Code Review by Paris Chrysanthou

1. Performance: No memoization of handlers

In the current implementation, pagination handlers are defined in the following way:

```
<button onClick={() => setPage(page - 1)}>Previous</button>
```

This leads to unnecessary re-creation of functions on every render, which can lead to:

- Prevent memoized components from optimizing correctly
- Cause unnecessary virtual DOM diffs and rerenders

To deal with this, we can utilize `useCallback` to memoize handlers:

```
const handlePrevPage = useCallback(() => setPage((page) => page - 1),
[]);
const handleNextPage = useCallback(() => setPage((page) => page + 1),
[]);
```

And in turn, update the usage:

```
<button onClick={handlePrevPage}...</button>
<button onClick={handleNextPage}...</button>
```

2. Accessibility: Redundant alt text and inaccessible items

Currently, each image uses an alt text based on the title, and the potentially clickable content items are plain divs:

```
<img src={item.thumbnail} alt={item.title} />
<div className="content-item" style={{ cursor: 'pointer' }}>...</div>
```

The following violates multiple WCAG guidelines:

- Images lack proper alt text descriptions
- Inability to navigate the items using a keyboard
- Bad screen reader accessibility
- No focus management for screen readers

To solve the following issues, we can use interactive HTML elements (button, a or article), improve alt text and add Aria labels to each item:

```
<button
   key={item.id}
   onClick={...}
   className="content-item"
   aria-label={`${item.title}, ${item.year}, rated ${item.rating} out of
10`}
>
   <img
       src={item.thumbnail}
       alt={`Poster of ${item.title}`}
   />
   <h3>{item.title}</h3>
   <p>{item.year} • {item.rating}/10</p>
</button>
```

3. Code Structure: Tight coupling of data fetching logic and error handling

Data fetching, error handling, and loading state are embedded directly in the component's useEffect:

```
useEffect(() => {
   setLoading(true);
   fetchContent(page)
       .then((data) => {
           setTrendingContent(data.categories.trending);
           setError('');
       })
       .catch((err) => {
           setError('Failed to load content');
           console.log(err);
       })
}, [page]);
```

This makes the component harder to test, harder to reuse, and violates separation of concerns.

We can properly structure this by extracting the logic into a reusable hook like so:

```
const useTrendingContent = (page: number) => {
  const [data, setData] = useState<ContentItem[]>([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState('');

  useEffect(() => {
    setLoading(true);
    fetchContent(page)
      .then((res) => {
        setData(res.categories.trending);
        setError('');
      })
      .catch(() => setError('Failed to load content'))
      .finally(() => setLoading(false));
  }, [page]);

  return { data, loading, error };
};
```

And use the hook inside the main component:

```
const { data: trendingContent, loading, error } =
useTrendingContent(page);
```