1. DISEÑO DE INTERFACES CON REACT

React puede cambiar tu forma de pensar en los diseños que miras y las aplicaciones que construyes. Cuando construyes una interfaz de usuario (UI) con React, primero la separarás en piezas denominadas componentes. Luego, describirás los diferentes estados visuales para cada uno de tus componentes. Para finalizar, conectarás tus componentes de forma tal que los datos fluyan por ellos.

2. BOCETO

Tenemos una API JSON y un boceto de un diseñador.

La API JSON devuelve algunos datos como estos:

```
[
  { category: "Fruits", price: "$1", stocked: true, name: "Apple" },
  { category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit" },
  { category: "Fruits", price: "$2", stocked: false, name: "Passionfruit" },
  { category: "Vegetables", price: "$2", stocked: true, name: "Spinach" },
  { category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin" },
  { category: "Vegetables", price: "$1", stocked: true, name: "Peas" }
]
```

El boceto luce así:

3. Separa la UI en una jerarquía de componentes

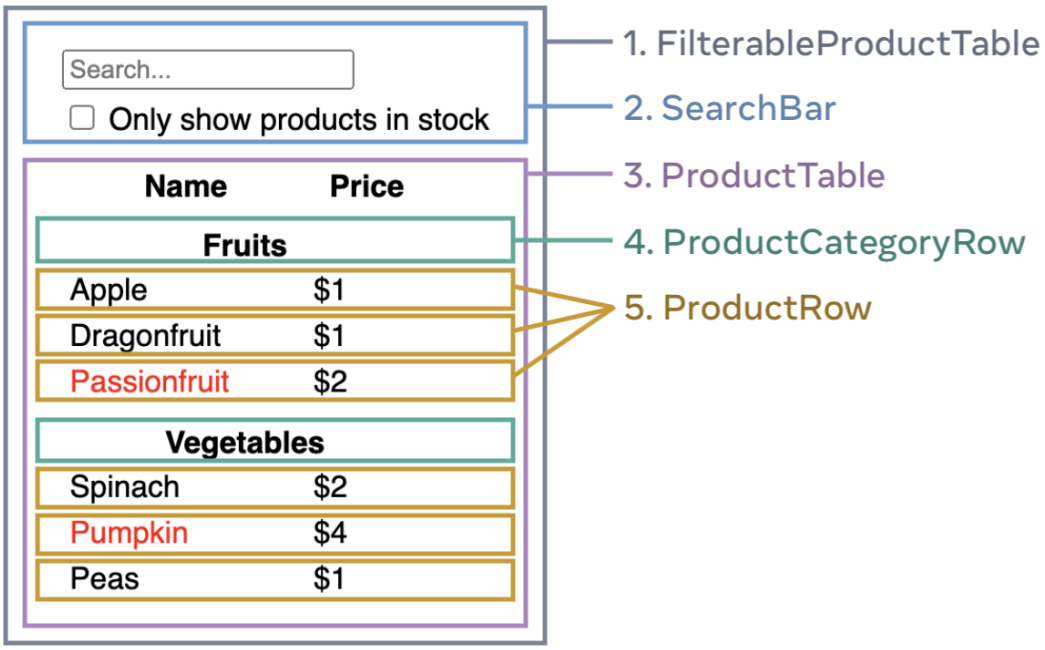
Comienza por dibujar cuadros alrededor de cada componente y subcomponente en el boceto y nómbralos. Si trabajas con un diseñador puede que ya les haya dado nombres a estos componentes en su herramienta de diseño.

En dependencia de tu formación y experiencia, puedes pensar sobre la separación de un diseño en componentes de diferentes maneras:

- Programación—usa las mismas técnicas para decidir si debes crear una nueva función u objeto. Una de esas técnicas es el principio de responsabilidad única, o sea, un componente debería hacer idealmente solo una cosa. Si termina creciendo, debería ser descompuesto en subcomponentes más pequeños.
- CSS—considera para qué cosas hubieras creado selectores de clase. (Sin embargo, los componentes son un poco menos granulares).
- Diseño—considera cómo organizarías las capas del diseño.

