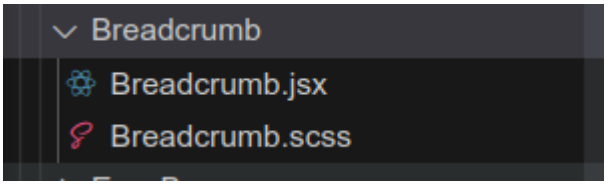
Documentación

Test FrontEnd Mercado libre - Ricardo Segura Cuanalo

Estructura del proyecto

Carpeta src/components

Son componentes de react que podrían ser reutilizables.
Seguí el siguiente convención para un crear un componente:

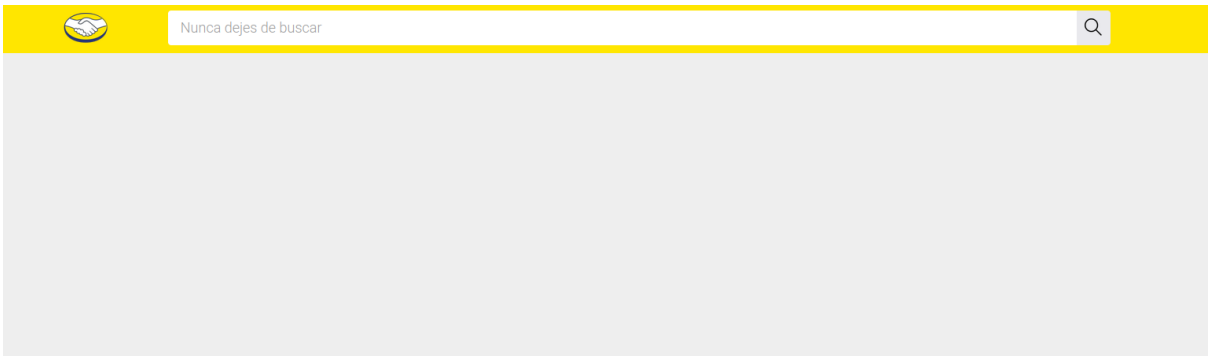


1. Crear una carpeta con el nombre del componente (por si es necesario agrupar más lógica o estilos en cada componente)
2. Crear un archivo con extensión .jsx y un archivo .scss con el mismo nombre (por facilidad al editar o buscar un componente)

Carpeta src/hooks

Contiene la lógica de negocio y algunos hooks genéricos como: useHttpClient y useQueryParams.

Carpeta src/layouts
Esta carpeta contiene partes de una página que son reutilizables, por ejemplo el componente SearchPage renderiza la barra de búsqueda y recibe una propiedad children que se muestra dentro del cuerpo de la página (Aprovechando el patrón composición en react).



Carpeta src/routes
Contiene las rutas definidas del proyecto, utilicé react-router-dom v6.
Define las 3 rutas requeridas en este test en el archivo index.jsx.

```
export const routes = [
  { path: "/", element: createLazyElement(SearchBoxPage) },
  { path: "/items", element: createLazyElement(SearchResultPage) },
  { path: "/items/:id", element: createLazyElement(ProductDetailPage) },
];

export const router = createBrowserRouter(routes);
```

Carpeta src/pages

Son componentes con páginas completas, es decir el componente que renderiza react-router-dom para cada elemento del router.

En este caso utilicé el componente de la carpeta layouts “SearchPage” para mostrar la barra de búsqueda y su funcionalidad.

```jsx
const ProductDetailPage = () => {
  const params = useParams();
  const { value, onChange, onSubmit } = useSearchBar({ name: "search" });
  const { categories, product } = useProductDetail({ id: params.id });
  const onClickBuy = (id) => console.log(`Buy product ${id}`);

  return (
    <SearchPage
      title="Buscador de productos | Mercado libre"
      searchValue={value}
      onChangeSearch={onChange}
      onSubmitSearch={onSubmit}
    >
      <Breadcrumb items={categories} />
      <ProductDetail
        id={product.id}
        title={product.title}
        description={product.description}
        picture={product.picture}
        price={`${product.price}`}
        condition={product.condition}
        soldQuantity={`${product.soldQuantity}`}
        onClickBuy={onClickBuy}
      />
    </SearchPage>
  );
};
```

Hook useHttpClient
Este hook es una abstracción de la librería que usé para hacer peticiones http (axios), por lo regular creo este hook para facilitar el manejo de errores y poder hacer cambios rápidamente en la implementación del cliente http (por ejemplo cambiar de axios a fetch).

