ChatGPT - React Window Large List

example of large list array object in react with using react window and show details of the itm

Using `react-window`, you can efficiently render large lists by virtualizing them. Here's an example of a React component that displays a large list of objects, each with a "Show Details" button that reveals more information about the selected item.

Steps

1. **Install `react-window`:**

   ```bash
   npm install react-window
   ```

2. **Code Example:**

```tsx
import React, { useState } from "react";
import { FixedSizeList as List } from "react-window";

// Sample data: A large list of objects
const generateItems = () => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

const ListItem = ({ index, style, data }: any) => {
  const item = data.items[index];
  const { setSelectedItem } = data;

  return (
    <div style={style} className="list-item">
      <div>
```

```
        <strong>{item.name}</strong>
        <button
          style={{ marginLeft: "10px" }}
          onClick={() => setSelectedItem(item)}
        >
          Show Details
        </button>
      </div>
    </div>
  );
};

const App: React.FC = () => {
  const [selectedItem, setSelectedItem] = useState(null);

  return (
    <div style={{ display: "flex" }}>
      <div style={{ flex: "1", height: "400px", border: "1px solid #ccc" }}>
        <h3>Item List</h3>
        <List
          height={400}
          itemCount={items.length}
          itemSize={50}
          width={300}
          itemData={{ items, setSelectedItem }}
        >
          {ListItem}
        </List>
      </div>
      <div style={{ flex: "1", marginLeft: "20px" }}>
        <h3>Details</h3>
        {selectedItem ? (
          <div>
            <p>
              <strong>ID:</strong> {selectedItem.id}
            </p>
            <p>
              <strong>Name:</strong> {selectedItem.name}
            </p>
            <p>
              <strong>Description:</strong> {selectedItem.description}
            </p>
          </div>
        ) : (
          <p>Select an item to view details.</p>
        )}
      </div>
    </div>
  );
};

export default App;
```

Explanation:

1. **Data Structure**:
   - The `items` array contains 1000 objects, each with an `id`, `name`, and `description`.
2. **Virtualization**:
   - `react-window`'s `FixedSizeList` is used to efficiently render only the visible items within the scroll viewport.
3. **Item Component**:
   - `ListItem` renders each item and includes a button to set the `selectedItem`.
4. **Details View**:
   - When an item is selected, its details are displayed in the adjacent pane.

Styling (Optional):

Add some CSS for better visuals:

```css
.list-item {
  display: flex;
  align-items: center;
  justify-content: space-between;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}
button {
  padding: 5px 10px;
  cursor: pointer;
}
```

Preview

- **Left Pane**: List of items.
- **Right Pane**: Details of the selected item.
- **Performance**: Handles thousands of items smoothly due to virtualization.

show it with css and not with state

If you want to implement the same functionality using **CSS** to show the details instead of managing it with React state, you can use a combination of CSS for styling and a `:hover` or `:focus` approach. Here's how you can achieve it:

Code Example (CSS-Based Approach)

```tsx
import React from "react";
import { FixedSizeList as List } from "react-window";
import "./App.css"; // Import CSS styles

// Sample data: A large list of objects
const generateItems = () => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

const ListItem = ({ index, style, data }: any) => {
  const item = data.items[index];

  return (
    <div style={style} className="list-item">
      <div className="item-header">
        <strong>{item.name}</strong>
        <button className="details-button">Show Details</button>
      </div>
      <div className="item-details">
        <p>
          <strong>ID:</strong> {item.id}
        </p>
        <p>
          <strong>Name:</strong> {item.name}
        </p>
        <p>
          <strong>Description:</strong> {item.description}
        </p>
      </div>
    </div>
  );
};

const App: React.FC = () => {
  return (
    <div style={{ height: "400px", width: "300px", border: "1px solid #ccc" }}>
      <List
        height={400}
        itemCount={items.length}
        itemSize={100}
        width={300}
        itemData={{ items }}
      >
        {ListItem}
      </List>
    </div>
  );
};

export default App;
```

CSS (`App.css`)

```css
/* General Styles */
.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
  overflow: hidden;
  transition: all 0.3s ease;
}

.item-header {
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.details-button {
  background-color: #007bff;
  color: white;
  border: none;
  padding: 5px 10px;
  cursor: pointer;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.details-button:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when hovering */
.list-item:hover .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}
```

Explanation:

1. **CSS Transition**:
    - The `.item-details` section is hidden initially (`max-height: 0; opacity: 0`).
    - On hover (`:hover`), it expands (`max-height: 100px; opacity: 1`).
2. **Details Visibility**:
    - No React state is needed. The visibility is purely controlled through CSS.
3. **Virtualization**:
    - The `react-window` library ensures performance when rendering a large list.
4. **Dynamic Styling**:
    - You can modify styles like `max-height` in `.item-details` to fit your content dynamically.

Preview

- **Default State**: Only item names are visible.
- **Hover State**: When you hover over an item, the details slide into view smoothly.

This approach avoids managing additional state in React while leveraging the power of CSS for dynamic effects.

show it with css checked

To achieve the "show details" functionality using a `:checked` CSS approach, you can use a hidden checkbox input. Here's how you can implement this:

Code Example (Using `:checked` in CSS)

```tsx
import React from "react";
import { FixedSizeList as List } from "react-window";
import "./App.css"; // Import CSS styles

// Sample data: A large list of objects
const generateItems = () => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

const ListItem = ({ index, style, data }: any) => {
  const item = data.items[index];

  return (
    <div style={style} className="list-item">
      <input type="checkbox" id={`toggle-${item.id}`} className="toggle-input" />
      <label htmlFor={`toggle-${item.id}`} className="toggle-label">
        <strong>{item.name}</strong>
        <span>Show Details</span>
      </label>
      <div className="item-details">
        <p>
          <strong>ID:</strong> {item.id}
        </p>
        <p>
          <strong>Name:</strong> {item.name}
        </p>
        <p>
          <strong>Description:</strong> {item.description}
        </p>
      </div>
    </div>
  );
};

const App: React.FC = () => {
  return (
    <div style={{ height: "400px", width: "300px", border: "1px solid #ccc" }}>
      <List
        height={400}
        itemCount={items.length}
        itemSize={100}
        width={300}
        itemData={{ items }}
      >
        {ListItem}
      </List>
    </div>
  );
};

export default App;
```

