React Hooks

Introduction

What are hooks?

Hooks are a new addition in React 16.8. They are functions that let you manage the internal state of component and handle post rendering side effects.
Note: Before we continue, note you can try Hooks in a few components without rewriting any existing code and hooks are 100% backwards-compatible. They don't contain any breaking changes.

Motivation to use hooks

Hooks can be used to address the following problems caused by using class based components:

1. **Hard to reuse stateful logic between components**
   Hooks provide a way to separate stateful logic from a component, which allows for independent testing and reusability. With Hooks, you can reuse this logic in multiple components without restructuring the component hierarchy. This flexibility makes it easy to share Hooks with others and promote community collaboration.

2. **Complex components become difficult to understand**
   In class-based components, related code often gets mixed with unrelated code, which can lead to bugs and inconsistencies. Hooks address this issue by allowing you to break down a component into smaller functions that handle related pieces of logic, like setting up a subscription or fetching data. This approach is more intuitive than forcing a split based on lifecycle methods.

3. **Classes can be confusing at times**
   Classes in React can be a significant hurdle for learning the framework. Not only do you have to understand how "this" works in JavaScript, which can be quite different from other languages, but they can also complicate code reuse and organization. Hooks provide a simpler way to use React's features without relying on classes. By embracing functions, Hooks allow for a more intuitive and functional programming style that is closer to the conceptual nature of React components.

Rules of Hooks

The two main rules of Hooks in React are:

1. **Only call Hooks at the top level:**

   This rule states that Hooks should only be called at the top level of a function component or another custom Hook. They should not be called inside loops, conditions, or nested functions, as this can lead to unexpected behavior and bugs.

   Example 1: **Incorrectly** using hooks inside a function

```
const showAlert = () => {

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

}
```

   Example 2: **Correctly** using hooks at the top level of the component

```
useEffect(() => {

     alert("Dependency has changed");

}, [dependency]);
```

2. **Only call Hooks from React function components:**

   This rule states that Hooks can only be used in React function components or other custom Hooks. They should not be used in class components or regular JavaScript functions, as this can cause errors or crashes.

   Example 1: **Incorrectly** using hooks inside a class based component

```
class Example extends Component{

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

}
```

   Example 2: **Correctly** using hooks inside a functional component

```
const Example = () => {

      const [dependency, setDependency] = useState("");

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

};
```

Adhering to these rules ensures that Hooks work as intended and can help to prevent common issues and errors when working with React.

Order of Hooks

In React, hooks are executed in the order in which they are written. The order of Hooks is important. The order in which Hooks are called within a component must always be the same, so that React can correctly associate state and props with each Hook.

State in Function & Class based components

State management in functional and class-based React components is essentially the same, but the syntax and implementation details differ.

Syntax:

Class-based components have lifecycle methods, such as **componentDidMount** and **componentDidUpdate**, which allow for fine-grained control over the state and behavior of the component. Functional components have **hooks** that let you hook into the state and lifecycle features of the component but the syntax and implementation are different.

Updates and Side effects:

In class-based components, state is updated via the **setState** method. In functional components with hooks, state is updated via the update function returned by the **useState** hook. Side effects in functional components are managed using the **useEffect** hook which is a replacement for the lifecycle methods used in class-based components, such as **componentDidMount** and **componentDidUpdate**.

Boilerplate:

Class-based components require more boilerplate code to manage state and lifecycle methods, which can make the code more verbose and harder to read. Functional components with hooks have less boilerplate code, making them more concise and easier to read.

The useState hook

useState is a React Hook that lets you add a state variable to your component.

Parameters

1. **Intial State:** The **useState** hook in React takes an initial state as a parameter. The initial state can be a value of any type. If a function is passed as the initial state, it will be treated as an initializer function. The initializer function should be a pure function, taking no arguments, and returning a value of any type. React will call the initializer function when initializing the component and store the return value as the initial state.
   The initial state is only used during the first render and subsequent calls to useState with a new initial state will override the previous initial state.

Returns

The **useState** hook returns an array with exactly two values:

1. The current state. During the first render, it will match the initialState you have passed.
2. The set function that lets you update the state to a different value and trigger a re-render.

Example: Usage useState hook

```
const [count, setCount] = useState(0);
```

This code snippet uses the **useState** hook to define a state variable named **count** with an initial value of 0, and a function named **setCount** that can be used to update the state.
The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

Example 1: Basic usage of state setter function

```
const [count, setCount] = useState(0);
setCount(1)
```

Example 2: Passing a callback function to set the state

```
const [count, setCount] = useState(0);
setCount((prevCount) => prevCount + 1);
```

Note: Using the state setter function with a callback allows you to access the latest state value at the time of the update and perform calculations or updates based on that value. This ensures that the state is always updated correctly and consistently, regardless of the timing of the updates.

The useEffect hook

useEffect is a React Hook that lets you synchronize a component with an external system.

Parameters

1. **Setup:** The **useEffect** hook in React takes a function as its first argument, which contains the logic of your effect. This function may also return a cleanup function. When your component is initially added to the DOM, React will execute the setup function you provided in the useEffect hook. On subsequent re-renders React will first call the cleanup function with the previous values. After that, React will run your setup function with the new values.
   Finally, when your component is removed from the DOM, React will execute your cleanup function one last time. This ensures that your component's side effects are properly added and removed throughout its lifecycle.

2. **Options (dependencies):** The **useEffect** hook in React takes a dependency array as its second argument, which contains all the reactive values referenced inside the setup code. These values include props, state, and all variables and functions declared directly in the component body. The list of

dependencies must have a constant number of items and be written inline in the form of **[dep1, dep2, dep3]**.
React compares each dependency with its previous value using a comparison algorithm. If you don't provide the dependency array, your effect will run after every re-render of the component.

Returns

The **useEffect** hook does not return an value.

Example 1: Usage of useEffect hook

```
  useEffect(() => {
   setInterval(() => {
     setTimer((prev) => prev++);
   }, 1000);
 }, []);
```

This code snippet uses the **useEffect** hook to set an interval that increments the state value of timer after every second. The effect runs only once on mount due to an empty dependency array.

Example 2: Usage of useEffect hook (with a cleanup function)

```
 useEffect(() => {
   const interval = setInterval(() => {
     setTimer((prev) => prev++);
   }, 1000);

   return () => clearInterval(interval)
 }, []);
```

This code snippet uses the **useEffect** hook to create a timer that updates every second. It sets up an interval to update the timer and clears it on unmount using the cleanup function. The effect runs only once on mount due to an empty dependency array.

The useReducer hook

useReducer is a React Hook that lets you add a reducer to your component. It is typically used when you have complex state transitions that involve multiple sub-values or when the next state depends on the previous state.

It is a more powerful alternative to the useState hook and is particularly useful when managing state for large or deeply nested objects. The **useReducer** hook provides a simple API for dispatching actions and updating state in a predictable way.

Parameters

1. **reducer:** In React, the **useReducer** hook takes a pure reducer function as its first argument, which defines how the state gets updated. The reducer function should take in the current state and an action as arguments and return the new state. The state and action can be of any type.

2. **initialState:** The value that represents the initial state of the component. This can be any value, including an object or an array.

Returns

useReducer returns an array with exactly two values:

1. The current state. During the first render, it’s set to the initialState.
2. The dispatch function that lets you update the state to a different value and trigger a re-render.

Example: Usage of useReducer hook

```
 const [state, dispatch] = useReducer(reducer, initialState);
```

This code snippet uses the **useReducer** hook to define a state variable named **state** with an initial value of **initialState**, and a function named **dispatch** that can be used to dispatch updates to the state.

The dispatch function

The **dispatch** function returned by **useReducer** lets you update the state to a different value and trigger a re-render. You need to pass the **action** as the only argument to the dispatch function

Example:

```
const [timer, dispatch] = useReducer(reducer, initialState)

const handleIncrement = () => {
  dispatch({ type: "INCREMENT_COUNT" });
};
```

This code snippet uses the **dispatch** function from the **useReducer** hook and passes an **action** object of type "INCREMENT_COUNT". The reducer function then checks this action type to update the state of the timer.

Writing the reducer function

The **reducer** function used in useReducer hook of React is a pure function that takes the current state and an action as arguments, and returns the new state.

The reducer function evaluates the type of the action and updates the state based on the type of action.

Example:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "incremented_age": {
      return {
        name: state.name,
        age: state.age + 1,
      };
    }
```

```
      case "changed_name": {
        return {
          name: action.nextName,
          age: state.age,
        };
      }
      default:
        return state;
    }
```

Custom hooks

Custom hooks are functions in React that allow you to reuse stateful logic across multiple components. They follow the naming convention of starting with the word "use" and can be defined and used in the same way as the built-in hooks provided by React.

Custom hooks are a way to abstract and share logic that is not tied to a specific component, which makes your code more modular, easier to read and maintain. They can encapsulate complex stateful logic and make it easy to use in multiple components, without having to repeat the same code in each component.

Custom hooks can use other built-in hooks and can also be composed with other custom hooks, which makes it easy to create complex and reusable logic that can be used across different parts of your application.

Example: The **useLocalStorage** custom hook

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

const useLocalStorage = (key, initialValue) => {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue !== null ? JSON.parse(storedValue) : initialValue;
  });
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
};
```

This custom hook allows you to store and retrieve data in the browser's localStorage. It takes a key and initial value as arguments and returns a state **value** and a **setValue** function to update it.

Summary

We have discussed the advantages of using hooks in functional components instead of relying on class-based lifecycle methods. The three main hooks covered were **useState**, **useEffect**, and **useReducer**, with useState used for state management, useEffect for handling side effects, and useReducer for more complex state management. These hooks are a newer addition to React and make managing stateful logic in functional components easier.

