useEffect - State Life Cycle - React JS & Native

Filter by Topic

useEffect - State Life Cycle - React Native - Class

What are Life Cycle Methods?

Life cycle methods in React refer to a set of special methods that are automatically called at different stages of a component's life in a React application. These methods are part of the **class component lifecycle** in React, and they allow you to run specific code at particular points during the component's existence — from when it is first created (mounted) to when it is removed from the DOM (unmounted).

With the advent of **functional components** and **hooks**, life cycle methods are now managed through the useEffect hook in functional components. However, understanding the traditional life cycle methods is crucial for grasping how React components work.

Why are they Called "Life Cycle Methods"?

The term **"life cycle"** is used because these methods define the stages or phases in the life of a React component. Just like in biology, where an organism goes through different stages such as birth, growth, reproduction, and death, a React component also goes through various stages — initialization, updating, and unmounting.

In React, the "life cycle" refers to the timeline during which the component is mounted, updated, and unmounted within the application. These stages dictate

when certain actions need to be performed, such as initializing state, fetching data, handling changes, or cleaning up resources.

The life cycle can be broken down into three major phases:

1. **Mounting**: When the component is being created and inserted into the DOM for the first time.
2. **Updating**: When the component's state or props change, causing the component to re-render.
3. **Unmounting**: When the component is being removed from the DOM.

Each phase has associated methods (in class components) or hooks (in functional components) that help manage and control the behavior of the component during that phase.

Life Cycle Methods in Class Components

In class components, life cycle methods are divided into three main phases:

Mounting:

- **constructor()**: Called when the component is first created. It is used to initialize state and bind methods.
- **componentDidMount()**: Called immediately after the component is mounted (inserted into the DOM). It is often used for data fetching or setting up subscriptions.

Updating:

- **shouldComponentUpdate()**: Called before re-rendering when the component's state or props change. It allows you to optimize performance by preventing unnecessary renders.
- **componentDidUpdate()**: Called immediately after the component updates (re-renders). It can be used to respond to state or prop changes, such as making API calls based on new data.
- **getSnapshotBeforeUpdate()**: Called right before the changes from the render method are applied to the DOM. It's useful for capturing information

(like scroll positions) before the update happens.

Unmounting:

- **componentWillUnmount()**: Called just before the component is removed from the DOM. It is used for cleanup tasks such as invalidating timers, canceling network requests, or unsubscribing from services.

Error Handling:

- **componentDidCatch()**: This method is invoked if an error is thrown during the rendering phase, lifecycle methods, or in event handlers. It allows you to handle errors gracefully and display a fallback UI.

useEffect - State Life Cycle - React Native - HOOKS

Life Cycle in Functional Components with Hooks

With the introduction of hooks in React 16.8, functional components gained the ability to handle life cycle behavior using hooks like useEffect and useState. Though functional components do not have traditional life cycle methods like class components, the useEffect hook provides a way to replicate the behavior of life cycle methods.

- **Mounting**: The code inside useEffect(() => { /* side effect */ }, []) runs when the component is first mounted. It's analogous to componentDidMount() in class components.
- **Updating**: The code inside useEffect(() => { /* side effect */ }, [dependencies]) runs when any dependency in the array changes. It's similar to componentDidUpdate().
- **Unmounting**: Returning a function from useEffect allows you to clean up resources (like unsubscribing or clearing intervals) when the component is about to be unmounted, similar to componentWillUnmount().

The Importance of Life Cycle Methods

1. **Component Initialization**: Life cycle methods allow developers to initialize the component's state, fetch data, or set up subscriptions as soon as the component is mounted. This is vital for performance optimization, such as fetching data only once when the component is first loaded.
2. **Handling Updates**: As state or props change, life cycle methods help in managing how and when to re-render a component. This ensures that your application is efficient and avoids unnecessary renders, which can lead to performance issues.
3. **Cleanup**: Life cycle methods such as componentWillUnmount() or the cleanup function in useEffect are essential for cleaning up resources like timers, event listeners, or API calls that would otherwise cause memory leaks or bugs.
4. **Error Boundaries**: Methods like componentDidCatch() in class components provide a way to handle JavaScript errors within a component tree gracefully, preventing the entire application from crashing.
5. **Performance Optimization**: By using life cycle methods such as shouldComponentUpdate(), you can control when a component should re-render, improving the performance of your application.