CSS (`App.css`)

```css
/* General Styles */
.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
  overflow: hidden;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}
```

```
.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}
```

Explanation:

1. **Checkbox as a Toggle**:
    - Each list item has a hidden `<input type="checkbox">` element.
    - The `:checked` pseudo-class is used to toggle the visibility of the details.
2. **Label for Interaction**:
    - The `<label>` is associated with the checkbox via the `htmlFor` attribute. Clicking the label toggles the checkbox.
3. **CSS Transition**:
    - The `.item-details` section is initially hidden (`max-height: 0; opacity: 0`).
    - When the checkbox is checked (`:checked`), it expands smoothly (`max-height: 100px; opacity: 1`).
4. **Virtualization**:
    - The `react-window` library ensures efficient rendering for large lists.

Preview

- **Default State**: The list displays only item names.
- **Checked State**: Clicking "Show Details" expands the item details.

This approach uses CSS's `:checked` pseudo-class for toggle functionality, avoiding JavaScript state management entirely.

all this example with react ts

Here is the full example implemented in **React with TypeScript**:

React Component (`App.tsx`)

```tsx
import React from "react";
import { FixedSizeList as List } from "react-window";
import "./App.css"; // Import CSS styles

// Define the type for an item
interface Item {
  id: number;
  name: string;
  description: string;
}

// Generate a large list of sample items
const generateItems = (): Item[] => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

// Props for ListItem
interface ListItemProps {
  index: number;
  style: React.CSSProperties;
  data: {
    items: Item[];
  };
}
```

```
// Render each list item
const ListItem: React.FC<ListItemProps> = ({ index, style, data }) => {
  const item = data.items[index];

  return (
    <div style={style} className="list-item">
      <input
        type="checkbox"
        id={`toggle-${item.id}`}
        className="toggle-input"
      />
      <label htmlFor={`toggle-${item.id}`} className="toggle-label">
        <strong>{item.name}</strong>
        <span>Show Details</span>
      </label>
      <div className="item-details">
        <p>
          <strong>ID:</strong> {item.id}
        </p>
        <p>
          <strong>Name:</strong> {item.name}
        </p>
        <p>
          <strong>Description:</strong> {item.description}
        </p>
      </div>
    </div>
  );
};

const App: React.FC = () => {
  return (
    <div style={{ height: "400px", width: "300px", border: "1px solid #ccc" }}>
      <List
        height={400}
        itemCount={items.length}
        itemSize={100}
        width={300}
        itemData={{ items }}
      >
        {ListItem}
      </List>
    </div>
  );
};

export default App;
```

CSS File (`App.css`)

```css

/* General Styles */
.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
  overflow: hidden;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

```

```
.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}
```

Explanation:

1. **TypeScript Interfaces**:
    - `Item`: Defines the structure of each list item.
    - `ListItemProps`: Specifies the props expected by the `ListItem` component.
2. **Virtualized List**:
    - `react-window` is used to render only the visible items in the viewport for efficiency.
3. **Checkbox Toggle**:
    - Each `ListItem` includes a hidden checkbox (`input[type="checkbox"]`).
    - The associated label (`<label>`) toggles the checkbox when clicked.
4. **CSS Transitions**:
    - The `.item-details` section is initially hidden.
    - When the checkbox is checked (`:checked`), the details expand smoothly.

Running the Example:

1. Install dependencies:

   bash

   ```bash
   npm install react react-dom react-scripts react-window
   ```

2. Start the project:

   bash

   ```bash
   npm start
   ```

Output:

- **Default State**: Displays the item name and a "Show Details" button.
- **Checked State**: Clicking the button reveals the item details below it.

This example provides a full TypeScript solution with clean code, efficient rendering, and CSS-based toggling using the `:checked` pseudo-class.

> all this example with pagination that have next and prev with list of 100 each page

Here's the full example with **pagination** for a list of 100 items per page, implemented using **React with TypeScript**. The pagination controls allow navigation between pages, showing only 100 items at a time.

React Component (`App.tsx`)

tsx

```tsx
import React, { useState } from "react";
import { FixedSizeList as List } from "react-window";
import "./App.css"; // Import CSS styles

// Define the type for an item
```

```
interface Item {
  id: number;
  name: string;
  description: string;
}

// Generate a large list of sample items
const generateItems = (): Item[] => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

// Props for ListItem
interface ListItemProps {
  index: number;
  style: React.CSSProperties;
  data: {
    items: Item[];
  };
}

// Render each list item
const ListItem: React.FC<ListItemProps> = ({ index, style, data }) => {
  const item = data.items[index];

  return (
    <div style={style} className="list-item">
      <input
        type="checkbox"
        id={`toggle-${item.id}`}
        className="toggle-input"
      />
      <label htmlFor={`toggle-${item.id}`} className="toggle-label">
        <strong>{item.name}</strong>
        <span>Show Details</span>
      </label>
      <div className="item-details">
```

```
        <p>
          <strong>ID:</strong> {item.id}
        </p>
        <p>
          <strong>Name:</strong> {item.name}
        </p>
        <p>
          <strong>Description:</strong> {item.description}
        </p>
      </div>
    </div>
  );
};

const App: React.FC = () => {
  const itemsPerPage = 100; // Number of items per page
  const [currentPage, setCurrentPage] = useState(1);

  // Calculate the items to display for the current page
  const startIndex = (currentPage - 1) * itemsPerPage;
  const currentItems = items.slice(startIndex, startIndex + itemsPerPage);

  // Handle navigation
  const totalPages = Math.ceil(items.length / itemsPerPage);
  const handleNext = () => {
    if (currentPage < totalPages) setCurrentPage(currentPage + 1);
  };

  const handlePrev = () => {
    if (currentPage > 1) setCurrentPage(currentPage - 1);
  };

  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      <div style={{ height: "400px", border: "1px solid #ccc" }}>
        <List
          height={400}
          itemCount={currentItems.length}
          itemSize={100}
          width={300}
          itemData={{ items: currentItems }}
        >
```

```
          {ListItem}
        </List>
      </div>
      {/* Pagination Controls */}
      <div className="pagination">
        <button
          className="pagination-button"
          onClick={handlePrev}
          disabled={currentPage === 1}
        >
          Previous
        </button>
        <span className="page-info">
          Page {currentPage} of {totalPages}
        </span>
        <button
          className="pagination-button"
          onClick={handleNext}
          disabled={currentPage === totalPages}
        >
          Next
        </button>
      </div>
    </div>
  );
};

export default App;
```