Un método interesante de useHttpClient es request(), siempre retorna una respuesta consistente (un objeto con propiedades: response, status y success). De esta manera puedo escribir código más sencillo en lugar de escribir try catch por todos lados.

```
const request = useCallback(
  async ({ method = "", endpoint = "", body = {}, params = {} }) => {
    try {
      const requestConfig = {
        method,
        url: endpoint,
        params,
        data: body,
      };
      const response = await httpInstance.request(requestConfig);
      const success = response.status === 200;
      return { response: response.data, status: response.status, success };
    } catch (error) {
      return {
        response: error.response.data,
        status: error.response.status,
        success: false,
      };
    }
  },
  [httpInstance]
);
```

Reglas para escribir componentes

1. Cada componente se escribe con un export nombrado, es decir no utilizar export default (facilita el intellisense y autocomplete)
2. En lugar de pasar un solo argumento “props” al componente, se pasa un objeto y se aprovecha la desestructuración de objetos de javascript, para facilitar la inicialización y dejar de escribir cosas como: “props.XPropiedad”.
3. Los componentes deben ser cortos y simples, en caso de tener un componente de más de 50 líneas, me replantearía la lógica y trataría de tener más componentes o hooks dependiendo del caso.
4. Deben validarse las props de alguna manera, en este caso utilicé el paquete prop-types que nos imprime en la consola un error cada vez que se renderiza un componente con props inválidas

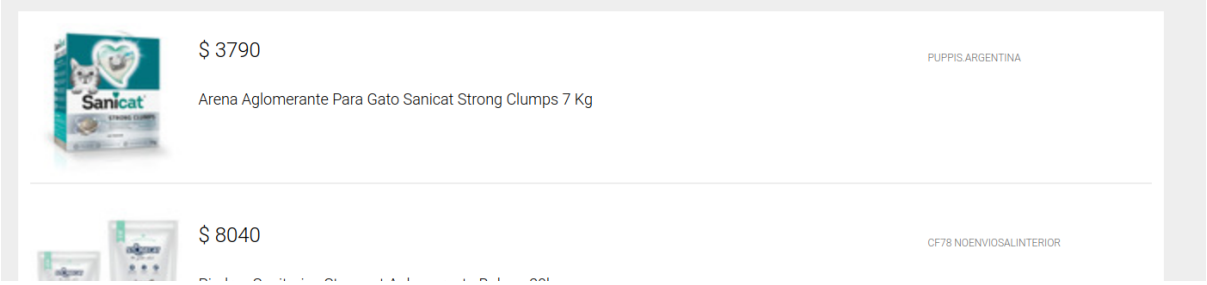
```jsx
import PropTypes from "prop-types";
import "./Breadcrumb.scss";

export const Breadcrumb = ({ items = [] }) => (
  <ul className="breadcrumb" title="Categorías" name="Categorías">
    {items.map((item) => (
      <li className="breadcrumb-item" key={item} title={item} name={item}>
        {item}
      </li>
    ))}
  </ul>
);

Breadcrumb.propTypes = {
  items: PropTypes.array.isRequired,
};
```

Componente ResponsiveImage

Este componente renderiza una imagen pequeña en caso de que el dispositivo tenga una resolución menor a cierto breakpoint y una imagen grande en caso contrario.
Utilicé este componente como base de otros componentes: Logo y FreeShippingIcon

```jsx
import PropTypes from "prop-types";

export const ResponsiveImage = ({
  alt = "",
  mobileImage = "",
  desktopImage = "",
  className = "",
  desktopBreakpoint = 1200,
}) => {
  const src = window.innerWidth <= desktopBreakpoint ? mobileImage : desktopImage;

  return <img src={src} alt={alt} className={className} title={alt} />;
};

ResponsiveImage.propTypes = {
  alt: PropTypes.string.isRequired,
  mobileImage: PropTypes.string.isRequired,
  desktopImage: PropTypes.string.isRequired,
  className: PropTypes.string,
  desktopBreakpoint: PropTypes.number,
};
```

Componente SearchResult
Muestra un resultado de búsqueda como el siguiente:



En este componente utilicé schema.org para facilitar a los buscadores trackear el producto fácilmente y mostrarlo en resultados más relevantes.