The term **"life cycle methods"** refers to methods that define the different stages a React component goes through during its life. These stages are **mounting**, **updating**, and **unmounting**, and each has specific methods that allow developers to handle component behavior during those times. In class components, life cycle methods are explicitly defined, while in functional components, hooks like useEffect are used to replicate the same behavior. Life cycle methods are crucial for managing initialization, updates, resource cleanup, and error handling, all of which contribute to a well-performing and efficient React application.

useEffect - State Life Cycle - React Native - Coding Examples

useEffect - State Life Cycle - React Native - Coding Examples

Code Snippet:

Copy Code

```
// 1. Basic useEffect Hook
// Theory
// The useEffect hook runs a side effect after the component renders. By default,

// Example
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';

export default function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('Component rendered or count changed');
  });

  return (
    <View>
      <Text>{`You clicked ${count} times`}</Text>
      <Button title="Click Me" onPress={() => setCount(count + 1)} />
    </View>
  );
}

// Explanation
// The useEffect hook will run every time the component renders or the count state
// The second parameter ([]) is optional and is discussed in later examples.



// 2. useEffect with Dependency Array
// Theory
// When you pass a dependency array as the second argument to useEffect, it only r

// Example
import React, { useEffect, useState } from 'react';
import { Text, View, Button } from 'react-native';

export default function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
```

```
    console.log('Effect triggered only when count changes');
  }, [count]);

  return (
    <View>
      <Text>{`You clicked ${count} times`}</Text>
      <Button title="Click Me" onPress={() => setCount(count + 1)} />
    </View>
  );
}

// Explanation
// useEffect will now only run when the count state changes. If you click the butt


// 3. useEffect for Cleanup (Component Unmount)
// Theory
// The cleanup function in useEffect runs when the component unmounts or before th

// Example
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';

export default function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('Effect is running');

    return () => {
      console.log('Cleanup: Component is unmounting or count is changing');
    };
  }, [count]);

  return (
    <View>
      <Text>{`You clicked ${count} times`}</Text>
      <Button title="Click Me" onPress={() => setCount(count + 1)} />
    </View>
  );
}

// Explanation
// The useEffect cleanup function runs before the component is unmounted or before
// This is useful for cleaning up subscriptions, timers, or any other side effects
```

```
// 4. Fetching Data in useEffect
// Theory
// useEffect is commonly used to fetch data asynchronously when the component moun

// Example
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';

export default function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts')
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error(error);
      }
    };

    fetchData();
  }, []);

  return (
    <View>
      <Text>{data ? `First post title: ${data[0].title}` : 'Loading...'}</Text>
    </View>
  );
}

// Explanation
// useEffect runs once when the component mounts because we passed an empty depend
// We use an async function inside the useEffect to fetch data and update the stat



// 5. useEffect with Multiple Effects
// Theory
// You can have multiple useEffect hooks in a component to handle different side e

// Example
import React, { useEffect, useState } from 'react';
```

```
import { Text, View } from 'react-native';

export default function App() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState(null);

  // Effect to log count change
  useEffect(() => {
    console.log('Count updated:', count);
  }, [count]);

  // Effect to fetch user data
  useEffect(() => {
    const fetchUser = async () => {
      const response = await fetch('https://jsonplaceholder.typicode.com/users/1')
      const data = await response.json();
      setUser(data);
    };

    fetchUser();
  }, []);

  return (
    <View>
      <Text>{`Count: ${count}`}</Text>