Summarising it

Let's summarise what we have learned in this Lecture:

- What are hooks?
- Motivation to use hooks.
- Rules of hooks.
- Order of hooks.
- State in function and class based components.
- The useState hook and usage
- The useEffect hook and usage
- The useReducer hook and usage
- Custom hooks in React

Some References:

- Lifecycle methods vs Hooks in React: link
- Hooks in React: link
- Custom hooks: link

Introduction to Firebase

Storing Data

Why does data get lost after refresh?

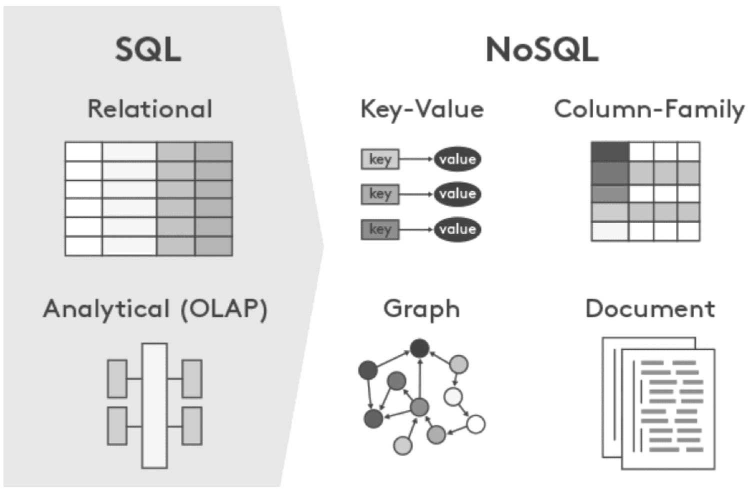
Here, we are storing the blog's data inside of the state locally in the form of an array. When you add a new blog, it gets added to the blogs array as well. But, when the page is reloaded, the App gets rerendered, and this array gets re-initialized to the empty array. So, This acts as temporary storage where data is not saved after refresh.

```
const [formData, setformData] = useState({title:"", content:""})
const [blogs, setBlogs] =  useState([]);
```

Using Databases

A database is an organized collection of data for easy access, management and updating. To save stored data even after the refresh, you need to connect your React App with some external database. Databases can be classified into two categories:

- SQL Databases or Relational Databases
- No SQL Databases

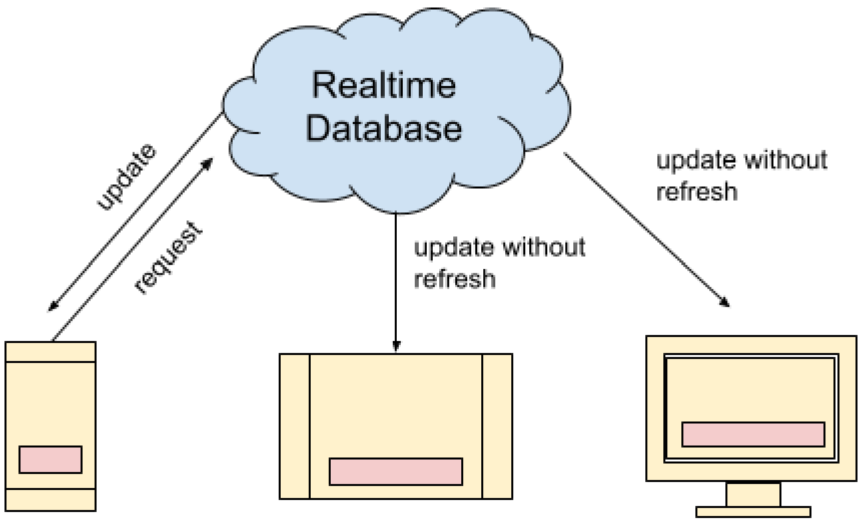


Properties	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Primary Purpose	General purpose and best suitable for structured, semi-structured, and unstructured data.	best suitable for structured data. Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Examples	Oracle, MySQL, Microsoft SQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB,

	Server, and PostgreSQL	Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune

Realtime Database

The Realtime database helps our users collaborate. It ships with mobile and web SDKs, allowing us to build our app without needing servers. When our users go offline, the Real-time Database SDKs use a local cache on the device for serving and storing changes. The local data is automatically synchronized when the device comes online.



Firebase

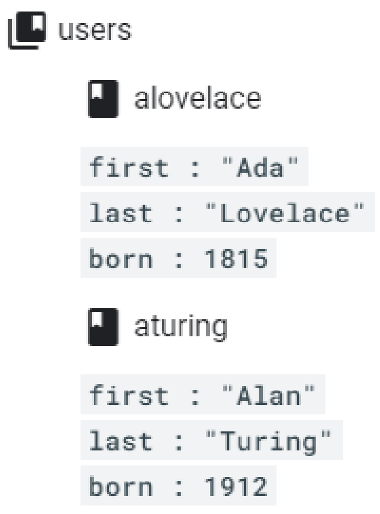
The Firebase Realtime Database is a cloud-hosted database in which data is stored as JSON. The data is synchronized in real-time to every connected client. Clients share one Realtime Database instance and automatically receive updates with the newest data when we build cross-platform applications with iOS and JavaScript SDKs. Firebase offers two cloud-based, client-accessible database solutions that support real-time data syncing:

- **Cloud Firestore** is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database. Data is stored in document format.

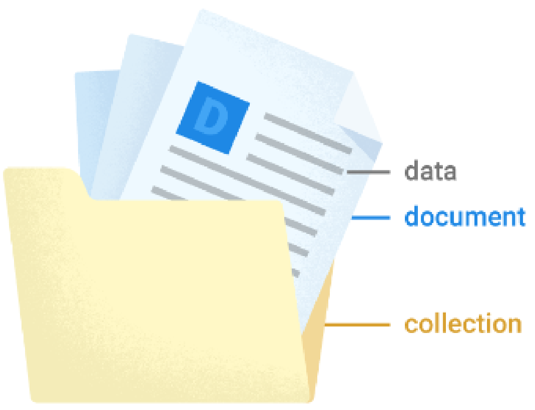
- **Realtime Database** is Firebase's original database. It's an efficient, low-latency solution for mobile apps requiring real-time synced states across clients. Data is stored in JSON format.

Cloud Firestore

In Cloud Firestore, the unit of storage is the document. A document is a lightweight record containing fields that map to values. Each document is identified by a name. Each document includes a set of key-value pairs. Cloud Firestore is optimized for storing extensive collections of small documents. Documents live in collections, which are simply containers for documents.
For example, you could have a users collection to contain your various users, each represented by a document:



```
users
    alovelace
    first : "Ada"
    last : "Lovelace"
    born : 1815
    aturing
    first : "Alan"
    last : "Turing"
    born : 1912
```

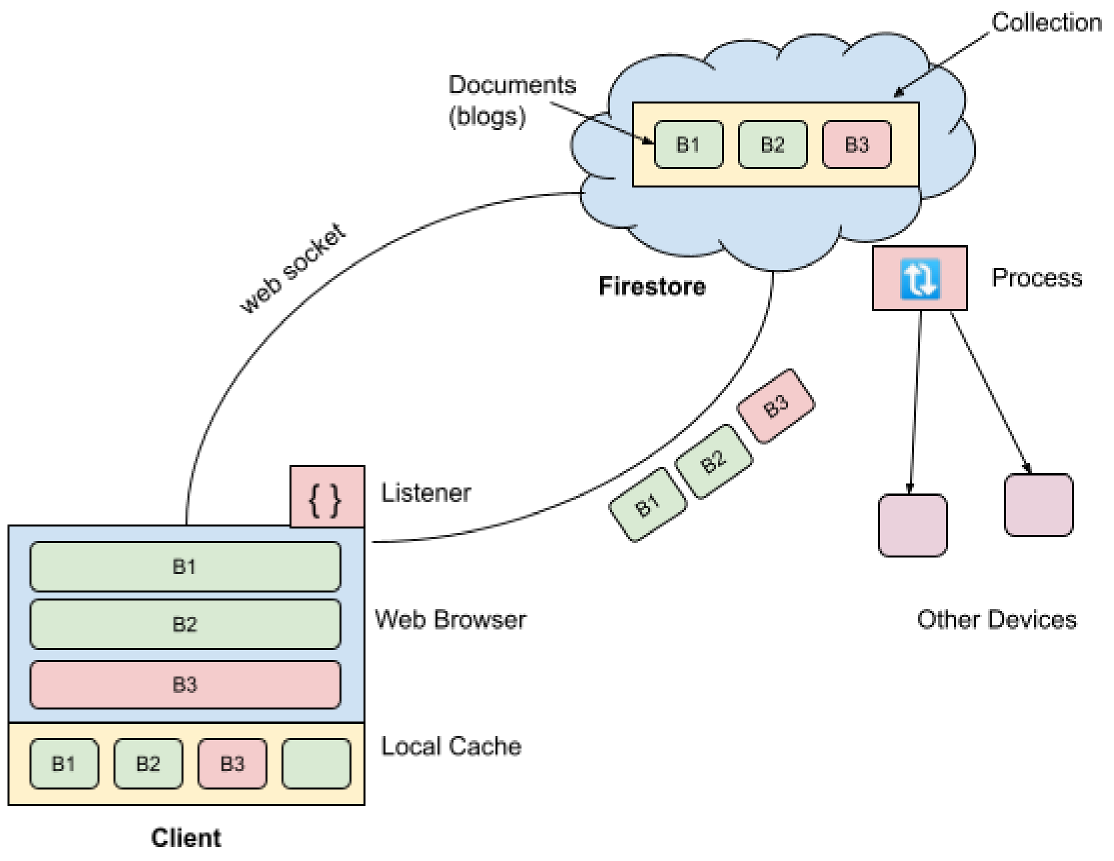


Data types that Cloud Firestore supports are Array, Boolean, Bytes, Date and time, Floating-point number, Geographical point, Integer, Map, Null, Reference and a Text string.