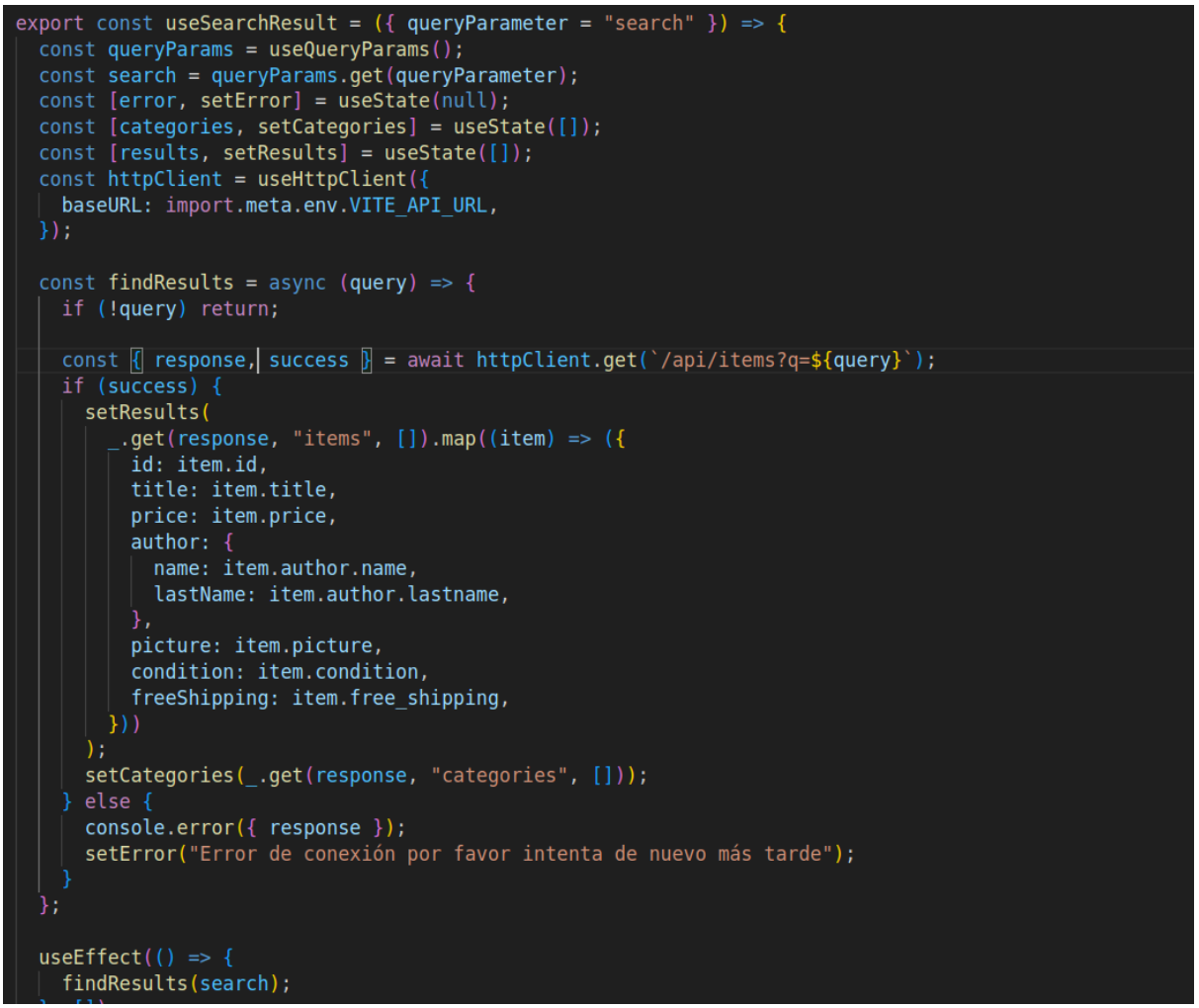
Las propiedades itemScope y itemType definen que la etiqueta “article” contiene información de un producto. La propiedad itemProp define que la etiqueta “img” contiene la imagen del producto.

```
<article
  className="search-result-article"
  itemScope
  itemType="https://schema.org/Product"
>
  <figure name="Imagen del producto">
    <img
      src={picture}
      alt={title}
      className="picture"
      title={title}
      itemProp="image"
    />
```

Hook useSearchResult

Este hook personalizado contiene la lógica necesaria para realizar una búsqueda a partir del parámetro de búsqueda “search”. Usé el hook “useEffect” con un array de dependencias vacío para detonar la petición al api al cargar la página.