CSS File (`App.css`)

css

```css
/* General Styles */
.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
  overflow: hidden;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Pagination Styles */
```

```css
.pagination {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-top: 10px;
}

.pagination-button {
  padding: 5px 10px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  transition: background-color 0.2s ease;
}

.pagination-button:disabled {
  background-color: #ddd;
  cursor: not-allowed;
}

.pagination-button:hover:not(:disabled) {
  background-color: #0056b3;
}

.page-info {
  font-size: 0.9em;
}
```

Explanation:

1. **Pagination Logic:**
   - The `currentPage` state keeps track of the current page.
   - Items are sliced based on the page using `items.slice(startIndex, startIndex + itemsPerPage)`.
2. **Next and Previous Buttons:**

    - `handleNext` and `handlePrev` update the `currentPage`.
    - Buttons are disabled when navigating beyond the available pages.
3. **Virtualized List**:
    - The `react-window` library efficiently renders only the visible items, even with pagination.
4. **CSS Transitions**:
    - The `.item-details` expands using `:checked` pseudo-class.
5. **Responsive Layout**:
    - The component is centered with a fixed width and pagination controls below the list.

How to Run:

1. Install dependencies:

```bash
npm install react react-dom react-scripts react-window
```

2. Start the project:

```bash
npm start
```

Result:

- **List**: Displays 100 items per page.
- **Pagination**: Click **Next** and **Previous** to navigate between pages.
- **Efficient Rendering**: Handles large lists smoothly with virtualization.

this example with pagination that have next and prev with list of 100 each page without react window

Here’s how to implement pagination for a list of 100 items per page **without** `react-window`, using a simple list rendering approach in React with TypeScript:

React Component (`App.tsx`)

```tsx
import React, { useState } from "react";
import "./App.css";

// Define the type for an item
interface Item {
  id: number;
  name: string;
  description: string;
}

// Generate a large list of sample items
const generateItems = (): Item[] => {
  return Array.from({ length: 1000 }, (_, index) => ({
    id: index + 1,
    name: `Item ${index + 1}`,
    description: `This is the description for Item ${index + 1}.`,
  }));
};

const items = generateItems();

const App: React.FC = () => {
  const itemsPerPage = 100; // Number of items per page
  const [currentPage, setCurrentPage] = useState(1);

  // Calculate the items to display for the current page
  const startIndex = (currentPage - 1) * itemsPerPage;
  const currentItems = items.slice(startIndex, startIndex + itemsPerPage);

  // Handle navigation
  const totalPages = Math.ceil(items.length / itemsPerPage);
  const handleNext = () => {
    if (currentPage < totalPages) setCurrentPage(currentPage + 1);
  };

  const handlePrev = () => {
    if (currentPage > 1) setCurrentPage(currentPage - 1);
  };

  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      <div className="list-container">
        {currentItems.map((item) => (
          <div key={item.id} className="list-item">
            <input
              type="checkbox"
              id={`toggle-${item.id}`}
              className="toggle-input"
            />
            <label htmlFor={`toggle-${item.id}`} className="toggle-label">
              <strong>{item.name}</strong>
              <span>Show Details</span>
            </label>
            <div className="item-details">
              <p>
                <strong>ID:</strong> {item.id}
              </p>
              <p>
                <strong>Name:</strong> {item.name}
              </p>
              <p>
                <strong>Description:</strong> {item.description}
              </p>
            </div>
          </div>
        ))}
      </div>
      {/* Pagination Controls */}
      <div className="pagination">
        <button
          className="pagination-button"
          onClick={handlePrev}
          disabled={currentPage === 1}
        >
          Previous
        </button>
        <span className="page-info">
          Page {currentPage} of {totalPages}
        </span>
        <button
          className="pagination-button"
          onClick={handleNext}
          disabled={currentPage === totalPages}
        >
          Next
        </button>
      </div>
    </div>
  );
};

export default App;
```

CSS File (`App.css`)

```css
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Pagination Styles */
.pagination {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-top: 10px;
}

.pagination-button {
  padding: 5px 10px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  transition: background-color 0.2s ease;
}

.pagination-button:disabled {
  background-color: #ddd;
  cursor: not-allowed;
}

.pagination-button:hover:not(:disabled) {
  background-color: #0056b3;
}

.page-info {
  font-size: 0.9em;
}
```

Explanation:

1. **Pagination Logic**:
    - The `currentPage` state tracks the current page.
    - The items for the current page are calculated using `items.slice`.
2. **Rendering**:
    - The `map` method renders only the `currentItems` for the current page.
3. **Styling**:
    - The `.list-container` limits the visible height of the list and adds a scrollbar if needed.
    - The `.pagination` section includes `Previous` and `Next` buttons and a page info display.

4. **Checkbox Toggle**:
    - Each list item has a checkbox to toggle visibility of the details.

How to Run:

1. Install dependencies (if not already installed):

    ```bash
    npm install react react-dom react-scripts
    ```

2. Start the project:

    ```bash
    npm start
    ```

Result:

- **List**: Displays 100 items per page with scrolling enabled.
- **Pagination**: Navigate between pages using the **Next** and **Previous** buttons.
- **Details Toggle**: Expand item details using the checkbox for each item.

This approach avoids `react-window` while still ensuring simplicity and efficiency for medium-sized lists.

this example but with call to api to backend

To adapt the previous example for fetching data from an API backend and implementing pagination, we will assume the backend provides an endpoint to fetch the data, and it supports pagination (e.g., `GET /items?page=1&limit=100`).