Understanding the working

Cloud Firestore caches data that your app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device returns online, Cloud Firestore synchronizes any local changes back to Cloud Firestore. To keep data in your apps current without retrieving your entire database each time an update happens, **real-time listeners** are added. Adding real-time listeners to your

app notifies you with a data snapshot whenever the data your client apps are listening to changes, retrieving only the new changes.
For Example, Your firebase has a collection of blogs with blogs B1 and B2 as documents. As soon as the client opens the app, a persistent connection will be established between the firestore and the client via web socket. On the client side, the listeners installed, which are nothing but a call-back function, listen to any changes happening to the client. Similarly, there is a process inside cloud firestore, which listens to any changes happening in the database. These listeners are used to notify changes in the apps.
When we open the app for the first time, it is not directly updated to the UI. First, the data gets stored inside the local cache of the device. Here B1 and B2 will be stored already in the local cache. When a new document B3, is added, it will be added to the local cache, and then the listener will be notified. The Listener will send all the data from the local cache to the firebase, including the changes. Now, the Process present in firebase gets notified. The Process will notify all the other devices about the changes, and changes will get updated for all the devices. Only the new data or changes get updated.



Using Firestore in your Application

For more detailed steps, you can check this link.

Create a Cloud Firestore Database

1. In the Firebase console, click **Add project**, then follow the on-screen instructions to create a Firebase project. Enter a project name, then click Continue. Select your Firebase account from the dropdown or click Create a new account if you don't already have one. Click Continue once the process completes.
2. Next, click the Web icon (</>) towards the top-left of the following page to set up Firebase for the web. Enter a nickname for your app in the provided field. Then click the Register app.
3. Copy the generated code and keep it for the following step (discussed in the following section). Click Continue to the console.
4. Navigate to the **Cloud Firestore** section of the Firebase console. Now, Follow the database creation workflow. Select a starting mode for your Cloud Firestore Security Rules:
   - **Test mode**
     Good for getting started with the mobile and web client libraries, but allows anyone to read and overwrite your data.
   - **Locked mode**
     Denies all reads and writes from mobile and web clients. Your authenticated application servers (C#, Go, Java, Node.js, PHP, Python, or Ruby) can still access your database.
5. Select a location for your database.
6. Click **Done**.

Initialize Firebase in Your React App

1. Install Firebase using npm:

```
npm install firebase
```

2. Create a firebaseinit.js file and paste the code generated earlier into this file. You can also find this code in Project Overview > Project Settings.
3. Replace the TODOs with your app's Firebase project configuration
4. Export the firebase db object from the file and import this object into the files where it is needed.

Adding Data to Firebase

Add a document

When you use set() to create a document, you must specify an ID for the document to create. For example:

```
import { doc, setDoc } from "firebase/firestore";

await setDoc(doc(db, "cities", "new-city-id"), data);
```

But sometimes there isn't a meaningful ID for the document, and it's more convenient to let Cloud Firestore auto-generate an ID for you. You can do this by calling the following language-specific add() methods:

```
import { collection, addDoc } from "firebase/firestore";

// Add a new document with a generated id.
const docRef = await addDoc(collection(db, "cities"), {
  name: "Tokyo",
  country: "Japan"
});
console.log("Document written with ID: ", docRef.id);
```

For Blogs app the syntax will be:

```
const docRef = collection(db, "blogs");
await addDoc(docRef, {
  title: formData.title,
  content: formData.content,
  createdOn: new Date()
});
```

Set a document

To create or overwrite a single document, use the following language-specific set() methods:

```
import { doc, setDoc } from "firebase/firestore";

// Add a new document in collection "cities"
await setDoc(doc(db, "cities", "LA"), {
  name: "Los Angeles",
  state: "CA",
  country: "USA"
});
```

If the document does not exist, it will be created. If the document does exist, its contents will be overwritten with the newly provided data unless you specify that the data should be merged into the existing document, as follows:

```
import { doc, setDoc } from "firebase/firestore";

const cityRef = doc(db, 'cities', 'BJ');
setDoc(cityRef, { capital: true }, { merge: true });
```

setDoc is useful where you are generating IDs by yourself or adding a new one.

Fetching Data from the Database

Get Data

The following example shows how to retrieve the contents of a single document using get():

```
import { doc, getDoc } from "firebase/firestore";

const docRef = doc(db, "cities", "SF");
const docSnap = await getDoc(docRef);

if (docSnap.exists()) {
  console.log("Document data:", docSnap.data());
} else {
  // doc.data() will be undefined in this case
  console.log("No such document!");
}
```

You can also retrieve multiple documents with one request by querying documents in a collection. For example, you can use where() to query for all of the documents that meet a certain condition, then use get() to retrieve the results:

```
import { collection, query, where, getDocs } from "firebase/firestore";

const q = query(collection(db, "cities"), where("capital", "==", true));

const querySnapshot = await getDocs(q);
querySnapshot.forEach((doc) => {
  // doc.data() is never undefined for query doc snapshots
  console.log(doc.id, " => ", doc.data());
});
```

In addition, you can retrieve all documents in a collection by omitting the where() filter entirely:

```
import { collection, getDocs } from "firebase/firestore";

const querySnapshot = await getDocs(collection(db, "cities"));
querySnapshot.forEach((doc) => {
  // doc.data() is never undefined for query doc snapshots
  console.log(doc.id, " => ", doc.data());
});
```

Data Sync - Getting Realtime updates

You can listen to a document with the onSnapshot() method. An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.

```
import { doc, onSnapshot } from "firebase/firestore";

const unsub = onSnapshot(doc(db, "cities", "SF"), (doc) => {
    console.log("Current data: ", doc.data());
});
```

For Blogs App, we are using the following code:

```
useEffect(() => {
  async function fetchData(){
      const snapShot =await getDocs(collection(db, "blogs"));
      console.log(snapShot);

      const blogs = snapShot.docs.map((doc) => {
          return{
              id: doc.id,
              ...doc.data()
          }
      })
      console.log(blogs);
      setBlogs(blogs);
  }

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

Deleting the Documents from Database

To delete a document, use the following language-specific delete() methods:

```
import { doc, deleteDoc } from "firebase/firestore";
await deleteDoc(doc(db, "cities", "DC"));
```

For blogs app, we are using the following code:

```
async function removeBlog(id){
  const docRef = doc(db,"blogs",id);
  await deleteDoc(docRef);
}
```

Summarizing it

Let’s summarize what we have learned in this Lecture:

- Learned about SQL and No SQL Databases.

- Learned about Realtime Databases.
- Learned about Cloud Firestore and configuration.
- Learned how to add and get data from Cloud Firestore.
- Learned how to display real-time updates.
- Learned how to delete documents from the database.

Some References:

- Realtime DB vs Cloud Firestore: link
- Cloud Firestore: link
- Data Model: link
- Firestore Configuration: link
- Add Data: link
- Get Data: link
- Get Real Time Updates: link
- Delete Documents: link

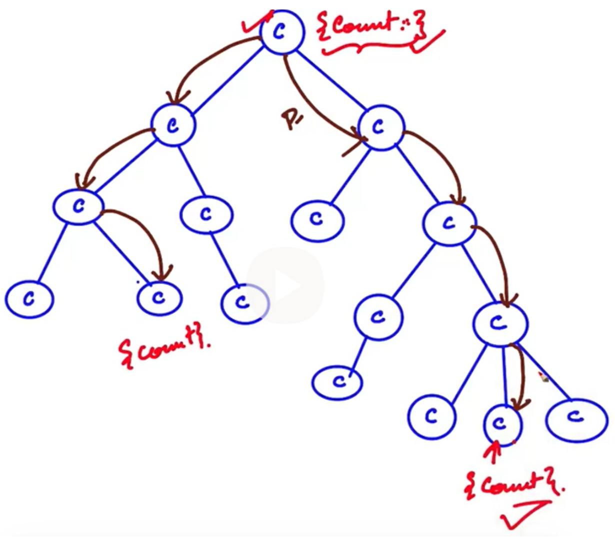
Context API

Passing Data From Parent to Child

Prop Drilling - Passing State from Parent to Child

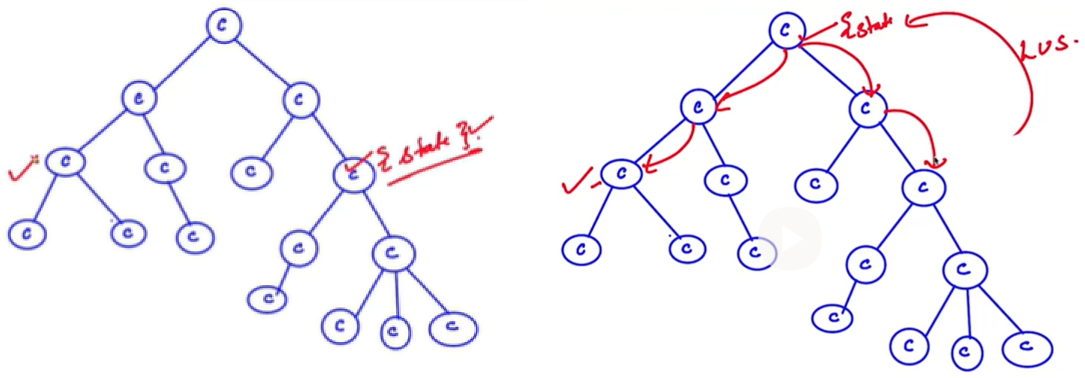
Your application might initially only have one component, but as it becomes more sophisticated, you must continue dividing it into smaller components. We can create a separation of concerns by isolating specific portions of a bigger application using components. Whenever anything in your program malfunctions, fault isolation makes it simple to pinpoint the problem area.

Props can be used to enable communication between our components in React. Prop drilling is a situation where data is passed from one component through multiple interdependent components until you get to the component where the data is needed. Prop drilling is not ideal as it quickly introduces complicated, hard-to-read code, re-rendering excessively, and slows down performance. Component re-rendering is especially damaging since passing data down multiple levels of components triggers the re-rendering of components unnecessarily.



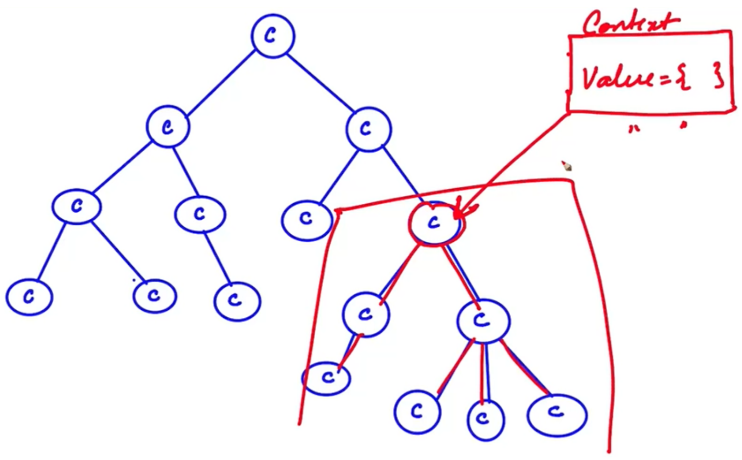
Lifting Up the State - Passing State to siblings

Lifting state up occurs when the state is placed in a common ancestor (or parent) of child components. Because each child component has access to the parent, they will then have access to the state (via prop drilling). If the state is updated inside the child component, it is lifted back to the parent container. However, as we are utilizing a poorly maintained pattern for the state, this approach can create issues in the future.



Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

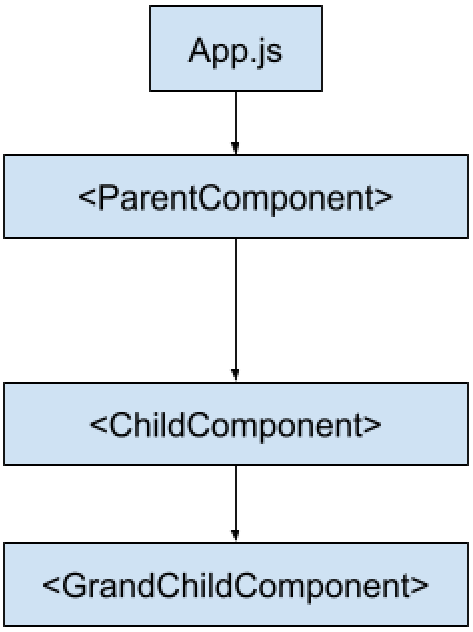


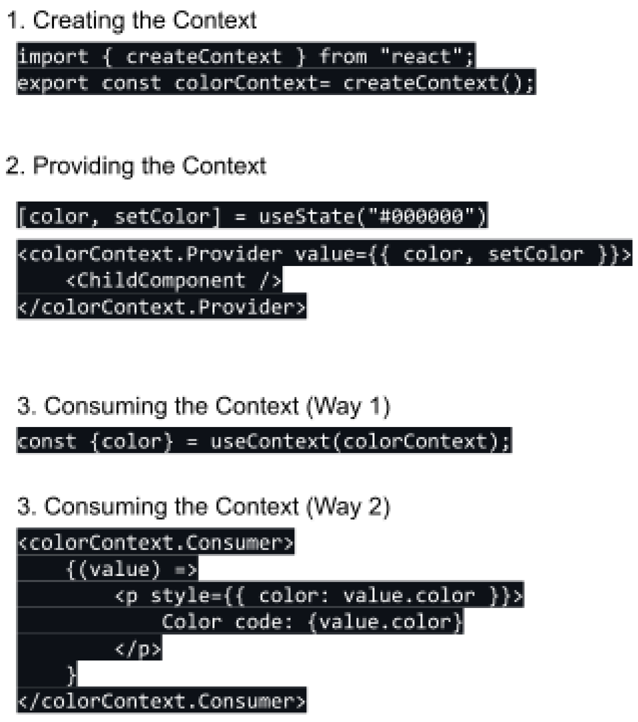
React Context API

The React Context API is a component structure that allows us to share data across all levels of the application. The main aim of Context API is to solve the problem of prop drilling (also called "Threading"). There are three steps to using React context:

1. **Create context -** using the `createContext` method.
2. **Provide Context -** Setup a `Context.provider` and define the data which you want to store.
3. **Consume the Context -** Use a `Context.consumer` or `useContext` hook whenever you need the data from the store.

Let's consider the following example:




1. Creating the Context

```
import { createContext } from "react";
export const colorContext= createContext();
```

2. Providing the Context

```
[color, setColor] = useState("#000000")
```

```
<colorContext.Provider value={{ color, setColor }}>
    <ChildComponent />
</colorContext.Provider>
```

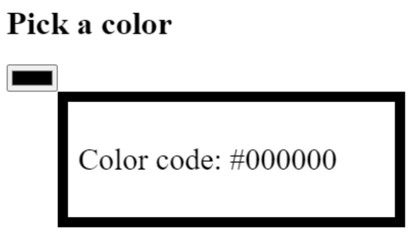
3. Consuming the Context (Way 1)

```
const {color} = useContext(colorContext);
```

3. Consuming the Context (Way 2)

```
<colorContext.Consumer>
    {(value) =>
        <p style={{ color: value.color }}>
            Color code: {value.color}
        </p>
    }
</colorContext.Consumer>
```

Output:


```
Pick a color

Color code: #000000
```

Creating the Context

React.createContext

`const MyContext = React.createContext(defaultValue);`

It is used for creating a Context object. When React renders a component subscribing to this Context object, it will read the current context value from the closest matching Provider above it in the tree.

```
import { createContext } from "react";
export const colorContext= createContext();
```

Providing the Context

Context.Provider

`<MyContext.Provider value={/* some value */}>`

Every Context object has a Provider React component which allows consuming components to subscribe to context changes. It acts as a delivery service. When a consumer component asks for something, it finds it in the context and provides it to where it is needed. The provider accepts a prop (value), and the data in this prop can be used in all the other child components. This value could be anything from the component state.

All consumers that are child components of a Provider will re-render whenever the Provider's value prop changes. Changes are determined by comparing the new and old values using the same algorithm as Object.is.

```
import { useState } from "react";
import ChildComponent from "./ChildComponent";
import { colorContext } from "../context";

const ParentComponent = (props) => {
  const [color, setColor] = useState("#000000");

  return (
    <>
      <h1>Pick a color</h1>
      <input type="color" onChange={(e) => { setColor(e.target.value);}}
value={color} />
      {/* Providing the context to the child component */}
      <colorContext.Provider value={{ color, setColor }}>
        <ChildComponent />
      </colorContext.Provider>
    </>
  );
};
```

Consuming the Context in Functional Components

useContext hook

`const value = useContext(SomeContext)`

useContext is a React Hook that lets you read and subscribe to context from your component.It can be used with the useState Hook to share the state between deeply nested components more easily.

```
import { useContext } from "react";
import { colorContext } from "../context";

const GrandChildComponent = () => {
  //Consuming the context
  const {value} = useContext(colorContext)
  return (
  <p style={{ color: value.color }}>Color code: {value.color}</p>
)
};
```

Consuming the Context in Class-Based Components

Context.Consumer

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
```

`</MyContext.Consumer>`

A React component that subscribes to context changes. Requires a function as a child. The function receives the current context value and returns a React node. The value argument passed to the function will equal the value prop of the closest Provider for this context in the component tree. If there is no Provider for this context, the value argument will be equal to the defaultValue, which was passed to createContext().

```
import React from 'react';
import { colorContext } from "../context";

class GrandChildComponent extends React.Component {
    render() {
        return (
            <colorContext.Consumer>
                {(value) => <p style={{ color: value.color }}>Color code:
{value.color}</p>}
            </colorContext.Consumer>
        );
    }
}
```

Using Multiple contexts

React also allows you to create multiple contexts. By providing multiple contexts in this way, components that require access to both context values can consume them both and be able to interact with their respective states. We should always try to separate contexts for different purposes to maintain the code structure and better readability.To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

For Example, the Items component may need access to the item state from itemContext and the total state from totalContext, allowing it to display the total number of items in the shopping cart along with the total cost.

```
import { itemContext } from "./itemContext";
import { totalContext } from "./totalContext";

```

```
function App() {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);
  return (
    // providing multiple contexts
    <itemContext.Provider value={{ item, setItem }}>
      <totalContext.Provider value={{ total, setTotal }}>
        <div className="App">
          <h2>Shopping Cart</h2>
          <Navbar />
          <Items />
        </div>
      </totalContext.Provider>
    </itemContext.Provider>
  );
}
export default App;
```

Custom Provider

It is a component which acts as a provider and it makes use of the default provider. Custom providers are created using the **createContext** function from the React library, which creates a new context object that can be passed down to child components using a provider component. The provider component is responsible for passing the context data down to its child components via a special prop called **value**.

By using a custom provider, you can centralize the management of shared data and state in a single place, making it easier to maintain and update your application. This can be particularly useful when working with complex applications that require a lot of shared state management, such as e-commerce sites or large data-driven applications.