Usé el paquete lodash (en el código es importado con un “_”) para evitar realizar un mapeo inseguro como: “response.items.map()”, no importa si el api retorna un error inesperado o no contesta, la página simplemente no mostrará ningún resultado (falla silenciosamente).



```
export const useSearchResult = ({ queryParameter = "search" }) => {
  const queryParams = useQueryParams();
  const search = queryParams.get(queryParameter);
  const [error, setError] = useState(null);
  const [categories, setCategories] = useState([]);
  const [results, setResults] = useState([]);
  const httpClient = useHttpClient({
    baseURL: import.meta.env.VITE_API_URL,
  });

  const findResults = async (query) => {
    if (!query) return;

    const { response, success } = await httpClient.get(`/api/items?q=${query}`);
    if (success) {
      setResults(
        _.get(response, "items", []).map((item) => ({
          id: item.id,
          title: item.title,
          price: item.price,
          author: {
            name: item.author.name,
            lastName: item.author.lastname,
          },
          picture: item.picture,
          condition: item.condition,
          freeShipping: item.free_shipping,
        }))
      );
      setCategories(_.get(response, "categories", []));
    } else {
      console.error({ response });
      setError("Error de conexión por favor intenta de nuevo más tarde");
    }
  };

  useEffect(() => {
    findResults(search);
```

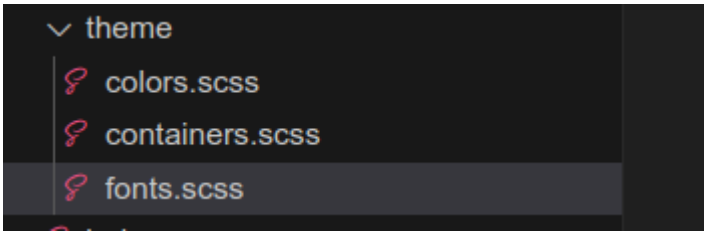
Hook useSearchBar

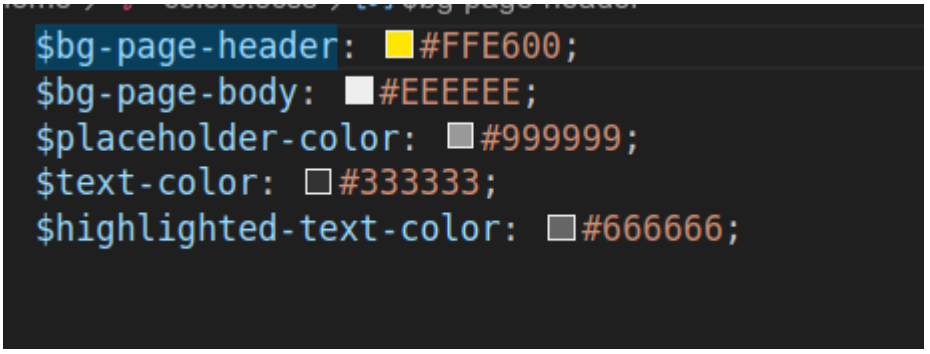
Este hook encapsula la lógica de la barra de búsqueda

- La función sanitizeQuery codifica un string para prevenir envíar carácteres inseguros en la url de la página de búsqueda)
- En caso de que ya exista un parámetro “search” en la url, se carga como valor inicial del input de búsqueda

```
import { useState } from "react";
import _ from "lodash";
import { useQueryParams } from "./useQueryParams";

export const useSearchBar = ({ name = "search" }) => {
  const params = useQueryParams();
  const querySearch = params.get(decodeURIComponent(name));
  const [search, setSearch] = useState(querySearch || "");
  const onChange = (e) => setSearch(_.get(e, "target.value", ""));

  const sanitizeQuery = (query = "") => {
    return encodeURIComponent(query.trim());
  };

  const onSubmit = (e) => {
    e.preventDefault();
    if (!search) return;
    window.location = `/items?search=${sanitizeQuery(search)}`;
  };

  return { name, value: search, onChange, onSubmit };
};
```

Estilos
Utilicé el paquete “sass” para poder importar archivos scss desde un archivo jsx.
Para los estilos que son comunes entre vistas o componentes hice una carpeta **theme/** que contiene archivos de scss con colores (colors.scss), utilidades para hacer columnas (containers.scss) y tipos de letra (fonts.scss).