      <Text>{user ? `User: ${user.name}` : 'Loading user...'}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
    </View>
  );
}

// Explanation
// The first useEffect listens to count changes, while the second fetches user dat
// This demonstrates that you can use multiple useEffect hooks for separate concer



// 6. useEffect for Timers
// Theory
// useEffect can be used to handle timers like setTimeout and setInterval.

// Example
import React, { useEffect, useState } from 'react';
import { Text, View, Button } from 'react-native';

export default function App() {
```

```
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(timer);
  }, []);

  return (
    <View>
      <Text>{`Timer: ${seconds} seconds`}</Text>
    </View>
  );
}

// Explanation
// useEffect sets up a timer that updates the seconds state every second.
// The cleanup function ensures that the interval is cleared when the component un



// 7. useEffect with Props Changes
// Theory
// useEffect can also be used to react to prop changes, updating state or performi

// Example
import React, { useEffect, useState } from 'react';
import { Text, View, Button } from 'react-native';

export default function App() {
  const [name, setName] = useState('John');

  useEffect(() => {
    console.log('Name prop updated:', name);
  }, [name]);

  return (
    <View>
      <Text>{`Name: ${name}`}</Text>
      <Button title="Change Name" onPress={() => setName('Alice')} />
    </View>
  );
}
```

```
// Explanation
// useEffect triggers whenever the name state changes, logging the updated name to



// 8. useEffect with Async Cleanup
// Theory
// If your useEffect function returns an async cleanup function, it needs to be ha

// Example
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';

export default function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts')
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error(error);
      }
    };

    fetchData();

    return () => {
      console.log('Cleanup: Cancel any active requests or subscriptions');
    };
  }, []);

  return (
    <View>
      <Text>{data ? `First post: ${data[0].title}` : 'Loading...'}</Text>
    </View>
  );
}

// Explanation
// In the cleanup function, we can cancel any active requests or subscriptions to
```

```
// 9. useEffect with Debounced Inputs
// Theory
// useEffect can be used with debounced inputs to limit how often certain operatio

// Example
import React, { useState, useEffect } from 'react';
import { TextInput, View, Text } from 'react-native';

export default function App() {
  const [query, setQuery] = useState('');
  const [debouncedQuery, setDebouncedQuery] = useState(query);

  useEffect(() => {
    const timer = setTimeout(() => {
      setDebouncedQuery(query);
    }, 500);

    return () => clearTimeout(timer);
  }, [query]);

  useEffect(() => {
    if (debouncedQuery) {
      console.log(`Searching for ${debouncedQuery}`);
      // API call or search logic here
    }
  }, [debouncedQuery]);

  return (
    <View>
      <TextInput
        value={query}
        onChangeText={setQuery}
        placeholder="Search"
      />
    </View>
  );
}

// Explanation
// The first useEffect debounces the input by waiting 500ms after the user stops t
// The second useEffect reacts to changes in the debounced value.



// 10. useEffect with Local Storage
// Theory
```

```
// useEffect can be used to store or retrieve data from local storage.

// Example
import React, { useEffect, useState } from 'react';
import { Text, View, Button } from 'react-native';

export default function App() {
  const [name, setName] = useState('');

  useEffect(() => {
    const savedName = localStorage.getItem('name');
    if (savedName) {
      setName(savedName);
    }
  }, []);

  const saveName = () => {
    localStorage.setItem('name', name);
  };

  return (
    <View>
      <Text>{`Saved Name: ${name}`}</Text>
      <Button title="Save Name" onPress={saveName} />
    </View>
  );
}

// Explanation
// The useEffect hook is used to load a saved name from local storage on mount.
// The saveName function stores the name in local storage when the button is press
```

useEffect - State Life Cycle - React JS - Coding Examples

state life cycle using useEffect in a **React JS** web application using async/await, try/catch, axios, and React-Vite.