For Example:

```
import { createContext, useState } from "react";
```

```
const itemContext = createContext();

function CustomItemContext({children}) {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);

  return(
    <itemContext.Provider value={{total,setTotal,item, setItem}}>
        {children}
    </itemContext.Provider>
  )
}

export { itemContext };
export default CustomItemContext;
```

Using Custom Providers:

```
import CustomItemContext, { itemContext } from "./itemContext";

function App() {
  return (
    // providing multiple contexts
    <CustomItemContext>
      <div className="App">
        <h2>Shopping Cart</h2>
        <Navbar />
        <Items />
      </div>
    </CustomItemContext>
  );
}
export default App;
```

Using Custom Hooks

Creation of a custom provider component that provides context data to its child components, as well as the creation of a custom hook that consumes the context data. Using custom hooks with custom providers allows for greater flexibility and reusability in sharing data across a React application. All the logic of the context file, updating logic and event handling, will be at one place

For Example, The **useValue** hook is defined to consume the context data provided by the **itemContext** using the **useContext** hook. The hook returns the total and item variables, functions handleAdd, and handleRemove from the context, making them available to any components that use the useValue hook.

```
import { createContext, useState, useContext } from "react";

const itemContext = createContext();

function useValue() {
  const value = useContext(itemContext);
  return value;
}

function CustomItemContext({ children }) {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);

  const handleAdd = (price) => {
    setTotal(total + price);
    setItem(item + 1);
  };

  const handleRemove = (price) => {
    if (total <= 0) {
      return;
    }
    setTotal((prevState) => prevState - price);
    setItem(item - 1);
  };

  return (
    <itemContext.Provider value={{ total, item, handleAdd, handleRemove }}>
      {children}
    </itemContext.Provider>
  );
}

export {  useValue };
export default CustomItemContext;
```

Summarising it

Let’s summarise what we have learned in this Lecture:

- Learned about Prop Drilling.
- Learned how to create Context.
- Learned how to provide Context.
- Learned how to consume the Context.
- Learned how to use multiple contexts.
- Learned about custom providers.
- Learned about custom hooks.

Some References:

- Context: link
- React Context for beginners: link

React Router

Routing Mechanism

Routing in React is used to manage the URLs of the application and map them to different views or components that need to be displayed on the page.

In MPAs, each page has its own URL, and when the user navigates to a new page, the browser sends a request to the server, and the server responds with a new HTML page, which replaces the current page in the browser. The server determines which page to return based on the URL requested by the client. This process is known as server-side routing. This approach can be slower and less responsive, and it can lead to longer load times.

In contrast, in SPAs, the application is loaded once, and all the content is loaded dynamically without the need for page refreshes. Instead of loading new pages, the application updates the current view by manipulating the DOM. SPAs use client-side routing, which means that the routing is handled by the client-side JavaScript code. This process is known as client-side routing.This allows for faster and more responsive navigation, as the entire page does not need to be reloaded.

React Router

ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. A Route is used to define and render components based on the specified path. When the user navigates to a particular URL, React Router renders the component associated with that route.

For Example,

```
src/
    components/
      Header.js
```

```
        Footer.js
        Button.js
        Input.js
        ...
    pages/
        Home.js
        About.js
        Contact.js
        ...
    routes.js
    App.js
    index.js
```

In this example, the components folder contains reusable UI components that can be used across different pages. The pages folder contains the different views or pages of the application.

The routes.js file contains the route definitions for the application. This is where the <Route> components are defined, along with the path and the component to be rendered for each route.

Types of React Router

In React Router v6.4, there are different types of routers that can be used depending on the needs of the application:

- **<BrowserRouter> -** This is the most commonly used router and is designed for applications that will be hosted on a server that will serve all requests to the application. It uses HTML5 history API to keep the UI in sync with the URL.
- **<HashRouter> -** This router is designed for applications that will be hosted on a server that does not support HTML5 history API, such as GitHub Pages or static file servers. It uses the URL hash to keep the UI in sync with the URL.
- **<MemoryRouter> -** This router is designed for testing and non-visual use cases, such as server-side rendering or testing.

- **<NativeRouter> -** This router is designed for React Native applications and uses the native routing features of the platform.
- **<StaticRouter> -** This router is designed for server-side rendering and generates a set of static routes based on a given location.

In v6.4, new routers were introduced that support the new data APIs and to create custom routers:

- **createBrowserRouter:** This function creates a custom <BrowserRouter> router with a custom history object. The custom history object can be used to manipulate the browser's URL and handle navigation between pages without causing a full page refresh.
- **createMemoryRouter:** This function creates a custom <MemoryRouter> router with a custom history object. The custom history object can be used to manipulate the router's state and handle navigation between pages.
- **createHashRouter:** This function creates a custom <HashRouter> router with a custom history object. The custom history object can be used to manipulate the browser's URL hash and handle navigation between pages without causing a full page refresh.

createBrowserRouter

This is the recommended router for all React Router web projects. It uses the DOM History API to update the URL and manage the history stack.

For example, WAY-1

```
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";

function App() {
  const router = createBrowserRouter([
    { path: "/", element: <Home /> },
    { path: "/about", element: <About /> },
  ]);
  return (
    <>
      <RouterProvider router={router} />
    </>
  );
}

export default App;
```

In this example, the createBrowserRouter function is used to create a custom <BrowserRouter> router with two routes: one for the home page, and one for the about page. The RouterProvider component is used to wrap the app and provide access to the custom router.

1. Import the necessary modules, including createBrowserRouter, RouterProvider, Home, and About.
2. Use createBrowserRouter to create a custom <BrowserRouter> router with the two routes: Home and About.
3. Wrap the app with RouterProvider and pass in the custom router as a prop. The RouterProvider component ensures that the routing context is available to all child components of your app.
4. Render the App component.

For Example, WAY-2

```
import {
    createBrowserRouter,
    createRoutesFromElements,
    Route,
    RouterProvider,
  } from "react-router-dom";
  import Home from "./pages/Home";
  import About from "./pages/About";
  import Items from "./pages/Items";

  function App() {
    const routes = createRoutesFromElements(
      <>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/items" element={<Items />} />
      </>
    );
    const router = createBrowserRouter(routes);

    return (
      <>
        <RouterProvider router={router} />
      </>
    );
  }

  export default App;
```

In this example, the **createRoutesFromElements** function is used to create an array of route objects from JSX elements and avoid manually creating an array of route objects. The **createBrowserRouter** function is then used to create a custom **<BrowserRouter>** router with the array of route objects. Finally, the **RouterProvider** component is used to wrap the app and provide access to the custom router.

<Link> Component

The <Link> component is a core component of React Router that is used to navigate between different routes in your application. It is a declarative way to create hyperlinks that allows you to handle navigation without reloading the entire page. The to prop specifies the target route. When a user clicks on a <Link> component, React Router will automatically update the URL and render the appropriate component for the target route.

For Example,

```
import { Link } from "react-router-dom";

function Home() {
  return (
    <>
      <main>
        <h1>Home Page</h1>
        <Link to="/about">About</Link>&nbsp;&nbsp;
        <Link to="/items">Items</Link>
      </main>
    </>
  );
}

export default Home;
```

The Link component can be used to create the back link, and the to prop is set to "/" to specify the target route. When the link is clicked, the user will be taken back to the home page.

```
import { Link } from "react-router-dom";
function About() {
  return (
    <>
      <main>
        <h1>About Page</h1>
        <Link to="/">back</Link>
      </main>
    </>
  );
}

export default About;
```

Nested Routes

Nested routes in React Router allow you to define routes that are nested within another route. This can be useful for building more complex UIs, such as layouts with multiple sections that have their own routes.
For Example,

```
import {
    createBrowserRouter,
    createRoutesFromElements,
    Route,
    RouterProvider,
  } from "react-router-dom";
```

```
import Home from "./pages/Home";
import About from "./pages/About";
import Items from "./pages/Items";
import Navbar from "./components/Navbar";

function App() {

  const router = createBrowserRouter([
    {
      path: "/",
      element: <Navbar />,
      children: [
        { index: true, element: <Home /> },
        { path: "/about", element: <About /> },
        { path: "/items", element: <Items /> },
      ],
    },
  ]);
  return (
    <>
      <RouterProvider router={router} />
    </>
  );
}

export default App;
```

Parent route is defined with the path of / and an element of Navbar. The children property is an array of child routes, which includes an index route, a route with the path of /about, and a route with the path of /items.

```
import { Link, Outlet } from "react-router-dom";

function Navbar() {
  return (
    <>
      <div className="nav">
        <Link to="/">
          <h4>HOME</h4>
        </Link>
        <Link to="/about">
          <h4>ABOUT</h4>
        </Link>

        <Link to="/items">
          <h4>ITEMS</h4>
        </Link>
      </div>
      <Outlet />
    </>
  );
}

export default Navbar;
```

The Navbar component renders links to the child routes using the Link component. The Outlet component renders the appropriate child component based on the current nested route. The Outlet component renders the appropriate child component based on the current nested route. For example, if the current URL is /about, the Outlet component will render the About component.

<NavLink> Component

The <NavLink> component is similar to the <Link> component in that it creates a hyperlink to a specified location. However, it adds the ability to add styling and active classes to the link when it is active. This can be useful for highlighting the current page or route in a navigation menu.

For Example,

```
import { NavLink, Outlet } from "react-router-dom";

function Navbar() {
  return (
    <>
      <div className="nav">
        <NavLink
          style={({ isActive }) => (isActive ? { color: "blue" } : undefined)}
          to="/"
        >
          <h4>HOME</h4>
        </NavLink>

        <NavLink
          style={({ isActive }) => (isActive ? { color: "blue" } : undefined)}
          to="/about"
        >
          <h4>ABOUT</h4>
        </NavLink>

        <NavLink
          style={({ isActive }) => (isActive ? { color: "blue" } : undefined)}
          to="/items"
        >
          <h4>ITEMS</h4>
        </NavLink>
      </div>
      <Outlet />
    </>
  );
}

export default Navbar;
```

In this example, the exact attribute is used with the to prop of the first NavLink. This ensures that the link is only active when the exact URL path matches the value of to.