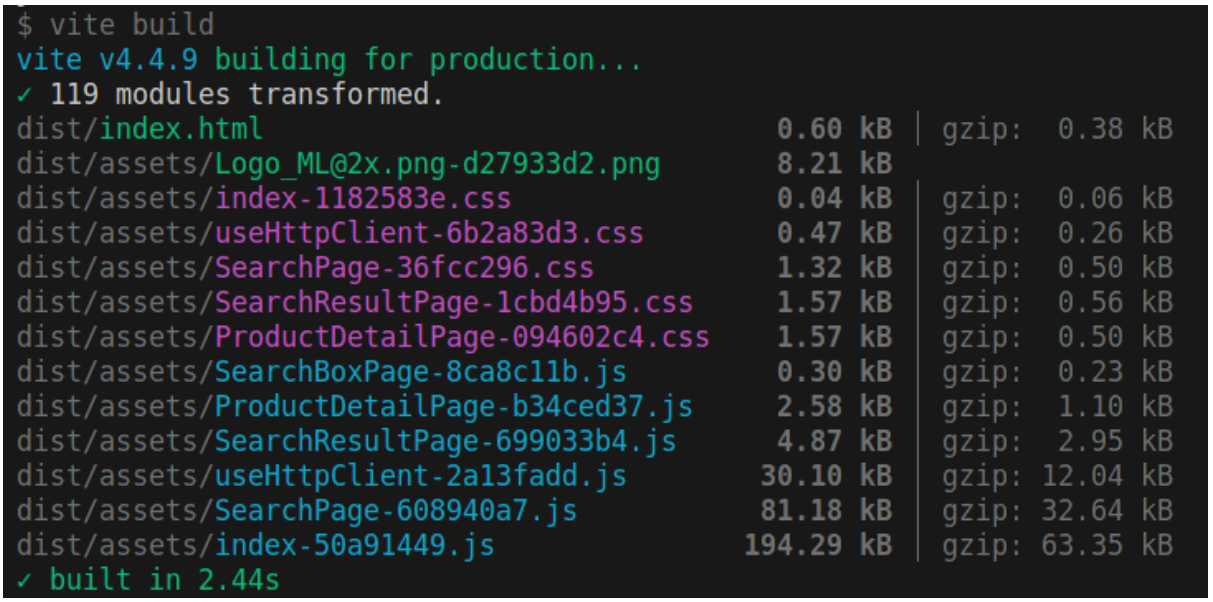




Compilación y performance
Como herramienta de compilación utilicé vite ya que es de los mejores bundlers existentes y en comparación a otros bundlers es más rápido y fácil de utilizar.

Utilicé la función de code splitting de react (lazy) para generar tres chunks diferentes, uno por cada página del proyecto. Esto incrementa la velocidad de carga de la página y genera compilados más pequeños.

En la imagen se muestran los chunks: SearchPage, ProductDetailPage y SearchBoxPage