Here’s how you can implement it in React with TypeScript, making API calls to fetch the data:

1. Backend API Example (for reference)

The backend should provide an API like:

```bash
GET /items?page=1&limit=100
```

- `page` : The page number.
- `limit` : The number of items per page.

The backend should respond with a paginated result like:

```json
{
  "total": 1000,
  "items": [
    { "id": 1, "name": "Item 1", "description": "Description for item 1" },
    { "id": 2, "name": "Item 2", "description": "Description for item 2" },
    // ...
  ]
}
```

2. React Frontend with Pagination and API Calls

Here's how you can modify the React code to fetch data from an API and handle pagination.

React Component (`App.tsx`)

```tsx
import React, { useState, useEffect } from "react";
import axios from "axios"; // Make sure to install axios
import "./App.css";

// Define the type for an item
interface Item {
  id: number;
  name: string;
  description: string;
```

```
}

// Define the API response structure
interface ApiResponse {
  total: number;
  items: Item[];
}

const App: React.FC = () => {
  const itemsPerPage = 100; // Number of items per page
  const [currentPage, setCurrentPage] = useState(1);
  const [items, setItems] = useState<Item[]>([]);
  const [totalItems, setTotalItems] = useState<number>(0);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);

  // Fetch data from the API
  const fetchData = async (page: number) => {
    setLoading(true);
    setError(null);

    try {
      const response = await axios.get<ApiResponse>(`/api/items`, {
        params: {
          page,
          limit: itemsPerPage,
        },
      });

      setItems(response.data.items);
      setTotalItems(response.data.total);
    } catch (error) {
      setError("Failed to fetch data.");
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData(currentPage);
  }, [currentPage]);
```

```
// Handle pagination navigation
const totalPages = Math.ceil(totalItems / itemsPerPage);

const handleNext = () => {
  if (currentPage < totalPages) setCurrentPage(currentPage + 1);
};

const handlePrev = () => {
  if (currentPage > 1) setCurrentPage(currentPage - 1);
};

return (
  <div style={{ width: "320px", margin: "0 auto" }}>
    {/* List Section */}
    <div className="list-container">
      {loading && <p>Loading...</p>}
      {error && <p style={{ color: "red" }}>{error}</p>}
      {!loading && !error && items.length === 0 && <p>No items found.</p>}
      {!loading &&
        !error &&
        items.map((item) => (
          <div key={item.id} className="list-item">
            <input
              type="checkbox"
              id={`toggle-${item.id}`}
              className="toggle-input"
            />
            <label htmlFor={`toggle-${item.id}`} className="toggle-label">
              <strong>{item.name}</strong>
              <span>Show Details</span>
            </label>
            <div className="item-details">
              <p>
                <strong>ID:</strong> {item.id}
              </p>
              <p>
                <strong>Name:</strong> {item.name}
              </p>
              <p>
                <strong>Description:</strong> {item.description}
              </p>
            </div>
            </div>
          ))}
      </div>

      {/* Pagination Controls */}
      <div className="pagination">
        <button
          className="pagination-button"
          onClick={handlePrev}
          disabled={currentPage === 1 || loading}
        >
          Previous
        </button>
        <span className="page-info">
          Page {currentPage} of {totalPages}
        </span>
        <button
          className="pagination-button"
          onClick={handleNext}
          disabled={currentPage === totalPages || loading}
        >
          Next
        </button>
      </div>
    </div>
  );
};

export default App;
```

CSS File (`App.css`)

```css
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
```

```css
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Pagination Styles */
.pagination {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-top: 10px;
}

.pagination-button {
  padding: 5px 10px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  transition: background-color 0.2s ease;
}

.pagination-button:disabled {
  background-color: #ddd;
  cursor: not-allowed;
}

.pagination-button:hover:not(:disabled) {
  background-color: #0056b3;
}

.page-info {
  font-size: 0.9em;
}
```

Explanation:

1. **State Management**:
    - `currentPage`: Keeps track of the current page number.
    - `items`: Holds the items for the current page fetched from the API.
    - `totalItems`: Total number of items available (for pagination calculation).
    - `loading`: Boolean to indicate if data is being fetched.
    - `error`: Stores error messages in case the API call fails.
2. **API Call**:
    - `fetchData` function uses `axios` to make an API request to the backend with the current `page` and `limit` (100 items per page).
    - The API response is expected to include the items and the total number of items available.
3. **Pagination Logic**:
    - The `handleNext` and `handlePrev` functions update the `currentPage` based on user input.
    - The total number of pages is calculated using `Math.ceil(totalItems / itemsPerPage)`.
4. **Conditional Rendering**:
    - Loading state is shown while the data is being fetched.
    - Error message is displayed if the API call fails.
    - List items are rendered only when data is available.
5. **Pagination Controls**:
    - "Previous" and "Next" buttons allow navigation between pages.
    - These buttons are disabled when on the first or last page, respectively.

How to Run:

1. **Install Dependencies**: You'll need `axios` for API requests. Install it using:

```bash
```

```
npm install axios
```

2. **Backend API**: Ensure you have a backend API that handles pagination requests (`/api/items?page=1&limit=100`).
3. **Start the React App**:

```bash
npm start
```

Result:

- **List**: Displays 100 items per page.
- **Pagination**: Navigate between pages using **Next** and **Previous** buttons.
- **API-Driven**: The items are fetched from a backend API, and the list updates dynamically based on the current page.

This solution ensures that the frontend fetches paginated data from the backend and displays it efficiently with pagination controls.

this example but with infinty scroll and not pagaintion

To implement **infinite scroll** instead of pagination, you need to modify the logic to fetch more data as the user scrolls to the bottom of the list. This involves detecting when the user reaches the bottom of the scrollable container and then triggering an API call to fetch more items.

Here’s how you can update the React app with infinite scroll functionality.

Steps:

- **Triggering data fetch**: As the user scrolls down and reaches the bottom of the list, more items are loaded.
- **Avoiding multiple API calls**: We need to ensure that multiple fetches don't happen simultaneously (by using a `loading` state).