Si tu JSON está bien estructurado, a menudo encontrarás que se corresponde naturalmente con la estructura de componentes de tu UI. Esto ocurre porque la UI y los modelos de datos a menudo tienen la misma arquitectura de información—o sea, la misma forma. Separa tu UI en componentes, de manera que cada componente se corresponda con una pieza de tu modelo de datos.

Hay cinco componentes en esta pantalla:



- FilterableProductTable (gris) contiene toda la aplicación.
- SearchBar (azul) recibe la entrada del usuario.
- ProductTable (lavanda) muestra y filtra la lista de acuerdo a la entrada del usuario.
- ProductCategoryRow (verde) muestra un encabezado para cada categoría.
- ProductRow (amarillo) muestra una fila para cada producto.

Si mirás a ProductTable (lavanda), verás que el encabezado de la tabla (que contiene las etiquetas «Name» y «Price») no es un componente independiente. Esto es una cuestión de preferencias, y podrías hacerlo de ambas formas. Para este ejemplo, es parte de ProductTable porque aparece dentro de la lista de ProductTable. Sin embargo, si este encabezado crece y se vuelve complejo (por ejemplo, si añades ordenamiento), tendría sentido convertirlo en un componente independiente ProductTableHeader.

Ahora que identificaste los componentes en el boceto, ordenalos en una jerarquía:

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow

4. Construye una versión estática en React

Ahora que tenemos la jerarquía de componentes, es momento de implementar la aplicación. El enfoque más sencillo consiste en construir una versión que renderiza la UI a partir de tu modelo de datos sin añadir ninguna interactividad. Suele ser más fácil construir primero la versión estática y luego añadir la interactividad de forma independiente. Construir una versión estática requiere mucha escritura y poco pensamiento, pero añadir interactividad requiere mucho pensamiento y no mucha escritura.

Para construir la versión estática de tu aplicación que renderiza tu modelo de datos querrás construir componentes que reutilicen otros componentes y pasen datos usando props. Las props son una forma de pasar datos de padres a hijos.

Puedes o bien construir de «arriba a abajo» comenzando por construir los componentes más arriba en la jerarquía (como FilterableProductTable) or de «abajo a arriba» trabajando con los componentes más abajo (como ProductRow). En ejemplos más simples, usualmente es más fácil ir de arriba a abajo, y en proyectos más grandes, es más fácil ir de abajo a arriba.

```
function ProductCategoryRow({ category }) {
  return (
    <tr>
      <th colSpan="2">
        {category}
      </th>
    </tr>
  );
}

function ProductRow({ product }) {
  const name = product.stocked ? product.name :
    <span style={{ color: 'red' }}>
      {product.name}
```

```
    </span>;

  return (
    <tr>
      <td>{name}</td>
      <td>{product.price}</td>
    </tr>
  );
}

function ProductTable({ products }) {
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (product.category !== lastCategory) {
      rows.push(
        <ProductCategoryRow
          category={product.category}
          key={product.category} />
      );
    }
```

Después de construir tus componentes, tendrás una biblioteca de componentes reutilizables que renderizan tu modelo de datos. Dado que esta es una aplicación estática, los componentes solo devuelven JSX. El componente en la cima de la jerarquía (FilterableProductTable) tomará tu modelo de datos como una prop. Este se conoce como flujo de datos en un sentido, porque estos datos fluyen hacia abajo desde el componente en el nivel superior hacia aquellos que están al final del árbol.

5. Encuentra la representación mínima pero completa del estado de la UI

Para hacer la UI interactiva, necesitas dejar que los usuarios cambien tu modelo de datos. Para esto utilizarás estado.

