Code Review by Paris Chrysanthou

1. Performance: No caching of paginated content

In the current implementation, when a new page is fetched, the previously loaded trending content is replaced entirely:

```
setTrendingContent(data.categories.trending);
```

This results in:

- Reduced performance
- Redundant data fetching when visiting previously loaded pages
- Unnecessary network requests if the user navigates back to a previous page

To solve this, we should implement a simple in-memory caching for paginated content. Store each page's results in an object like:

```
const [cache, setCache] = useState<Record<number, ContentItem[]>>({});
```

And update the useEffect:

```
useEffect(() => {
   if (cache[page]) {
     setTrendingContent(cache[page]);
     return;
   }
   setLoading(true);
   fetchContent(page)
     .then((data) => {
       const items = data.categories.trending;
       setCache((prev) => ({ ...prev, [page]: items }));
       setTrendingContent(items);
       setError('');
     })
     .catch(() => setError('Failed to load content'))
     .finally(() => setLoading(false));
 }, [page, cache]);
```

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);
```