React Component (`App.tsx`)

```tsx
```

```
import React, { useState, useEffect, useRef } from "react";
import axios from "axios"; // Make sure to install axios
import "./App.css";

// Define the type for an item
interface Item {
  id: number;
  name: string;
  description: string;
}

// Define the API response structure
interface ApiResponse {
  total: number;
  items: Item[];
}

const App: React.FC = () => {
  const itemsPerPage = 100; // Number of items to load per request
  const [items, setItems] = useState<Item[]>([]);
  const [totalItems, setTotalItems] = useState<number>(0);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
  const [page, setPage] = useState<number>(1); // Current page
  const containerRef = useRef<HTMLDivElement>(null); // For detecting scroll position

  // Fetch data from the API
  const fetchData = async (page: number) => {
    if (loading) return; // Prevent multiple simultaneous API calls

    setLoading(true);
    setError(null);

    try {
      const response = await axios.get<ApiResponse>(`/api/items`, {
        params: {
          page,
          limit: itemsPerPage,
        },
      });
      setItems((prevItems) => [...prevItems, ...response.data.items]); // Append new items
      setTotalItems(response.data.total);
    } catch (error) {
      setError("Failed to fetch data.");
    } finally {
      setLoading(false);
    }
  };

  // Infinite scroll logic - trigger API call when user scrolls to bottom
  const handleScroll = () => {
    if (containerRef.current) {
      const bottom =
        containerRef.current.scrollHeight === containerRef.current.scrollTop + containerRef.current.clientHeight;
      if (bottom && !loading && items.length < totalItems) {
        setPage((prevPage) => {
          const newPage = prevPage + 1;
          fetchData(newPage); // Fetch next page of items
          return newPage;
        });
      }
    }
  };

  useEffect(() => {
    fetchData(page); // Fetch initial data on component mount
  }, [page]);

  useEffect(() => {
    // Attach scroll listener to the container when component mounts
    const container = containerRef.current;
    if (container) {
      container.addEventListener("scroll", handleScroll);
    }

    return () => {
      if (container) {
        container.removeEventListener("scroll", handleScroll); // Cleanup scroll listener on unmount
      }
    };
  }, [loading]);

  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      {/* List Section */}
      <div ref={containerRef} className="list-container">
        {loading && <p>Loading...</p>}
        {error && <p style={{ color: "red" }}>{error}</p>}
        {!loading && !error && items.length === 0 && <p>No items found.</p>}
        {!loading &&
          !error &&
          items.map((item) => (
            <div key={item.id} className="list-item">
              <input
                type="checkbox"
                id={`toggle-${item.id}`}
                className="toggle-input"
              />
              <label htmlFor={`toggle-${item.id}`} className="toggle-label">
                <strong>{item.name}</strong>
                <span>Show Details</span>
              </label>
              <div className="item-details">
                <p>
                  <strong>ID:</strong> {item.id}
                </p>
                <p>
                  <strong>Name:</strong> {item.name}
                </p>
                <p>
                  <strong>Description:</strong> {item.description}
                </p>
              </div>
            </div>
          ))}
      </div>

      {/* Footer or Loading State */}
      {loading && <div className="loading-footer">Loading more items...</div>}
    </div>
  );
```

```
};

export default App;
```

CSS File (`App.css`)

```css
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
  padding-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Loading Footer */
.loading-footer {
  text-align: center;
  padding: 10px;
  color: #007bff;
  font-weight: bold;
}
```

Explanation:

1. **State Management**:
   - `page`: Keeps track of the current page for fetching data.
   - `items`: Holds the list of items fetched from the API.

  - `totalItems`: The total number of items available from the API (for stopping infinite scroll when all items are loaded).
  - `loading`: Boolean to indicate when data is being fetched (to prevent multiple API calls at once).
  - `error`: Holds any error messages if the API call fails.
2. **API Fetching**:
   - `fetchData`: Makes an API call using `axios` to fetch items based on the current page and appends them to the existing list.
   - API URL: `/api/items?page=${page}&limit=100`, which returns items for the requested page.
3. **Infinite Scroll**:
   - `handleScroll`: Listens for scroll events and checks if the user has reached the bottom of the list (`containerRef.current.scrollHeight === containerRef.current.scrollTop + containerRef.current.clientHeight`). If true, it triggers a fetch for the next set of items.
   - Scroll listener is added when the component is mounted and removed when the component is unmounted to avoid memory leaks.
4. **Loading State**:
   - Displays a "Loading..." message while data is being fetched.
   - Displays a "Loading more items..." message at the bottom of the list while more data is being loaded.

How to Run:

1. **Install Dependencies**: You'll need `axios` for making API requests:

   ```bash
   npm install axios
   ```

2. **Backend API**: Make sure your backend supports the `GET /api/items?page=1&limit=100` endpoint and returns paginated data.

3. **Start the React App**:

```bash
npm start
```

Result:

- **Infinite Scroll**: As you scroll to the bottom of the list, new items are fetched and appended to the list.
- **Loading State**: Shows a loading message when data is being fetched.
- **Error Handling**: Displays an error message if the API call fails.
- **Efficient Data Fetching**: Fetches data in chunks (100 items per request) and appends it to the list.

This approach makes sure the user can keep scrolling and fetching more items without needing pagination controls, improving user experience with infinite scrolling.

this example with hook for infinty scroll

To refactor the infinite scroll functionality into a custom hook, we can create a hook that handles the scroll detection, API fetching, and state management for infinite scrolling. This custom hook will encapsulate the logic, making it reusable across different components.

Here's how you can implement the **infinite scroll** logic using a custom hook in React with TypeScript.

Steps:

- Create a custom hook called `useInfiniteScroll` to handle the logic.
- Use the custom hook inside the component.