Relative vs Absolute Paths

In web development, a path is a URL endpoint that specifies the location of a specific resource or content on a web server. A base URL is the root URL that serves as the starting point for all the other URLs in a website. In React Router, paths are used to define routes that map to specific components in your application. These routes can be either relative or absolute.

A relative path is a path that is relative to the current location. For example, if the current location is "/users", and you want to link to the "profile" page, the relative path would be "profile". The resulting link would be "/users/profile".

An absolute path is a path that starts with a forward slash ("/") and is relative to the root of the website. For example, if the current location is "/users", and you want to link to the "home" page, the absolute path would be "/home". The resulting link would be "/home".

In general, it is recommended to use relative paths whenever possible, as they are more flexible and can be used in different contexts. Absolute paths are useful when you need to link to a specific page that is not in the current directory or when you want to link to a page in a different part of the website.

Dynamic Routes

Dynamic routes are routes that contain parameters, which can be used to dynamically generate content based on user input or other data. In React Router, dynamic routes are defined using a colon (:) followed by the name of the parameter. For example, a dynamic route for a user profile page might look like /users/:id, where :id is the parameter that will be replaced with the actual user ID.

To access the parameter in your component, you can use the useParams hook provided by React Router. For example:

```
import { useParams } from "react-router-dom";

function UserProfile() {
  const { id } = useParams();
  return <div>User profile for user {id}</div>;
}
```

Summarizing it

Let’s summarize what we have learned in this Lecture:

- Learned about Routing Mechanism in React.

- Learned about React Router.
- Learned about types of React Router.
- Learned about createBrowserRouter.
- Learned about Nested Routes.
- Learned about <Link> and <NavLink> Components.
- Learned about Relative, Absolute and Dynamic Routes.

Some References:

- React Router Documentation: link
- Client Side Routing: link

Pure Functions

Pure Functions in Javascript

A pure function is a function that always returns the same output given the same input and has no side effects (i.e., it doesn't modify any variables outside of its scope, it doesn't mutate its input arguments, and it doesn't have any I/O operations such as reading from or writing to a file or a database). Pure functions are predictable and easier to reason about since they don't have any hidden dependencies or side effects. Pure functions can be composed together to create more complex functions or pipelines of functions since their input and output types are well-defined and consistent.

```
function add(a, b){ // A pure function adding two integers passed in it.
    return a+b;
}

function divide(a, b){  // Pure function to divide two integers passed
in it.
    return a/b;
}

function multi(a, b){    // Pure function to multiply two integers
passed in it.
    return a*b;
}
console.log(        // Calling all the pure functions
  add(2,5),
  multi(3,2),
  divide(20,5)
);
```

All three functions in the above code are pure functions. Their return value depends on the input arguments, they don't mutate any non-local state, and they have no side effects (we will discuss side effects further in this article). Examples of pure functions in JavaScript include Math.abs(), parseInt(), JSON.stringify(), and many others.

Pure functions can be used with functional programming techniques such as higher-order functions, currying, and partial application. JavaScript libraries and frameworks such as Redux, Ramda, and Lodash emphasize using pure functions and functional programming principles.

Impure Functions

An impure function is a function that either modifies variables outside of its scope, mutates its input arguments, has I/O operations such as reading from or writing to a file or a database, or has other side effects that are not purely computational. Impure functions can have hidden dependencies and side effects, which can make them harder to reason about and debug.

```
const message = 'Hi ! ';
function myMessage(value) {
    return `${message} ${value}`
}
console.log(myMessage('Aayushi'));
```

In the above code, the result the function returns is dependent on the variable that is not declared inside the function. That's why this is an impure function.
Examples of impure functions in JavaScript include console.log(), Math.random(), and Date.now(), Array.sort(), Array.splice(), and many others.

Impure functions can be necessary for tasks such as reading and writing to a file or a database, generating random numbers, or interacting with the user interface. However, it's important to minimize impure functions and keep them separate from pure functions as much as possible, to maintain a clear separation of concerns and avoid unexpected interactions or bugs. Pure functions can be composed together to create complex logic and pipelines of functions, whereas impure functions can only be used in a more limited and isolated way.

JavaScript libraries and frameworks such as React and Angular provide mechanisms for managing the state of an application and minimizing the use of impure functions in the user interface.

Redux

Issues with Prop Drilling

Prop drilling is a common problem in React applications where data is passed down through multiple layers of components via props. Prop drilling can lead to several issues:

1. **Storage Issue:** When large amounts of data is passed down through many layers of components via props, It can lead to issues with data storage and retrieval, as well as code maintainability and performance.
2. **Predictability of data**: Prop drilling can also make it difficult to predict where data comes from and how it will be used. It can be difficult to keep track of where the data is being used and where it is being modified. This can lead to issues with data consistency and can make it difficult to debug issues.
3. **Flow of Data:** Prop drilling can also make it difficult to pass data back up the component hierarchy.
4. **Data from multiple sources:** In complex applications, data may come from multiple sources, such as APIs or external services. Prop drilling can make it difficult to manage data and adds complexity to the application.

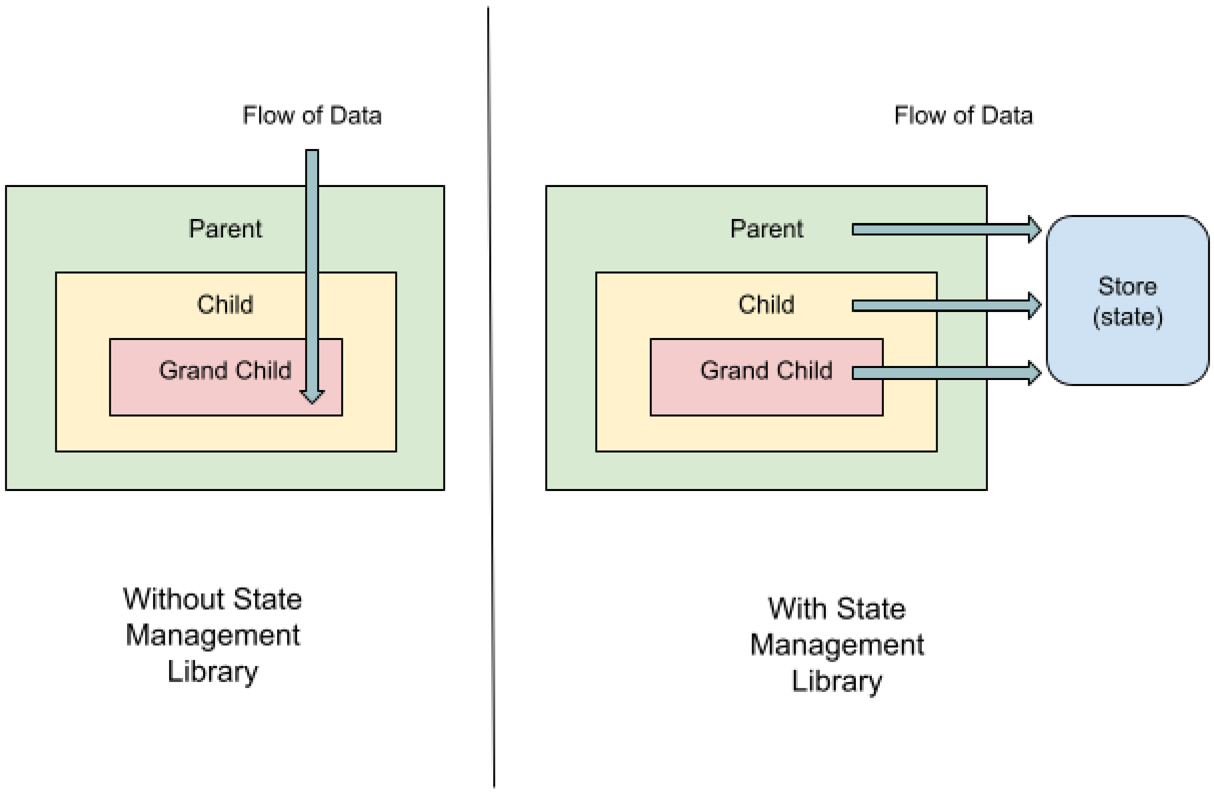
State Management

State management is a way to facilitate communication and sharing of data across components. State management libraries are tools used to manage and organize the state of an application predictably and efficiently. These libraries provide a set of rules and techniques for storing, retrieving, and updating application states. Advantages of using state management libraries are:

- **Centralized state:** State management libraries typically use a centralized store to manage the application state. This store is often implemented as a

JavaScript object that can be accessed and modified by components throughout the application.

- **Unidirectional data flow:** Data flows in a single direction in state management libraries, from the store to components. Components can update the store, but they cannot update other components directly.
- **Predictable state updates:** State management libraries provide a set of rules for updating the state, which helps to ensure that state changes are predictable and consistent across the application.
- **Immutable state:** Many state management libraries encourage the use of immutable data structures, which can help to prevent unintended side effects and make state updates more predictable.



Context API

Context API is a feature in React that provides a way to pass data through the component tree without having to pass props down manually at every level. It allows you to create a global state that can be accessed and modified by any component in the tree without the need for prop drilling. Context API can be useful for managing states in cases where a small amount of data needs to be shared across multiple components, but is not ideal for larger and more complex state management needs.

Limitations

- **Overuse of context:** Overusing context can lead to a complex and difficult-to-manage application. Context should be used sparingly and only for data that needs to be shared across multiple components.
- **Designed for static content:** Context is designed for passing static data through the component tree, so it may not be the best choice for managing a dynamic state that changes frequently.
- **Re-renders the Context Consumers:** Whenever the value of the context changes, all the components that consume that context will re-render. This can lead to performance issues if the context value changes frequently.
- **Performance:** Context can cause performance issues if the context value is deeply nested and needs to be updated frequently.
- **Difficult to debug:** When an issue arises, debugging can be difficult since the data flow is not always clear. It can be difficult to trace where data is being passed and where it is being modified.
- **Difficult to extend and scale:** As an application grows in size and complexity, context can become difficult to manage and maintain.

