SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

Course Code and Name : 19TS601 FULL STACK DEVELOPMENT

Unit 2 : REACT

Topic : Async State Initialization in React.js

1.

Async State Initialization

- In a synchronous system, you wait for one person to finish serving themselves before you start.
- In an asynchronous system, everyone serves themselves at the same time without waiting.
- Asynchronous state in React refers to the concept of initializing or updating a component's state without blocking other operations. This means the state can be set or updated after performing tasks like fetching data from an API, reading files, or waiting for user input, all while the rest of the application continues to run smoothly.

1.

Async State Initialization

- Asynchronous state initialization in React.js can be managed in multiple ways depending on the complexity of the application and the tools being used.
- Here are some common approaches:
- Using useState with an Initializer Function
- Using Redux with Async Thunks
- Server-Side Rendering (SSR) or Static Site Generation (SSG)
- Combining State Libraries with React Query
- Asynchronous Middleware in Redux

Using ’useState' with an Initializer Function

- React's useState allows to initialize state with a function, which is useful for asynchronous logic.
- However, since useState does not directly support async functions, you typically use it alongside useEffect.
- Example:jsx

```
import React, { useState, useEffect } from 'react';
const MyComponent = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('/api/data');
      const result = await response.json();
      setData(result);
      setLoading(false);
    };
```

```
  fetchData();
  }, []);

  if (loading) return <div>Loading...</div>;

  return <div>{JSON.stringify(data)}</div>;
};
```

- Use useEffect to handle side effects like fetching data asynchronously.
- Manage loading and error states explicitly.

Using Redux with Async Thunks

- For more complex state management needs, Redux Toolkit provides createAsyncThunk, which simplifies handling asynchronous state initialization.
- Example:js

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
// Define an async thunk
export const fetchInitialData = createAsyncThunk(
  'data/fetchInitialData',
  async () => {
    const response = await fetch('/api/data');
    return response.json();
  }
);
```

```
// Create a slice
const dataSlice = createSlice({
  name: 'data',
  initialState: { data: null, status: 'idle', error: null },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchInitialData.pending, (state) => {
        state.status = 'loading';
      })
```

```
.addCase(fetchInitialData.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.data = action.payload;
      })
      .addCase(fetchInitialData.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  },
});
export default dataSlice.reducer;
```

- Use createAsyncThunk for API calls and manage loading/error states automatically.
- Dispatch the thunk during app initialization to load the initial state

Server-Side Rendering (SSR) or Static Site Generation (SSG)

- For frameworks like Next.js, asynchronous data fetching can be done server-side using functions like getServerSideProps or getStaticProps.
- The fetched data is passed as initial props to the React component.
- Example:js

```
export async function getServerSideProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();
  return { props: { initialData: data } };
}
const MyPage = ({ initialData }) => {
  return <div>{JSON.stringify(initialData)}</div>;
};
export default MyPage;
```

- This approach avoids loading spinners on the client side as data is pre-fetched

Combining State Libraries with React Query

- React Query or similar libraries can simplify managing asynchronous state by handling caching, deduplication of requests, and background updates.
- Example:js

```
import { useQuery } from '@tanstack/react-query';
const MyComponent = () => {
  const { data, error, isLoading } = useQuery(['fetchData'], async () => {
    const res = await fetch('/api/data');
    return res.json();
  });
  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;
  return <div>{JSON.stringify(data)}</div>;
};
```

React Query abstracts much of the boilerplate for managing async states and retries

Asynchronous Middleware in Redux

- Sometimes you may need to initialize the entire Redux store asynchronously. This can be achieved by dispatching an action after creating the store.
- Example: Javascript
- store.dispatch(fetchInitialData());

Choosing the Right Approach

- Use React's built-in hooks (useState, useEffect) for simple cases.
- Adopt Redux Toolkit for scalable applications requiring centralized state management.
- Leverage React Query or similar libraries for efficient server-state management.
- Opt for SSR/SSG in frameworks like Next.js for seamless user experience without client-side delays.

Event Handling Communicating from Child to Parent

- To enable communication from a child to a parent component in React, you can follow these steps using callback functions:
- Steps for Child-to-Parent Communication
- Define State in the Parent Component:
  - Create a state in the parent component to store the data received from the child.
- Pass a Callback Function to the Child:
  - Define a function in the parent component that updates its state and pass it as a prop to the child component.
- Invoke the Callback in the Child Component:
  - Call the passed function inside the child component and pass the data as an argument.

Example Implementation

- Parent Component (ParentComponent.js):

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';
const ParentComponent = () => {
  const [message, setMessage] = useState('Hello from Parent');
  const handleDataFromChild = (data) => {
    setMessage(data); // Update state with data from child
  };
```

Example Implementation

```
  return (
    <div>
      <h1>Message: {message}</h1>
      <ChildComponent sendDataToParent={handleDataFromChild} />
    </div>
  );
};

export default ParentComponent;
```

Child Component (ChildComponent.js):

```
import React from 'react';
const ChildComponent = ({ sendDataToParent }) => {
  const sendData = () => {
    const data = 'Hello from Child!';
    sendDataToParent(data); // Invoke callback with data
  };
```

```
return (
    <div>
      <button onClick={sendData}>Send Data to Parent</button>
    </div>
  );
};

export default ChildComponent;
```

How It Works

- The parent component defines a state (message) and a callback function (handleDataFromChild).
- The callback is passed to the child as sendDataToParent.
- When the button in the child is clicked, sendDataToParent is called with a message, updating the parent's state.