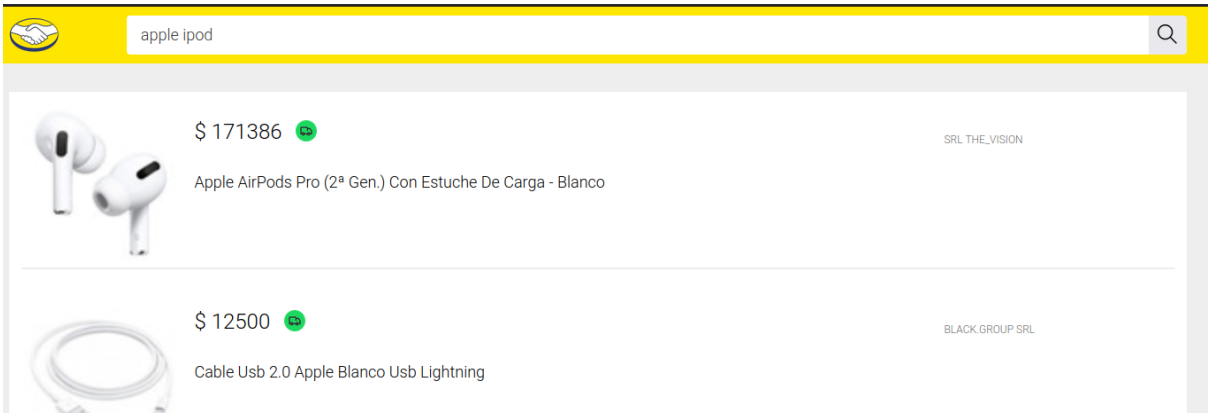


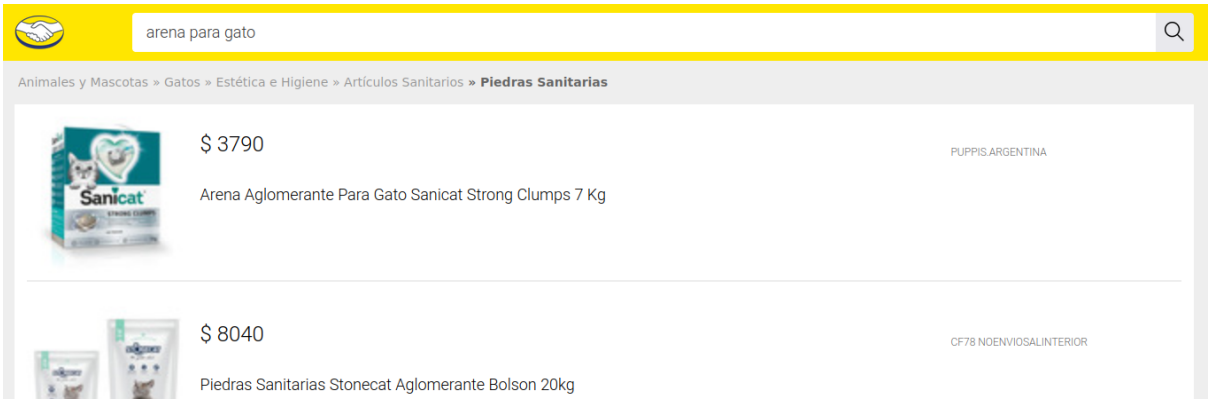
Posibles mejoras

1. Incluí las imágenes del proyecto en la carpeta assets, esto no es lo mejor debido a que cada vez que se compila el proyecto vite debe copiar estas imágenes a la carpeta **dist/**, como mejora se podrían poner esas imágenes en un servicio de distribución de contenido especializado como cloudfront o cloudfare.
2. El componente ProductDetail es muy grande y podría refactorizarse en componentes más pequeños (BuyButton, ProductDescription, ProductQuantity, ProductPicture)
3. No utilicé ningún framework de css para enfatizar el uso de “sass”, sin embargo para tener una implementación más simple en cuanto a columnas y diseño responsive, lo mejor podría ser usar tailwind o bulma css y sacar provecho de sus sistemas de columnas y contenedores (Aunque aumentaría el tamaño del compilado pero acortaría el tiempo de desarrollo)
4. El código del api en express solo cumple con lo requerido, pero debido a que no es el objetivo de este test probar mis habilidades como desarrollador backend no le presté mucha atención.

Dificultades en el desarrollo

1. El endpoint GET https://api.mercadolibre.com/sites/MLA/search?q=query no siempre retorna categorías para todas las búsquedas, por lo que las categorías no se muestran en casos como “Apple Ipod”, pero si se muestran con “Arena para gato”





2. Para mostrar las categorías en la página de detalle de producto los endpoints GET https://api.mercadolibre.com/items/{id}/description y GET https://api.mercadolibre.com/items/{id} no retornan las categorías del producto. Pensé en agregar las categorías al localStorage del navegador una vez que se realiza una búsqueda, pero realmente no es una buena solución ya que si se accede a la ruta **/items/{id}** directamente tampoco se mostrarían las categorías. Creo que la mejor implementación sería que el backend retornara la lista de categorías filtrando por id de producto en otro endpoint.