Code Snippet:

Copy Code

```
// 1. Basic Example: Fetching Data Using useEffect with async/await
// Theory
// useEffect is used to handle side effects such as fetching data when the compone

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/pos
        setData(response.data);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      <h1>Data Fetch Example</h1>
      {data ? (
        <ul>
          {data.slice(0, 5).map(post => (
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}

export default App;

// Explanation
// The fetchData function is async and uses axios to fetch data.
// The useEffect hook ensures the data is fetched only once when the component is
// The try/catch block handles errors gracefully if the fetch fails.
```

```
// 2. Handling Errors with try/catch in useEffect
// Theory
// Using try/catch allows you to handle asynchronous errors in useEffect.
// If an error occurs during the API call or in any asynchronous operation, it can

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/pos
        setData(response.data);
      } catch (error) {
        setError('Failed to fetch data');
        console.error('Error:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      <h1>Data Fetch with Error Handling</h1>
      {error && <p>{error}</p>}
      {data ? (
        <ul>
          {data.slice(0, 5).map(post => (
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}
```

```
export default App;

// Explanation
// An error state is used to store any errors encountered during the fetch.
// If an error occurs, the error message is displayed on the UI.



// 3. Using useEffect to Handle API Calls Based on State Changes
// Theory
// You can trigger API calls in useEffect whenever certain state variables change.
// This is useful for making dynamic requests based on user input or other state c

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);

  useEffect(() => {
    const fetchResults = async () => {
      if (!query) return; // Don't make the request if the query is empty

      try {
        const response = await axios.get(`https://api.example.com/search?q=${query
        setResults(response.data);
      } catch (error) {
        console.error('Error fetching search results:', error);
      }
    };

    fetchResults();
  }, [query]);

  return (
    <div>
      <input
        type="text"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search"
      />
      <ul>
```

```
        {results.map(result => (
          <li key={result.id}>{result.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;

// Explanation
// useEffect listens to changes in the query state and triggers the API request wh
// If the query is empty, no request is made.



// 4. Debouncing API Requests Using useEffect
// Theory
// Debouncing is a technique used to delay an API request until the user stops typ
// This reduces the number of API calls.

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [query, setQuery] = useState('');
  const [debouncedQuery, setDebouncedQuery] = useState(query);
  const [results, setResults] = useState([]);

  useEffect(() => {
    const timer = setTimeout(() => {
      setDebouncedQuery(query);
    }, 500);

    return () => clearTimeout(timer); // Cleanup previous timer
  }, [query]);

  useEffect(() => {
    const fetchResults = async () => {
      if (!debouncedQuery) return;

      try {
        const response = await axios.get(`https://api.example.com/search?q=${debou
        setResults(response.data);
      } catch (error) {
```

```
        console.error('Error fetching debounced search results:', error);
      }
    };

    fetchResults();
  }, [debouncedQuery]);

  return (
    <div>
      <input
        type="text"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search"
      />
      <ul>
        {results.map(result => (
          <li key={result.id}>{result.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;

// Explanation
// The first useEffect sets up a timer to debounce the input.
// The second useEffect makes the API call when the debounced query changes.



// 5. Clean-up with useEffect
// Theory
// You can perform clean-up tasks in useEffect when a component unmounts.
// This is important to avoid memory leaks, especially when dealing with asynchron

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const controller = new AbortController();
```

```
    const fetchData = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/pos
          signal: controller.signal
        });
        setData(response.data);
      } catch (error) {
        if (error.name !== 'AbortError') {
          console.error('Error:', error);
        }
      }
    };

    fetchData();

    return () => {
      controller.abort(); // Clean up the request if the component unmounts