Currying

Currying is defined as changing a function having multiple arguments into a sequence of functions with a single argument.
When currying a function in JavaScript, closures are used to retain the values of previous arguments that have been passed to the curried function. This is because each time a new argument is passed, a new function is returned that has access to the previous arguments.

For Example:

```
function sum(x){
    return function(y){
        return function(z){
            return x+y+z;
        }
    }
}

const sumXResult = sum(2);
const sumYResult = sumXResult(4);
const sumZResult = sumYResult(6);
console.log(sumZResult);
```

sum is a curried function that takes one argument x and returns another function that takes one argument y, which returns a third function that takes one argument z. The final function returns the sum of **x, y, and z**.

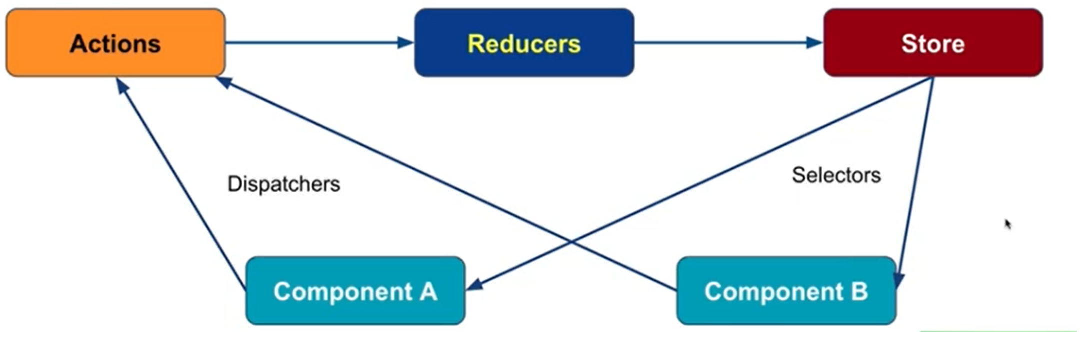
- When **sum(2)** is called, it returns a new function that takes one argument y. This returned function is assigned to **sumXResult**.
- When **sumXResult(4)** is called, it returns a new function that takes one argument z. This returned function is assigned to **sumYResult**.
- When **sumYResult(6)** is called, the final function is invoked, and it returns the sum of 2, 4, and 6, which is 12.

Finally, the result of **sumZResult** is printed to the console, which outputs 12.

Benefits of Currying

- Reusability: Currying allows you to create reusable functions by breaking down a function that takes multiple arguments into smaller functions that can be reused.
- Readability: Currying can improve the readability of the code by reducing the number of arguments passed to a function.
- Function Composition: Currying is useful for function composition, where the output of one function is the input to another function. It makes it easier to chain multiple functions together and create new functions that perform complex operations.

Redux Architecture



In Redux, the state of an application is represented as a single JavaScript object called the "store". The store is responsible for managing the application's state and updating the view when the state changes. The data flow in Redux is unidirectional, meaning that the data flows in one direction, from the view to the store and back to the view. This makes it easier to reason about the application's state and simplifies debugging.

Redux uses a "one-way data flow" app structure.

- The state describes the condition of the app at a point in time, and UI renders based on that state
- When something happens in the app:
  - The UI dispatches an action
  - The store runs the reducers, and the state is updated based on what occurred
  - The store notifies the UI that the state has changed
- The UI re-renders based on the new state

Installation of Redux

Run the following command to install Redux as a dependency for your project:

```
npm install redux
```

Components of the Redux Architecture

Store

The store is the central place where the application's state is stored. It is created using a reducer function that determines how the state should be updated based on the actions dispatched to the store.

For Example:

```
// store
const redux = require('redux');
const store = redux.createStore(todoApp);
```

Actions

Actions are plain JavaScript objects that describe what should happen in the application. They are created by calling action creator functions and are dispatched to the store using the store.dispatch() method. Payload is the data that needs to be transferred.

For Example:

```
// action types
const ADD_TODO = 'ADD_TODO';
const TOGGLE_TODO = 'TOGGLE_TODO';

// action creators
const addTodo = (text) => ({ type: ADD_TODO, text });
const toggleTodo = (index) => ({ type: TOGGLE_TODO, index });
```

Reducers

Reducers are pure functions that take the current state and an action as arguments and return a new state. They are responsible for updating the application's state based on the actions dispatched to the store.

For Example:

```
// reducer
```

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return {
        ...state,
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false,
          },
        ],
      };
    case TOGGLE_TODO:
      return {
        ...state,
        todos: state.todos.map((todo, i) => {
          if (i === action.index) {
            return {
              ...todo,
              completed: !todo.completed,
            };
          }
          return todo;
        }),
      };
    default:
      return state;
  }
}
```

View

The view is responsible for displaying the current state of the application to the user. It subscribes to changes in the store and updates the view when the state changes.

Dispatchers

a dispatcher is a function that receives an action object and dispatches it to the Redux store, triggering a state change. The dispatcher then passes the action object to the store's reducer function, which applies the update to the application state according to the rules defined by the application's business logic.

For Example:

```
// dispatch actions
store.dispatch(addTodo('Learn Redux'));
store.dispatch(addTodo('Build an app'));
store.dispatch(toggleTodo(0));

// get the current state
console.log(store.getState());
```

Selectors

Selectors are functions that extract specific data from the application's state. They provide a way to access and transform the data stored in the store.

For Example:

```
// selector
const getCompletedTodos = (state) => {
    return state.todos.filter((todo) => todo.completed === true);
    };
```

Principles of Redux

- Global app state is kept in a single store
- The store state is read-only to the rest of the app
- Reducer functions are used to update the state in response to actions

Summarizing it

Let's summarize what we have learned in this Lecture:

- Learned about Issues with Prop Drilling.
- Learned about State Management.

- Learned about Context API.
- Learned about Redux and Architecture.
- Learned about components of Redux.
- Learned about the Principles of Redux.

Some References:

- Context: link
- Redux Documentation: link

Redux in React

React Redux Library

React Redux is a popular library that provides a predictable state container for JavaScript applications using the React library. The react-redux library provides a centralized store for state management in React applications. It offers several hooks that help in optimizing code and improving application performance. Hooks are an important concept in React Redux because they allow developers to extract logic from components and reuse it across the application.

Installation

To use React Redux with your React app, install it as a dependency:

```
// If you use npm:
npm install react-redux

// Or if you use Yarn:
yarn add react-redux
```

Provider Component

The Provider component is a React component that allows you to provide the Redux store to all components in your application. It is the top-level component that wraps your entire application and makes the store available to all child components.
The Provider component takes a store prop, which is the Redux store that you want to provide to your application.

Here's an example of how to use the Provider component in a ToDo application:

```
import { useState } from "react";
import { Provider } from "react-redux";
import TodoForm from "./components/ToDoForm/ToDoForm";
import TodoList from "./components/ToDoList/ToDoList";
import store from "./redux/store";

import './App.css';

function App() {
  const [todos, setTodos] = useState([]);

  const createTodo = (text) => {
    setTodos([...todos, { id: todos.length + 1, text, completed: false}]);
  };

  const toggleTodo = (index)=>{
    const list = [...todos];
    list[index].completed = !list[index].completed;
    setTodos(list);
  }

  return (
    <div>
      <h1>To Do App</h1>
      <Provider store={store}>
        <TodoForm onCreateTodo={createTodo} />
        <TodoList todos={todos} onToggle={toggleTodo} />
      </Provider>
    </div>
  );
}

export default App;
```

By default, when the Provider element is used, all child components will have access to the entire Redux store. However, it is possible to scope store access to specific components using the store prop of the Provider element. To do this, you can create a separate Redux store for each component that requires scoped access to the store. Then, when rendering the component, pass in the appropriate store as a prop to the Provider element.

Hooks

React Redux provides a pair of custom React hooks that allow your React components to interact with the Redux store.

useSelector

The **useSelector** hook is used to extract data from the Redux store. It takes a selector function as input and returns the selected data from the store. So, if store gets updated it will not directly impact the components. This also helps in abstraction and encapsulation of store by hiding the important object. For example, Here the useSelector hook is used to retrieve the todos state from the store. The todos state is then mapped over and rendered to the screen as a list of todo items.

```
import { useSelector } from "react-redux";
import "./ToDoList.css";

function ToDoList() {

  const todos=useSelector((state)=> state.todos)

  return (
    <div className="container">
    <ul>
      {todos.map((todo,index) => (
        <li key={todo.id}>
          <span className="content">{todo.text}</span>
          <span className={todo.completed ?
'completed':'pending'}>{todo.completed ? 'Completed': 'Pending'}</span>
          <button className="btn btn-warning">Toggle</button>
          </li>
      ))}
    </ul>
    </div>
  );
}

export default ToDoList;
```

The useSelector hook is useful for optimizing performance by avoiding unnecessary re-renders. It allows you to select only the data you need from the store, which can help reduce the amount of data that needs to be processed by the component.

useDispatch

The **useDispatch** hook is used to dispatch actions to modify the state. It returns a reference to the dispatch function provided by the store. This hook can be used to dispatch actions from any component in the application, without the need for prop drilling. The useDispatch hook can be used to dispatch actions from any component in the application.

For example, Each todo item includes a button that, when clicked, dispatches a toggleTodo action to the store. The toggleTodo action is imported from the todoActions file, which contains action creators for various todo-related actions.

The dispatch function is used to dispatch the toggleTodo action, passing in the index of the current todo item as a parameter. This will update the completed property of the selected todo item in the store, which will trigger a re-render of the TodoList component.