1. Create the Custom Hook (`useInfiniteScroll.ts`)

```tsx
```

```
import { useState, useEffect, useRef } from "react";
import axios from "axios";

// Define the type for an item
export interface Item {
  id: number;
  name: string;
  description: string;
}

// Define the API response structure
export interface ApiResponse {
  total: number;
  items: Item[];
}

const useInfiniteScroll = (
  apiUrl: string,
  itemsPerPage: number
) => {
  const [items, setItems] = useState<Item[]>([]);
  const [totalItems, setTotalItems] = useState<number>(0);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
  const [page, setPage] = useState<number>(1); // Current page
  const containerRef = useRef<HTMLDivElement>(null); // For detecting scroll
position

  // Fetch data from the API
  const fetchData = async (page: number) => {
    if (loading) return; // Prevent multiple simultaneous API calls

    setLoading(true);
    setError(null);

    try {
      const response = await axios.get<ApiResponse>(apiUrl, {
        params: {
          page,
          limit: itemsPerPage,
        },
      });
      setItems((prevItems) => [...prevItems, ...response.data.items]); // Append new items
      setTotalItems(response.data.total);
    } catch (error) {
      setError("Failed to fetch data.");
    } finally {
      setLoading(false);
    }
  };

  // Infinite scroll logic - trigger API call when user scrolls to bottom
  const handleScroll = () => {
    if (containerRef.current) {
      const bottom =
        containerRef.current.scrollHeight === containerRef.current.scrollTop + containerRef.current.clientHeight;
      if (bottom && !loading && items.length < totalItems) {
        setPage((prevPage) => {
          const newPage = prevPage + 1;
          fetchData(newPage); // Fetch next page of items
          return newPage;
        });
      }
    }
  };

  useEffect(() => {
    fetchData(page); // Fetch initial data on component mount
  }, [page]);

  useEffect(() => {
    // Attach scroll listener to the container when component mounts
    const container = containerRef.current;
    if (container) {
      container.addEventListener("scroll", handleScroll);
    }

    return () => {
      if (container) {
        container.removeEventListener("scroll", handleScroll); // Cleanup scroll listener on unmount
      }
    };
  }, [loading]);

  return {
    items,
    loading,
    error,
    containerRef,
  };
};

export default useInfiniteScroll;
```

---

2. Use the Custom Hook in the Component (`App.tsx`)

Now, let's use the custom hook in the main component where you display the list of items with infinite scroll.

```tsx
import React from "react";
import "./App.css";
import useInfiniteScroll, { Item } from "./useInfiniteScroll";

// Define your API URL
const API_URL = "/api/items"; // Update this with your actual backend URL

const App: React.FC = () => {
  const itemsPerPage = 100;
  const { items, loading, error, containerRef } = useInfiniteScroll(API_URL,
itemsPerPage);

  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      {/* List Section */}
      <div ref={containerRef} className="list-container">
        {loading && <p>Loading...</p>}
        {error && <p style={{ color: "red" }}>{error}</p>}
        {!loading && !error && items.length === 0 && <p>No items found.</p>}
        {!loading &&
          !error &&
          items.map((item: Item) => (
            <div key={item.id} className="list-item">
              <input
                type="checkbox"
                id={`toggle-${item.id}`}
                className="toggle-input"
              />
              <label htmlFor={`toggle-${item.id}`} className="toggle-label">
                <strong>{item.name}</strong>
                <span>Show Details</span>
              </label>
              <div className="item-details">
                <p>
                  <strong>ID:</strong> {item.id}
                </p>
                <p>
                  <strong>Name:</strong> {item.name}
                </p>
                <p>
                  <strong>Description:</strong> {item.description}
                </p>
              </div>
            </div>
          ))}
      </div>

      {/* Footer or Loading State */}
      {loading && <div className="loading-footer">Loading more items...</div>}
    </div>
  );
};

export default App;
```

---

3. CSS File (`App.css`)

```css
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
  padding-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Loading Footer */
.loading-footer {
  text-align: center;
  padding: 10px;
  color: #007bff;
  font-weight: bold;
}
```

Explanation:

1. **Custom Hook (`useInfiniteScroll`)**:
   - Handles the scroll logic and API fetching in a reusable way.
   - Manages the `loading`, `items`, `error`, and `page` state internally.
   - Provides the `containerRef` which is used in the component to attach the scroll event listener.
2. **API Call**:
   - The custom hook makes an API call to the `apiUrl` endpoint with the given `page` and `limit` parameters.
   - It appends new items to the existing list instead of replacing them.
3. **Infinite Scroll Logic**:

- When the user scrolls to the bottom, the `handleScroll` function triggers a fetch for the next page of items.
- The scroll listener is cleaned up when the component unmounts to avoid memory leaks.

4. **Component (`App.tsx`):**
   - The `useInfiniteScroll` hook is used in the `App` component to manage the infinite scroll behavior.
   - The `containerRef` is attached to the scrollable container to detect when the user has scrolled to the bottom.
5. **CSS**:
   - The `.list-container` has a fixed height with `overflow-y: auto` to enable scrolling.
   - A loading footer message is displayed while fetching more data.

How to Run:

1. **Install Dependencies**: You'll need `axios` to make API calls:

   ```bash
   npm install axios
   ```

2. **Backend API**: Ensure your backend has a paginated API endpoint that supports `GET /api/items?page=1&limit=100` and returns the appropriate data.
3. **Start the React App**:

   ```bash
   npm start
   ```

Result:

- **Infinite Scroll**: As you scroll to the bottom of the list, the app will automatically fetch and append new items.
- **Reusable Hook**: The `useInfiniteScroll` hook encapsulates the logic for fetching and scrolling, making it reusable in other components.
- **Loading State**: Displays a loading message when fetching more items.

This solution encapsulates the infinite scroll logic into a custom hook, making your components cleaner and more reusable across your app.

this example with observer

To implement **infinite scroll** with the **Intersection Observer API**, you can replace the scroll detection logic with an observer that watches a target element, such as a "loading" or "footer" element. When this target element intersects (appears) in the viewport, a new API request is triggered to fetch more data.

Here's how you can refactor the infinite scroll example to use **Intersection Observer**:

Steps:

- Use the **Intersection Observer API** to observe when a target element (usually a loading spinner or a footer) comes into view.
- Trigger the API call when the target element is in view, allowing you to fetch the next set of data.