    };
  }, []);

  return (
    <div>
      <h1>Data Fetch with Cleanup</h1>
      {data ? (
        <ul>
          {data.slice(0, 5).map(post => (
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}

export default App;

// Explanation
// An AbortController is used to cancel the ongoing request if the component is un
// The cleanup function in useEffect aborts the request when the component unmount



// 6. Using async/await with Conditional Fetching
// Theory
```

```
// Sometimes, API requests should only be triggered under certain conditions. For

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState(null);
  const [fetchData, setFetchData] = useState(false);

  useEffect(() => {
    const fetchDataHandler = async () => {
      if (!fetchData) return;

      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/pos
        setData(response.data);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchDataHandler();
  }, [fetchData]);

  return (
    <div>
      <button onClick={() => setFetchData(true)}>Fetch Data</button>
      <div>
        {data ? (
          <ul>
            {data.slice(0, 5).map(post => (
              <li key={post.id}>{post.title}</li>
            ))}
          </ul>
        ) : (
          <p>No data to display</p>
        )}
      </div>
    </div>
  );
}

export default App;

// Explanation
```

```
// The fetchData state controls whether the data fetching should be triggered or n



// 7. Updating State After API Call
// Theory
// State can be updated based on the response from an API call to trigger re-rende

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/pos
        setPosts(response.data);
      } catch (error) {
        console.error('Error fetching posts:', error);
      }
    };

    fetchPosts();
  }, []);

  return (
    <div>
      <h1>Posts</h1>
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;

// Explanation
// The posts state is updated after the data is fetched, and the component re-rend
```

```
// 8. Handling Multiple API Calls Concurrently
// Theory
// Sometimes, you may need to trigger multiple API calls concurrently. Promise.all

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [users, setUsers] = useState([]);
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const [usersResponse, postsResponse] = await Promise.all([
          axios.get('https://jsonplaceholder.typicode.com/users'),
          axios.get('https://jsonplaceholder.typicode.com/posts')
        ]);

        setUsers(usersResponse.data);
        setPosts(postsResponse.data);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      <h1>Users and Posts</h1>
      <div>
        <h2>Users</h2>
        <ul>
          {users.map(user => (
            <li key={user.id}>{user.name}</li>
          ))}
        </ul>
      </div>
      <div>
        <h2>Posts</h2>
        <ul>
          {posts.map(post => (
```

```
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default App;

// Explanation
// Promise.all is used to fetch both users and posts concurrently. Both requests r



// 9. Handling Pagination with useEffect
// Theory
// Pagination is a common use case in applications that load large datasets. Here'

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [page, setPage] = useState(1);
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const response = await axios.get(`https://jsonplaceholder.typicode.com/pos
        setPosts(response.data);
      } catch (error) {
        console.error('Error fetching posts:', error);
      }
    };

    fetchPosts();
  }, [page]);

  return (
    <div>
      <h1>Paginated Posts</h1>
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.title}</li>
```

```
          ))}
        </ul>
        <button onClick={() => setPage(prev => prev + 1)}>Next Page</button>
      </div>
    );
}

export default App;

// Explanation
// The page state controls which page of posts is displayed.
// Each time page changes, a new API call is made.



// 10. Handling Authorization Token in API Requests
// Theory
// If the API requires authentication, you often need to pass a token in the heade

// Example
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState([]);
  const token = 'your-authorization-token'; // Replace with actual token

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://api.example.com/protected', {
          headers: {
            Authorization: `Bearer ${token}`
          }
        });
        setData(response.data);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
```

```
        <h1>Protected Data</h1>
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      </div>
    );
}

export default App;

// Explanation
// The Authorization header is passed in the axios request to authenticate the req
```

