prueba introduciendo datos en el formulario y la consola mantiene impresa esa factura; aun si se agrega un nuevo ítem desde el formulario, la **data** se mantiene intacta.

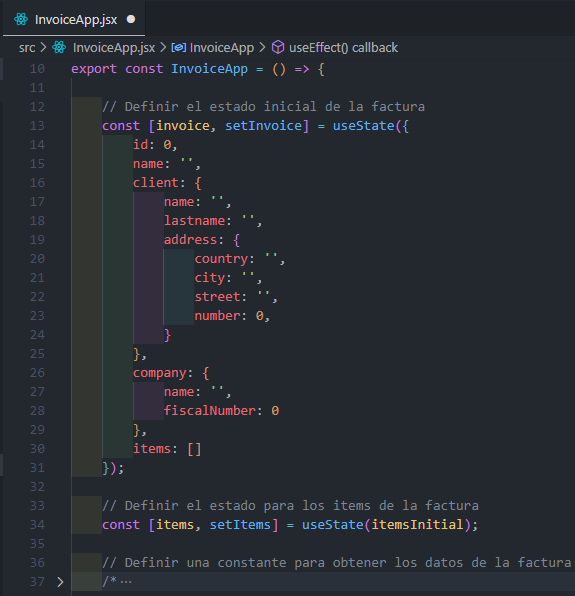


Paso 89: Trasladar el estado de items a un nivel superior

Al intentar mostrar los elementos de la factura en la interfaz de usuario, se encuentra que estos elementos no se renderizan inicialmente. Esto se debe a que en el estado inicial de **items** (definido con **itemsInitial**) se inicializa como una lista vacía (toma el atributo **items** del objeto inicial en el estado de **invoice**). Para resolver este problema y establecer el estado de los elementos de la factura, realiza lo siguiente:

Cambia la posición del estado de los **items** a un nivel superior, debajo del estado inicial de la factura.



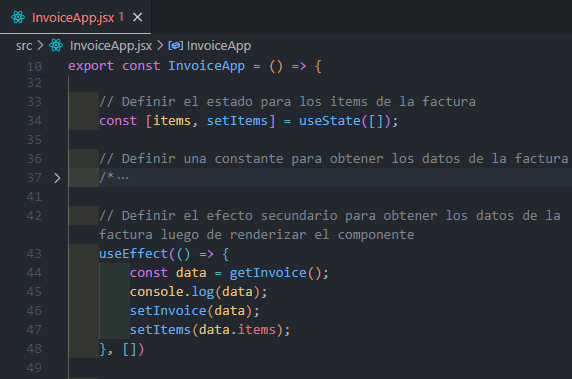


Luego, dentro de la función del hook **useEffect**, debajo del llamado de la función **setInvoice**, se procede a actualizar el estado de **items** con **setItems**, pasando solamente el arreglo de **items** del objeto **invoice** definido con la constante **data**.



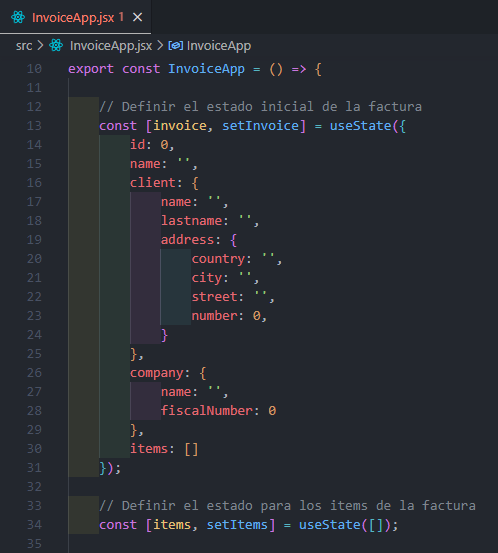


Con el **useEffect**, se pasa el estado de la factura completo y se modifica el estado de **items** porque los ítems se manejan aparte con su propio estado. Asimismo, se puede ver que **itemsInitial** aún no se define porque la definición de esa constante está en un nivel inferior. La solución es reemplazar el estado inicial de **items** por unos corchetes vacíos.