```
import { useSelector, useDispatch } from "react-redux";
import { toggleTodo } from "../../redux/actions/todoActions";

import "./ToDoList.css";

function ToDoList() {

  const todos=useSelector((state)=> state.todos);
  const disptach = useDispatch();

  return (
    <div className="container">
    <ul>
      {todos.map((todo,index) => (
        <li key={todo.id}>
          <span className="content">{todo.text}</span>
          <span className={todo.completed ?
'completed':'pending'}>{todo.completed ? 'Completed': 'Pending'}</span>
          <button className="btn btn-warning"
          onClick={()=>{dispatch(toggleTodo(index))}}
          >Toggle</button>
          </li>
      ))}
    </ul>
    </div>
  );
}

export default ToDoList;
```

The useDispatch hook is useful for optimizing code efficiency by providing a simplified way of dispatching actions. It allows you to avoid the need to pass dispatch down as a prop to child components.

Multiple Reducers

In a typical React Redux application, the state is managed by reducers, which are functions responsible for handling different parts of the state. The decision to use multiple reducers or a single reducer depends on the complexity of your application's state. If your application's state is simple and straightforward, a single reducer may suffice. However, as your application grows in complexity, it can become difficult to manage a large state with a single reducer. Using multiple reducers in your React Redux application can provide better organization, improved scalability, and better performance. For example, let's say you have an e-commerce application that manages user accounts, products, and orders. You can create three separate reducers for each of these parts of the state.

Combining Reducers

Combining reducers is a technique used in React Redux to manage a complex state in a more organized and manageable way. It involves creating multiple reducers that handle different parts of the state and then combining them into a single root reducer using the **combineReducers** function. The combineReducers function takes an object as its argument, where the keys represent the keys of the root state object, and the values represent the individual reducers.

For example in this case, the root state object has two keys, todos and notes, each of which maps to the corresponding reducer.

```
import * as redux from "redux";
import { combineReducers } from "redux";
import { noteReducer } from "./reducers/noteReducer";
import {todoReducer} from "./reducers/todoReducer";

const result = combineReducers({
    todos:todoReducer,
    notes:noteReducer
})

export const store = redux.createStore(result);
```

With this setup, you can now dispatch actions to update the state managed by each reducer separately. For example, to add a todo item, you can dispatch the following action:

```
{
  type: 'ADD_TODO',
  id: 1,
  text: 'Buy milk'
}
```

This action will be handled by the todoReducer, which will update the todos state accordingly. Similarly, to add a note, you can dispatch the following action:

```
{
  type: 'ADD_NOTE',
  id: 1,
  text: 'Call John'
}
```

This action will be handled by the noteReducer, which will update the notes state accordingly.

Summarizing it

Let’s summarize what we have learned in this Lecture:

- Learned about React Redux library.
- Learned about Provider Component.
- Learned about useSelector Hook.
- Learned about useDispatch Hook.
- Learned about Multiple Reducers.
- Learned about Combining the Reducers

Some References:

- Redux API Reference: link
- React Redux Quick Start: link

Redux Toolkit

Introduction

Redux is a state management library that helps manage application states in a predictable and consistent manner. It is widely used in modern web development with React and has been an essential tool for building scalable and maintainable applications. However, working with Redux can sometimes be challenging, especially when it comes to setting up and managing reducers, actions, and selectors. To address these challenges, the Redux team has released Redux Toolkit, a package that simplifies the process of setting up and using Redux.

Challenges with Redux

- **Boilerplate Code:** Setting up Redux involves a lot of boilerplate code. You must create separate files for actions, reducers, and store configuration. This can be time-consuming and error-prone.
- **Complex Reducers:** Writing complex reducers can be difficult, especially when dealing with nested data structures or asynchronous actions.
- **Debugging:** Debugging Redux applications can be challenging, especially when dealing with large and complex application states.

Redux Toolkit library

Redux Toolkit provides a streamlined way of working with Redux, eliminating many of the common pain points of building large-scale Redux applications. It is designed to be backward-compatible with existing Redux code, making it easy to adopt and integrate into existing projects. Some of its key features include:

- A "slice" API that simplifies the process of creating Redux reducers
- A "createAsyncThunk" API that simplifies the process of handling asynchronous actions
- A simplified and standardized file structure for Redux code

- Automatic generation of Redux actions and reducers for common use cases
- A collection of other useful utilities and middleware for Redux.

Migrating from Redux to Redux Toolkit

Install Redux Toolkit

Start by installing Redux Toolkit in your application by running the command:

```
npm install @reduxjs/toolkit
yarn add @reduxjs/toolkit
```

Create slices

Replace your existing Redux actions and reducers with "slices," which are predefined Redux logic blocks in Redux Toolkit. You can create slices using the createSlice() function provided by Redux Toolkit.
To define a slice using createSlice, you need to call the function and pass in an object that contains the name, initialState, and reducers properties.

- name: This property is used to define the name of the slice. It is used to generate the action types for the slice automatically and to create action creators with the correct names.
- initialState: This property is used to define the initial state of the slice. It is used to set the initial state of the store when it is first created.
- reducers: This property defines a set of reducer functions that can update the slice's state. It takes an object as an argument where each key-value pair represents a case reducer. The key is the name of the action, and the value is a function that updates the state. When an action is dispatched, the corresponding reducer function will be called, and the slice's state will be updated accordingly. The payload property of the action object can be accessed using action.payload inside the reducer functions.

```
const todoSlice = createSlice({
  name:'todo',
  initialState:initialState,
  reducers:{
      // this is add action
      add:(state, action)=>{
              state.todos.push({
                  text:action.payload,
                  completed: false
              })
      },
      toggle:(state, action)=>{
          state.todos.map((todo, i)=>{
              if(i==action.payload){
                  todo.completed=!todo.completed;
              }
              return todo;
          })
      }
  }
});
```

Migrating Store

Replace your existing store creation code with the configureStore() function provided by Redux Toolkit. This function simplifies the store creation process by automatically adding common middleware and other settings.

```
export const store = configureStore({
  reducer:{
      todoReducer,
      noteReducer
  }
})
```

Dispatching Actions

The actions object is exported separately from the reducer.

```
export const todoReducer=todoSlice.reducer;
export const actions = todoSlice.actions;
```

You can use this object to access your actions by importing them and passing them as arguments to the dispatch function.

```
<button className="btn btn-warning"
onClick={()=>{disptach(actions.toggle(index))}}>Toggle</button>
```

Setting Up Selectors

You can define a new selector function called todoSelector, which extracts the todoReducer slice of the state.

```
// selector
export const todoSelector = (state)=>state.todoReducer.todos;
```

Once you have defined your selectors, you can use the useSelector hook from the react-redux library to access the selected data in your components.

Create React App with Redux Template

Redux Toolkit provides a template for creating a React app with Redux preconfigured, which you can use to start quickly with building a new Redux-powered React application. Run the following command to create a new React app with the Redux Toolkit template:

```
npx create-react-app my-app --template redux
```

This will create a new React app with the Redux Toolkit template and install all the necessary dependencies.

Extra Reducers

Extra Reducer allows you to execute an action which is the action of some other reducer. It allows you to share an action, invoke an action or dispatch an action that belongs to some other reducer. Using extra reducers like this can help simplify your code and make it easier to share actions between different reducers in your Redux store.
For Example, if we want to dispatch a notification when a todo is added. Whenever an action with the type todo/add is dispatched, the function defined in the extraReducers property is executed.

```
const notificationSlice = createSlice({
  name:'notification',
  initialState,
  reducers:{
      reset: (state, action)=>{
          state.message="";
      }
  },
  extraReducers:{
      "todo/add":(state, action)=>{
          console.log("todo/add in notificationReducer");
        state.message="Todo is created";
      }
  }
});
```

The reset action allows you to clear the notification message when it is no longer needed, such as after a user has read the message. "reset" action that can be dispatched to reset the message property to an empty string.

Creating Extra Reducers using Builder and addCase

Creating extra reducers using the Builder and Case API in Redux Toolkit allows you to handle actions dispatched from other slices of your Redux store without hardcoding their names into your reducer.

```
const notificationSlice = createSlice({
  name:'notification',
  initialState,
  reducers:{
      reset: (state, action)=>{
          state.message="";
      }
  },
  extraReducers:(builder)=>{
      builder.addCase(actions.add, (state, action)=>{
          state.message="Todo is created";
      })
  }
});
```

The builder argument is an instance of the ActionReducerMapBuilder class, which provides methods for adding new case reducers to your slice. The addCase method takes two arguments: the action creator function and a callback function that handles the action. In this case, we pass the actions.add action creator function, which is defined elsewhere in our application, and a callback function that sets the message property of our state to "Todo is created". This approach allows us to handle actions from other parts of our application without tightly coupling our reducers to the actions that they handle.

Creating Extra Reducers using Maps

The extraReducers field uses a map object to define an action handler for the add action. The key of the map object is the action type, and the value is the function that will handle the action. In this case, when the add action is dispatched, the message is set to "Todo is created".

```
const notificationSlice = createSlice({
  name:'notification',
  initialState,
  reducers:{
      reset: (state, action)=>{
          state.message="";
      }
  },
  extraReducers:{
      // map objects: [key]: value
      [actions.add]: (state, action)=>{
          state.message="Todo is created";
      }
  }
});
```

Summarizing it

Let's summarize what we have learned in this Lecture:

- Learned about challenges with Redux.
- Learned what Redux Toolkit is.

- Learned how to migrate from redux to the redux toolkit.
- Learned how to create react app with the redux template.
- Learned about Extra Reducers.

Some References:

- Getting started with Redux Toolkit: link
- Installation: link