FlatList & ScrollView in React Native: Loops

FlatList and ScrollView are used to display a list of items, and their outputs can appear the same in terms of scrollable content.

However, they have significant differences in performance and intended use cases. Here's a breakdown of how they compare:

1. **Output (Visual Display):**

- **ScrollView**:
- Displays a list of items that can be scrolled vertically or horizontally.
- Suitable for smaller lists or cases where you want to display a relatively fixed set of content.
- All child components are rendered at once, regardless of whether they are visible on the screen or not.
- **FlatList**:
- Also displays a list of items that can be scrolled vertically or horizontally.
- Suitable for large datasets or dynamic lists.
- Only renders the items that are currently visible on the screen, and it efficiently renders new items as the user scrolls (this is known as "lazy loading" or "windowing").

2. Performance:

- **ScrollView**:
- **Not optimized for large lists**: ScrollView renders all its child components immediately, regardless of whether they are currently visible on the screen or not.
- This can lead to performance issues when dealing with long lists, as all items (and their components) are in memory, which increases the rendering time and memory usage.
- **FlatList**:
- **Optimized for large lists**: FlatList only renders the visible items and batches the rendering of additional items as the user scrolls. It is optimized for large datasets and provides smoother performance.
- It also supports features like item recycling (where invisible items are reused when scrolling back up), automatic handling of loading more items, and handling of separators between items.

3. Use Cases:

- **ScrollView**:
- Best for smaller lists or a small number of child components where performance is not a concern.
- Suitable when you need to display a relatively static layout with a few elements or when the list doesn't grow dynamically.
- **FlatList**:
- Ideal for large or dynamic lists where items might change, be added, or removed frequently.
- Best suited for cases where you need to handle large datasets efficiently, like displaying posts, messages, comments, or product lists.

4. Features:

- **ScrollView**:
- Basic scrolling behavior.
- No built-in optimizations for large lists or items.
- **FlatList**:
- Supports features like:
- **keyExtractor**: Automatically handles unique keys for each item.
- **renderItem**: A custom function to render each item.
- **onEndReached**: Triggered when the user scrolls to the end, allowing for infinite scrolling or loading more data.
- **initialNumToRender**: The number of items to render initially.

- **getItemLayout**: Helps with fast scrolling by providing a way to pre-compute the layout of the list items.
- **ListHeaderComponent** and **ListFooterComponent**: Used for adding headers or footers to the list.

Code Snippet:

Copy Code

```
// FlatList

import React, { useEffect, useState } from 'react';
import { Text, View, FlatList } from 'react-native';

export default function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, []);

  const renderItem = ({ item }) => (
    <View key={item.id} style={{ marginBottom: 10 }}>
      <Text style={{ fontWeight: 'bold' }}>Title: {item.title}</Text>
      <Text>Body: {item.body}</Text>
    </View>
  );

  return (
    <FlatList
      data={data}
      renderItem={renderItem}
      keyExtractor={item => item.id.toString()}
    />
  );
}
```

```
// ScrollView
import React, { useEffect, useState } from 'react';
import { Text, View, ScrollView } from 'react-native';

export default function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, []);

  return (
    <ScrollView>
      <View>
        {data.map(post => (
          <View key={post.id} style={{ marginBottom: 10 }}>
            <Text style={{ fontWeight: 'bold' }}>Title: {post.title}</Text>
            <Text>Body: {post.body}</Text>
          </View>
        ))}
      </View>
    </ScrollView>
  );
}


// Using Traditional for Loop (ES-5)
import React, { useEffect, useState } from 'react';
import { Text, View, ScrollView } from 'react-native';

export default function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts')
```

```
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error(error);
      }
    };

    fetchData();
  }, []);

  const renderPosts = () => {
    const posts = [];
    if (data) {
      for (let i = 0; i < data.length; i++) {
        posts.push(
          <View key={data[i].id} style={{ marginBottom: 10 }}>
            <Text style={{ fontWeight: 'bold' }}>Title: {data[i].title}</Text>
            <Text>Body: {data[i].body}</Text>
          </View>
        );
      }
    }
    return posts;
  };

  return (
    <ScrollView>
      <View>
        {data ? renderPosts() : <Text>Loading...</Text>}
      </View>
    </ScrollView>
  );
}


// Using Traditional Loop: forEach()
import React, { useEffect, useState } from 'react';
import { Text, View, ScrollView } from 'react-native';

export default function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts')
```

```
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error(error);
      }
    };

    fetchData();
  }, []);

  const renderPosts = () => {
    const posts = [];
    if (data) {
      data.forEach(post => {
        posts.push(
          <View key={post.id} style={{ marginBottom: 10 }}>
            <Text style={{ fontWeight: 'bold' }}>Title: {post.title}</Text>
            <Text>Body: {post.body}</Text>
          </View>
        );
      });
    }
    return posts;
  };

  return (
    <ScrollView>
      <View>
        {data ? renderPosts() : <Text>Loading...</Text>}
      </View>
    </ScrollView>
  );
}
```