Paso 90: Separar los valores iniciales del estado de la factura

Para que el código quede más limpio, procede a cortar el contenido del estado inicial de **invoice** y pégalo fuera de la función principal del componente. Define una nueva constante llamada **invoiceInitial** que va a contener los valores iniciales del estado **invoice**. Asimismo, llama a la constante **invoiceInitial** en el estado de **invoice** para establecer los valores iniciales.





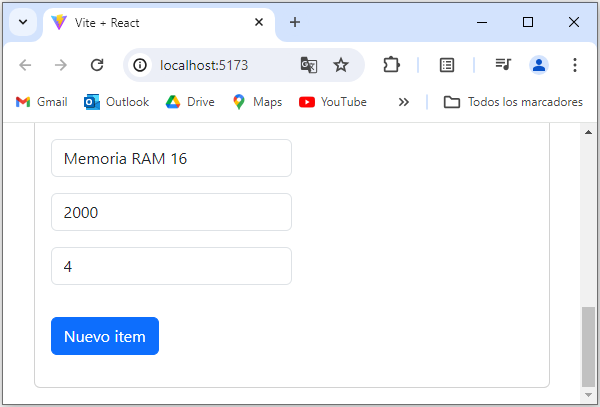


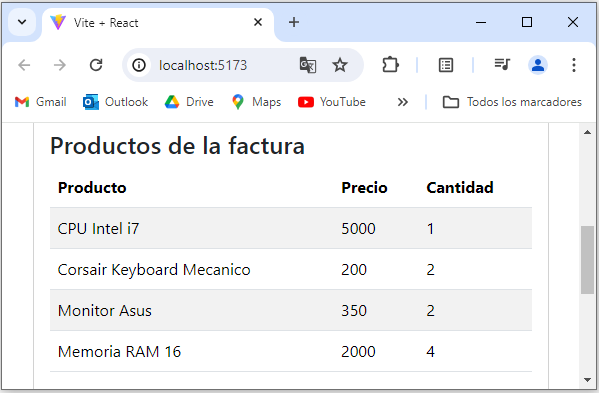
Paso 91: Realizar una prueba en el navegador

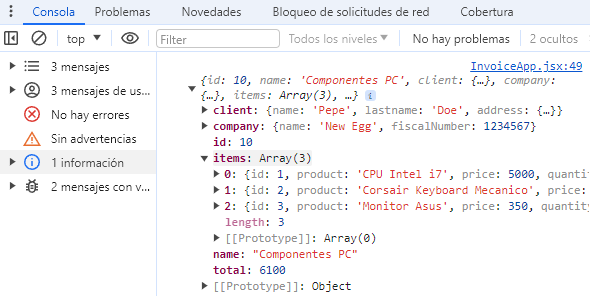
En el navegador, se puede visualizar que se tiene la tabla con los datos de los ítems tal y como estaba antes. Cada vez que se actualiza el estado al escribir texto en el campo del formulario, en la consola no se imprime seguidamente el objeto **invoice** porque solamente con el hook **useEffect** lo imprime una sola vez a pesar de que cambia el estado con los cuadros de texto, el contador, etc.

Así, se puede agregar otros productos, manipular el estado agregando un nuevo producto, llenando campos del formulario y nunca más irá por segunda vez al backend, solamente la primera vez cuando se carga la página o se actualiza la página cuando se construye el componente.

La aplicación queda más optimizada en términos de rendimiento.







Ten en cuenta que cuando se conecta a una API remota, es importante optimizar para evitar retrasos y costos en el rendimiento. Usar **useEffect** garantiza que la solicitud de datos solo se realice una vez al montar el componente. **useEffect** actúa como un evento que se dispara cuando el componente se crea o cuando cambian las dependencias.

En la aplicación web, se obtiene la factura y se guarda en el estado usando **useState**, lo mismo para los ítems de la factura, que se manejan en un estado paralelo. La desestructuración del objeto **invoice** se maneja con **useState**, cambiando la forma de obtener los datos.