Piensa en el estado como el conjunto mínimo de datos cambiantes que la aplicación necesita recordar. El principio más importante para estructurar datos es mantenerlos DRY (Don't Repeat Yourself o No te repitas). Encuentra la representación absolutamente mínima del estado que tu aplicación necesita y calcula lo demás bajo demanda. Por ejemplo, si estás construyendo una lista de compra, puedes almacenar los elementos en un arreglo en el estado. Si también quieres mostrar el número de elementos en la lista, no almacenes el número de elementos como otro valor de estado—en cambio, lee el tamaño de tu arreglo.

Ahora pensá en todas las piezas de datos en esta aplicación de ejemplo:

- La lista original de productos
- El texto de búsqueda que el usuario ha escrito
- El valor del checkbox
- La lista de productos filtrada

¿Cuáles de estos son estado? IdentificÁ los que no lo son:

- ¿Se mantiene sin cambios con el tiempo? Si es así, no es estado.
- ¿Se pasa desde un padre por props? Si es así, no es estado.
- ¿Puedes calcularlo basado en estado existente on en props en tu componente? Si es así, ¡definitivamente no es estado!

Lo que queda probablemente es estado.

Veámoslos uno por uno nuevamente:

- La lista original de productos se pasa como props, por lo que no es estado.
- El texto de búsqueda parece ser estado dado que cambia con el tiempo y no puede ser calculado a partir de algo más.
- El valor del checkbox parece ser estado porque cambia con el tiempo y no puede ser calculado a partir de algo más.
- La lista filtrada de productos no es estado porque puede ser calculada tomando la lista original de productos y filtrándola de acuerdo al texto de búsqueda y el valor del checkbox.

6. Identificar dónde debe vivir tu estado

Después de identificar los datos mínimos de estado de tu aplicación, debes identificar qué componente es responsable de cambiar este estado, o posee*

el estado. Recuerda: React utiliza un flujo de datos en una sola dirección, pasando datos hacia abajo de la jerarquía de componentes desde el componente padre al hijo. Puede no ser inmediatamente claro qué componente debe poseer qué estado. Esto puede suponer un resto si este concepto es nuevo para ti, pero puedes lograrlo si sigues los siguientes pasos.

Por cada pieza de estado en tu aplicación:

- Identifica cada componente que renderiza algo basado en ese estado.
- Encuentra su componente ancestro común más cercano—un componente que esté encima de todos en la jerarquía
- Decide dónde debe residir el estado:
    - A menudo, puedes poner el estado directamente en su ancestro común.
    - También puedes poner el estado en algún componente encima de su ancestro común.
    - Si no puedes encontrar un componente donde tiene sentido poseer el estado, crea un nuevo componente solo para almacenar ese estado y añádelo en algún lugar de la jerarquía encima del componente ancestro común.

En el paso anterior, encontraste dos elementos de estado en esta aplicación: el texto de la barra de búsqueda, y el valor del checkbox. En este ejemplo, siempre aparecen juntos, por lo que es más fácil pensar en ellos como un solo elemento de estado.

Ahora utilicemos nuestra estrategia para este estado:

Identifica componentes que usen estado:

- ProductTable necesita filtrar la lista de productos con base en ese estado (texto de búsqueda y valor del checkbox).
- SearchBar necesita mostrar ese estado (texto de búsqueda y valor del checkbox).

Encuentra su ancestro común:

- El primer componente ancestro que ambos componentes comparten es FilterableProductTable.

Decide donde reside el estado:

- Mantendremos el texto de filtrado y el estado de valor seleccionado en FilterableProductTable.

Por tanto, los valores del estado residirán en FilterableProductTable.