Key Points

- React follows unidirectional data flow (parent-to-child), so to send data back, you "lift state up" using callbacks.

- This approach ensures that the parent has control over its state while allowing dynamic updates triggered by child components.

1.What is async state initilization?

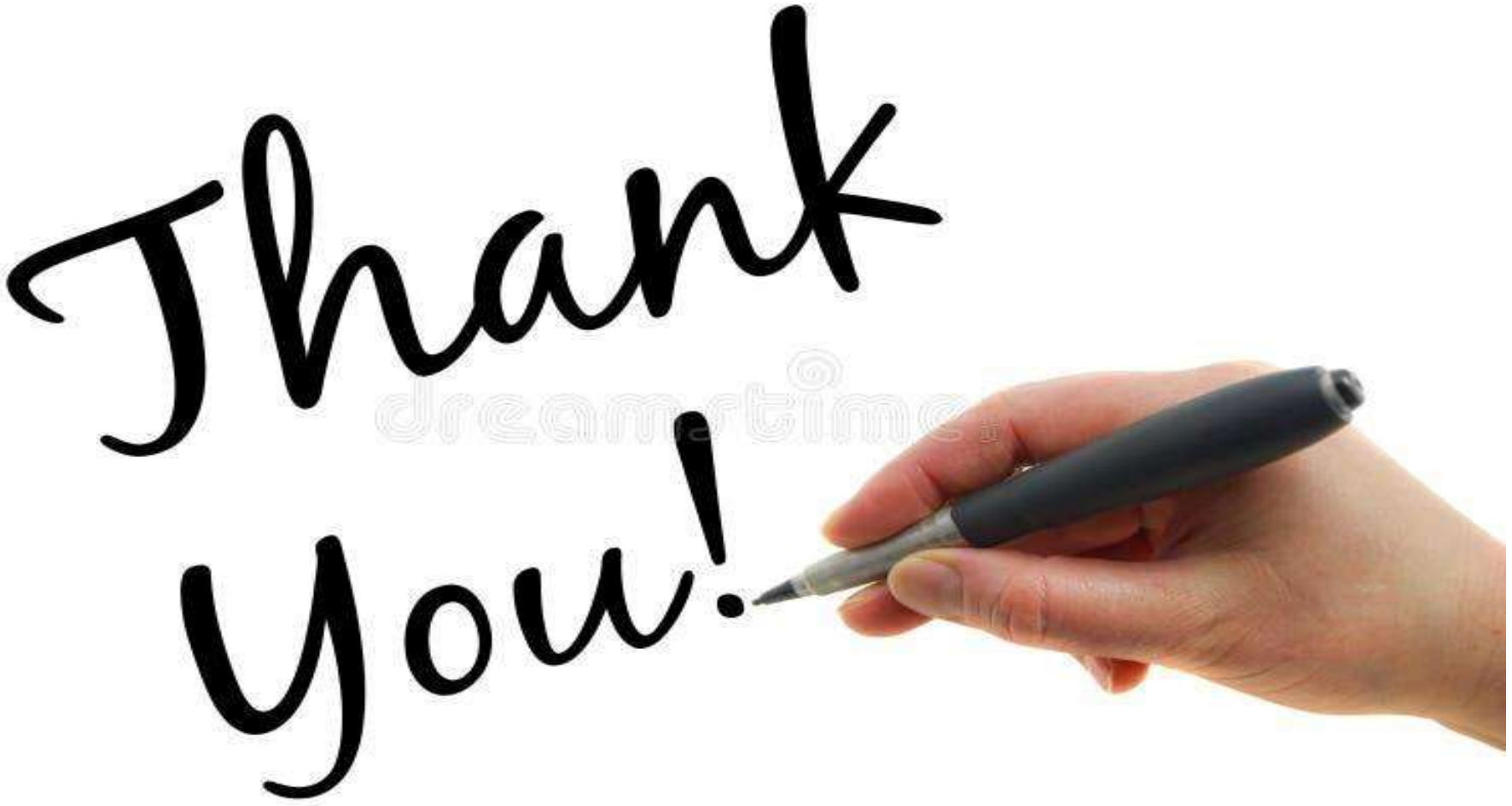
Text Book:

1.Pro MERN Stack, Full Stack Web App Development with Mongo, Express, React, and Node, Vasan Subramanian, A Press Publisher, 2019.

Reference:

David Flanagan, “Java Script: The Definitive Guide”, O’Reilly Media, Inc, 7 th Edition, 2020

2. Matt Frisbie, “Professional JavaScript for Web Developers” Wiley Publishing, Inc, 4th Edition, ISBN: 978-1-119-36656-0, 2019


Thank
You!