1. Custom Hook (`useInfiniteScrollWithObserver.ts`)

We will create a hook that uses the **Intersection Observer** to trigger the API call when the "loading" element becomes visible.

```tsx
import { useState, useEffect, useRef } from "react";
import axios from "axios";

// Define the type for an item
export interface Item {
  id: number;
  name: string;
  description: string;
}
```

```
// Define the API response structure
export interface ApiResponse {
  total: number;
  items: Item[];
}

const useInfiniteScrollWithObserver = (
  apiUrl: string,
  itemsPerPage: number
) => {
  const [items, setItems] = useState<Item[]>([]);
  const [totalItems, setTotalItems] = useState<number>(0);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
  const [page, setPage] = useState<number>(1); // Current page
  const observerRef = useRef<IntersectionObserver | null>(null);
  const loadingRef = useRef<HTMLDivElement>(null); // Target element to observe

  // Fetch data from the API
  const fetchData = async (page: number) => {
    if (loading) return; // Prevent multiple simultaneous API calls

    setLoading(true);
    setError(null);

    try {
      const response = await axios.get<ApiResponse>(apiUrl, {
        params: {
          page,
          limit: itemsPerPage,
        },
      });

      setItems((prevItems) => [...prevItems, ...response.data.items]); // Append new items
      setTotalItems(response.data.total);
    } catch (error) {
      setError("Failed to fetch data.");
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData(page); // Fetch initial data on component mount
  }, [page]);

  // Intersection Observer logic
  useEffect(() => {
    const handleIntersection = (entries: IntersectionObserverEntry[]) => {
      const entry = entries[0];
      if (entry.isIntersecting && !loading && items.length < totalItems) {
        setPage((prevPage) => {
          const nextPage = prevPage + 1;
          fetchData(nextPage); // Fetch next page of items
          return nextPage;
        });
      }
    };

    if (loadingRef.current) {
      observerRef.current = new IntersectionObserver(handleIntersection, {
        rootMargin: "100px", // Trigger observer a bit before the element is in view
        threshold: 1.0, // When 100% of the target element is in view
      });

      observerRef.current.observe(loadingRef.current); // Start observing the target
element
    }

    return () => {
      if (observerRef.current && loadingRef.current) {
        observerRef.current.unobserve(loadingRef.current); // Clean up observer on
unmount
      }
    };
  }, [loading, items.length, totalItems]);

  return {
    items,
    loading,
    error,
    loadingRef, // Reference to the loading indicator to observe
  };
};

export default useInfiniteScrollWithObserver;
```

---

2. Using the Custom Hook in the Component (`App.tsx`)

Now, let's use the custom hook in the component. It will automatically fetch more data when the "loading" element becomes visible due to the **Intersection Observer**.

```tsx
import React from "react";
import "./App.css";
import useInfiniteScrollWithObserver, { Item } from
"./useInfiniteScrollWithObserver";

// Define your API URL
const API_URL = "/api/items"; // Update this with your actual backend URL

const App: React.FC = () => {
  const itemsPerPage = 100;
  const { items, loading, error, loadingRef } =
useInfiniteScrollWithObserver(API_URL, itemsPerPage);

  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      {/* List Section */}
      <div className="list-container">
        {loading && <p>Loading...</p>}
        {error && <p style={{ color: "red" }}>{error}</p>}
        {!loading && !error && items.length === 0 && <p>No items found.</p>}
        {!loading &&
          !error &&
          items.map((item: Item) => (
            <div key={item.id} className="list-item">
              <input
                type="checkbox"
                id={`toggle-${item.id}`}
                className="toggle-input"
              />
              <label htmlFor={`toggle-${item.id}`} className="toggle-label">
                <strong>{item.name}</strong>
                <span>Show Details</span>
              </label>
              <div className="item-details">
                <p>
                  <strong>ID:</strong> {item.id}
                </p>
                <p>
                  <strong>Name:</strong> {item.name}
                </p>
                <p>
                  <strong>Description:</strong> {item.description}
                </p>
              </div>
            </div>
          ))}
      </div>

      {/* Loading Footer */}
      <div ref={loadingRef} className="loading-footer">
        {loading && <p>Loading more items...</p>}
      </div>
    </div>
  );
};

export default App;
```

---

3. CSS File (`App.css`)

The CSS file remains the same as before. However, we will add a specific class for the footer where the loading message will appear.

```css
```

```
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
  padding-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Loading Footer */
.loading-footer {
  text-align: center;
  padding: 10px;
  color: #007bff;
  font-weight: bold;
}
```

Explanation:

1. **Intersection Observer**:
   - The `IntersectionObserver` API watches the `loadingRef` target element (usually a loading spinner or footer) to detect when it becomes visible in the viewport.
   - When the target element is in view, the API call is triggered to load the next set of items.
2. **Custom Hook (`useInfiniteScrollWithObserver`)**:
   - The hook encapsulates the logic for infinite scroll using the `IntersectionObserver`.
   - It sets up the observer when the component mounts and cleans it up when the component unmounts.
3. **Component (`App.tsx`)**:
   - The `loadingRef` is used to reference the target element (e.g., a footer) that the observer will monitor.

    - When the footer comes into view, the hook triggers the API call to fetch more items.
4. **CSS**:
    - The `.loading-footer` class is used to style the footer element that will be observed.
    - It's where the "Loading more items..." message will appear while fetching data.

How to Run:

1. **Install Dependencies**: You'll need `axios` to make API calls:

    ```bash
    npm install axios
    ```

2. **Backend API**: Ensure your backend has a paginated API endpoint that supports `GET /api/items?page=1&limit=100` and returns the appropriate data.
3. **Start the React App**:

    ```bash
    npm start
    ```

Result:

- **Infinite Scroll**: As you scroll to the bottom of the list, the app will automatically fetch and append new items when the footer is in view.
- **Intersection Observer**: The `IntersectionObserver` API detects when the loading indicator (footer) is visible in the viewport and triggers the next page fetch.
- **Loading State**

this example with infinty scroll laibary

To implement **infinite scroll** with a popular library, such as `react-infinite-scroll-component`, we can make the process easier. This library automatically handles the scrolling detection and triggering of fetches when the user reaches the bottom of the page.