Añade estado al componente con el Hook useState(). Los Hooks te permiten «engancharte» al ciclo de renderizado de un componente Añade dos variables de estado al inicio de FilterableProductTable y especifica el estado inicial de tu aplicación:

```
function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);
```

Pasa entonces filterText e inStockOnly a ProductTable y SearchBar como props:

```
<div>
  <SearchBar
    filterText={filterText}
    inStockOnly={inStockOnly} />
  <ProductTable
    products={products}
    filterText={filterText}
    inStockOnly={inStockOnly} />
</div>
```

Puedes comenzar a ver como tu aplicación se comportará. Edita el valor inicial de filterText de useState('') a useState('fruit') en el ejemplo de código debajo. Verás que tanto el texto del cuadro de texto como la tabla se actualizan:

```
import { useState } from 'react';

function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);

  return (
    <div>
```

```
      <SearchBar
        filterText={filterText}
        inStockOnly={inStockOnly} />
      <ProductTable
        products={products}
        filterText={filterText}
        inStockOnly={inStockOnly} />
    </div>
  );
}

function ProductCategoryRow({ category }) {
  return (
    <tr>
      <th colSpan="2">
        {category}
      </th>
    </tr>
  );
}

function ProductRow({ product }) {
  const name = product.stocked ? product.name :
    <span style={{ color: 'red' }}>
      {product.name}
    </span>;

  return (
    <tr>
```

```
      <td>{name}</td>
      <td>{product.price}</td>
    </tr>
  );
}

function ProductTable({ products, filterText, inStockOnly })
{
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (
      product.name.toLowerCase().indexOf(
        filterText.toLowerCase()
      ) === -1
    ) {
      return;
    }
    if (inStockOnly && !product.stocked) {
      return;
```

En el ejemplo de código de arriba ProductTable y SearchBar leen las props filterText e inStockOnly para renderizar la tabla, el cuadro de texto, y el checkbox. Por ejemplo, aquí tenemos como SearchBar puebla el valor del cuadro de texto:

```
function SearchBar({ filterText, inStockOnly }) {
  return (
    <form>
      <input
```

```
        type="text"
        value={filterText}
        placeholder="Search..."/>
```

Sin embargo, no haz añadido ningún código para responder a las acciones del usuario como la escritura en el teclado.

7. Añade flujo de datos inverso

Actualmente tu aplicación se renderiza correctamente con props y estado fluyendo hacia abajo en la jerarquía. Pero para cambiar el estado de acuerdo a la entrada del usuario necesitarás ser capaz de manejar datos fluyendo en la otra dirección: los componentes de formulario que se encuentran debajo en la jerarquía necesitan actualizar el estado en FilterableProductTable.

React hace este flujo de datos explícito, pero requiere un poco más de escritura que el enlazado de datos en doble sentido. Si tratas de escribir o seleccionar el checkbox en el ejemplo de arriba, verás que React ignora tu entrada. Esto es intencional. Al escribir <input value={filterText} />, haz establecido que la prop value del input sea siempre igual al estado filterState pasado desde FilterableProductTable. Dado que el estado filterText nunca es modificado, el input nunca cambia.

Debes lograr que cuando el usuario cambie las entradas del formulario, el estado se actualice para reflejar esos cambios. El estado lo posee FilterableProductTable, por lo que solo él puede llamar a setFilterText y setInStockOnly. Para permitir que SearchBar actualice el estado de FilterableProductTable necesitas pasar estas funciones para abajo hacia SearchBar:

```
function FilterableProductTable({ products }) {
  const [filterText, setFilterText] = useState('');
  const [inStockOnly, setInStockOnly] = useState(false);

  return (
    <div>
```

```
<SearchBar
  filterText={filterText}
  inStockOnly={inStockOnly}
  onFilterTextChange={setFilterText}
  onInStockOnlyChange={setInStockOnly} />
```

Dentro de SearchBar, añadirás el manejador del evento onChange y modificarás el estado del padre desde allí:

```
<input
  type="text"
  value={filterText}
  placeholder="Search..."
  onChange={(e) => onFilterTextChange(e.target.value)} />
```