Here’s how you can implement **infinite scroll** using the `react-infinite-scroll-component` library in your React app with TypeScript:

1. Install Dependencies

First, install `react-infinite-scroll-component`:

```bash
npm install react-infinite-scroll-component axios
```

2. Create the Component (`App.tsx`)

In the component, we'll use `react-infinite-scroll-component` to automatically handle the infinite scrolling behavior and trigger data fetching as the user scrolls.

```tsx
import React, { useState, useEffect } from "react";
import axios from "axios";
import InfiniteScroll from "react-infinite-scroll-component";

// Define the type for an item
export interface Item {
  id: number;
  name: string;
  description: string;
}

// Define the API response structure
export interface ApiResponse {
  total: number;
  items: Item[];
}

const API_URL = "/api/items"; // Replace with your actual backend URL

const App: React.FC = () => {
  const itemsPerPage = 100;
  const [items, setItems] = useState<Item[]>([]);
  const [totalItems, setTotalItems] = useState<number>(0);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
  const [page, setPage] = useState<number>(1); // Current page

  // Fetch data from the API
  const fetchData = async (page: number) => {
    if (loading) return; // Prevent multiple simultaneous API calls

    setLoading(true);
    setError(null);

    try {
      const response = await axios.get<ApiResponse>(API_URL, {
        params: {
          page,
          limit: itemsPerPage,
        },
      });

      setItems((prevItems) => [...prevItems, ...response.data.items]); // Append new items
      setTotalItems(response.data.total);
    } catch (error) {
      setError("Failed to fetch data.");
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchData(page); // Fetch initial data on component mount
  }, [page]);

  // Infinite scroll component
  return (
    <div style={{ width: "320px", margin: "0 auto" }}>
      <InfiniteScroll
        dataLength={items.length}
        next={() => setPage((prevPage) => prevPage + 1)}
        hasMore={items.length < totalItems}
        loader={<p>Loading...</p>}
        endMessage={<p>No more items to load</p>}
        scrollThreshold={0.95} // Trigger fetch when 95% of the page is scrolled
      >
        {items.map((item) => (
          <div key={item.id} className="list-item">
            <input
              type="checkbox"
              id={`toggle-${item.id}`}
              className="toggle-input"
            />
            <label htmlFor={`toggle-${item.id}`} className="toggle-label">
              <strong>{item.name}</strong>
              <span>Show Details</span>
            </label>
            <div className="item-details">
              <p>
                <strong>ID:</strong> {item.id}
              </p>
              <p>
                <strong>Name:</strong> {item.name}
              </p>
              <p>
                <strong>Description:</strong> {item.description}
              </p>
            </div>
          </div>
        ))}
      </InfiniteScroll>

      {loading && <div className="loading-footer">Loading more items...</div>}
      {error && <div className="error-footer">{error}</div>}
    </div>
  );
};

export default App;
```

3. CSS (`App.css`)

Add the necessary styles, including for the loading spinner and toggling details.

```css
```

```
/* General Styles */
.list-container {
  max-height: 400px; /* Limit the container height */
  overflow-y: auto; /* Enable scrolling for the container */
  border: 1px solid #ccc;
  margin-bottom: 10px;
  padding-bottom: 10px;
}

.list-item {
  position: relative;
  padding: 10px;
  border-bottom: 1px solid #ddd;
}

.toggle-input {
  display: none; /* Hide the checkbox */
}

.toggle-label {
  display: flex;
  justify-content: space-between;
  align-items: center;
  cursor: pointer;
  user-select: none;
}

.toggle-label span {
  background-color: #007bff;
  color: white;
  padding: 5px 10px;
  border-radius: 3px;
  font-size: 0.9em;
  transition: background-color 0.2s ease;
}

.toggle-label span:hover {
  background-color: #0056b3;
}

.item-details {
  max-height: 0;
  opacity: 0;
  overflow: hidden;
  transition: all 0.3s ease;
}

/* Show details when checkbox is checked */
.toggle-input:checked ~ .item-details {
  max-height: 100px; /* Adjust based on content */
  opacity: 1;
  margin-top: 10px;
}

/* Loading Footer */
.loading-footer {
  text-align: center;
  padding: 10px;
  color: #007bff;
  font-weight: bold;
}

/* Error Footer */
.error-footer {
  text-align: center;
  padding: 10px;
  color: red;
  font-weight: bold;
}
```

Explanation of the Changes:

1. **Infinite Scroll Library (`react-infinite-scroll-component`):**
    - `InfiniteScroll` component is used to detect when the user has scrolled to the bottom of the page and automatically trigger the `next()` function to load more items.
    - `dataLength`: The current number of items being displayed.
    - `next()`: Function that increments the page number when the scroll reaches the bottom.

    - `hasMore`: A boolean indicating whether there are more items to load (we check if the current number of items is less than the total items).
    - `loader`: Displays a loading message or spinner when new data is being fetched.
    - `endMessage`: Displays a message when there are no more items to load.
2. **Fetch Data**:
    - The `fetchData` function fetches the data from the backend based on the current page and appends new items to the existing list.
3. **Error Handling**:
    - If there's an error during the API request, it is captured and displayed using the `error` state.
4. **Scroll Threshold**:
    - The `scrollThreshold` prop is set to `0.95`, meaning the fetch will trigger when 95% of the scrollable area has been reached.
5. **CSS**:
    - Basic styles to handle the layout and display of items and their details.
    - Loading and error footers are styled to show appropriate messages when fetching data or when an error occurs.

How to Run:

1. **Install Dependencies**: If you haven't already, install the necessary packages:

    ```bash
    npm install react-infinite-scroll-component axios
    ```

2. **Backend API**: Make sure your backend has a paginated API endpoint like `GET /api/items?page=1&limit=100` and returns the items in the appropriate format.
3. **Start the React App**:

    ```bash
    ```

```
npm start
```

Result:

- **Infinite Scroll**: When you reach the bottom of the list, the next set of items will automatically load.
- **Loader and End Messages**: While loading more items, a loading message will be displayed. Once all items are loaded, an end message will appear.
- **Reusable Component**: You can reuse the `InfiniteScroll` component across various parts of your application.

This setup simplifies the infinite scrolling functionality and handles the fetching of data when the user scrolls using the `react-infinite-scroll-component` library.