React is a library

Frameworks have strict rules
Libraries don't.

Topics – (Just React)

Core of react (state or UI manipulation, JSX)
Component reusability
Props
Propagate change (hooks)

Add-on topics

Router
State Management (**Context API**, Redux, **Redux Toolkit**, Zustand)
Class based component – Legacy
BAAS apps (Social Media, E-commerce, etc.)

After React

Frameworks – **Next.js**, Gatsby, Remix, etc.

Get started

1. Create project with “npm create-react-app”, “npm create-next-app”, “npm create vite@latest”.
2. “npm i” or “npm install” updates or installs the packages mentioned in “package.json”.
3. “package.json” contains scripts, dependencies, name of project, etc.

Project File Structure

- node_modules
- public
  - index.html
- src
  - index.js / main.jsx
  - App.jsx
  - Components
    - Component_name.jsx

Working

- index.js / main.jsx is injected in index.html.
- App.jsx is the main component with returns a JSX element or fragment.
- Each **component** is just a method (<App /> or App()) returning JSX.
- It is imported and rendered in index.js / main.jsx by creating root with ReactDOM.
  - ReactDom.createRoot(document.querySelector(“#root”).render(<App />);
- ReactDOM is Virtual DOM.

JSX element is converted into object

React.createElement('p',{attr: 'value'},"inner html",variables)

ReactDom.createRoot(document.querySelector(“#root”).render(reactElement);

render(elem) - elem should be jsx or obj created with react.createElement(..)

render(App()) - App() returns jsx

render(<App />) - jsx functions can be used as custom tags in jsx

State – useState() hook

Instead of variables we use state for storing dynamic data.
useState() hook is used to get a state for a data.

const [data, setData] = useState(“Raunak”); *here, data = “Hello”*
setData(“R”); *here, data = “R”*

Here, ‘data’ is state variable.
‘setData’ is a function to update state, because ‘data’ can not be directly update, it does not reflects in the UI.

Patching:

```
setData( data + 1 )
setData( data + 1 )
setData( data + 1 )
```

at the end data will be “R1” and not “R111” as it was expected

When updater function (setData) Is called multiple times one after another, it is executed only once.
To prevent it, we use:

```
setData((prevData)=> prevData + 1)
setData((prevData)=> prevData + 1)
setData((prevData)=> prevData + 1)
```

at the end data will be “R111”

React Fiber

Reconciliation: the diffing algorithm which compares the real DOM and virtual DOM to determine which parts need to be changed.

After reconciliation, rendering of only required parts of DOM happens.

Fiber is advanced algorithm which renders in batches.

- Some substantial components are not diffed, they are directly replaced.
- For diffing lists, keys are used which are stable, predictable, and unique.

Fiber enables to:

- Pause work and come back to it later.
- Assign priority to different types of work.
- Reuse previously completed work.
- Abort work if no longer needed.

Components

Functional component is just a simple JavaScript function; it accepts the data in the form of props and returns the react element.

It helps to work on each individual parts of page like footer, header, cards, etc.

If we need to show multiple cards, we will just import card component and use it as <Card /> multiple times.

If each card shows dynamic and different value, we can pass value as **props** to the card.

Even functions can be passed as props and it can be executed by the components.

```
<Card cardNumber = {5} cardName = ‘Raunak’ handleClick={ ()=>{ console.log(“Clicked 1!” } } />

<Card cardNumber = {15} cardName = ‘Ronnie’ handleClick={ ()=>{ console.log(“Clicked 2!” } } />
```

Props can be accepted and used in component function as:

```
function Card(props) {
        return (
                <>
                        <h1> {props.cardNumber} </h1>
                        <p> {props.cardName} </p>
                        <button onClick={props.handleClick}> Click </button>
                </>
        )
}
```

All components are created in ‘src/components’.
There can be a ‘index.js’ file in ‘src/components’ which will import all the components within the folder and will export all at once.
So, in App.jsx or other files we can just export all components from a single file i.e. ‘src/components/index.js’

Hooks

useCallback()

useCallback() memoizes a callback function and returns a memoized version of the callback that only changes if one of the dependencies has changed.

It is used if dependencies doesn’t change too often.

Optimizes performance by preserving stable function references across renders, useful when passing callbacks to child components, reducing unnecessary re-renders.

useEffect()

Runs the callback function on page load and every time a dependency value changes.

useRef()

It generates a reference variable of an element.
It's useful for storing values or accessing DOM elements without impacting rendering.
Creates a mutable reference that persists across renders without causing re-renders.

useId()

Generates a unique ID.

Custom hooks

We can create our own function, in ‘src/hooks’ and we can use it as a hook.

React Router DOM

Getting Started

- Install it with

  npm install react-router-dom
- Create a Layout component.
- Use <Outlet /> imported from react-router-dom for dynamic components.
- Create a router in main.jsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import {
  Route,
  createBrowserRouter,
  createRoutesFromElements,
  RouterProvider,
} from "react-router-dom";
import Layout from "./Layout.jsx";
import { AboutUs, Home } from "./components/";

const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path="/" element={<Layout />}>
      <Route path="" element={<Home />} />
      <Route path="about-us" element={<AboutUs />} />
    </Route>
  )
);

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

- For navigation instead of <a href="#"> we use <Link to="/"> or <NavLink to="/">.
- NavLink is a special type of link which know whether its active or not.

```
  <Link to="/" className=="css classes" >Home</Link>

  <NavLink to="/about-us"
    className={({ isActive }) =>`block py-2 pr-4
     ${ isActive ? "text-orange-700" : "" }`
    }
  > About Us </NavLink>
```

Dynamic Routing

In main.jsx

```
      <Route path="user" element={<User />}>
        <Route path=":uid" element={<User />} />
      </Route>
```

In User.jsx

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

function User() {
  const { uid } = useParams();
  return <div>User: {uid ? uid : "not found"}</div>;
}
```

Loader

In main.jsx

```

      <Route loader={gifLoader} path="Gif" element={<Gif />} />

```

In Gif.jsx

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

export default function Gif() {
  const data = useLoaderData();
  return <img src={data.url} width={300} alt="" />;
}

export async function gifLoader() {
  const response = await fetch("https://api.waifu.pics/sfw/cuddle");
  return response.json();
}
```

State Management

Context API – Step by step

Create Context: sampleContext.js

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

Create Context Provider and values: SampleContextProvider.jsx

```
import { sampleContext } from "./sampleContext";

function SampleContextProvider({ children }) {
  const [data, setData] = useState([]);
  return (
    <sampleContext.Provider value={{ data, setData }}>
      {children}
    </sampleContext.Provider>
  );
}
```

Wrap everything within Context Provider: App.jsx or any

```
import SampleContextProvider from "./context/SampleContextProvider";

function App() {
  return (
    <SampleContextProvider>
      <Layout />
    </SampleContextProvider>
  );
}
```

Use context: in any file

```
import { sampleContext } from "../../context/sampleContext";

function Sample() {
  const { data } = useContext(sampleContext);
  return <div>{data}</div>;
}
```

Context API – Shortened (Mostly used)

Creating Context, Context provider and custom hook to use context

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

const productsContext = createContext({
  products: [],
  alterProducts: () => {},
});

export const ProductsContextProvider = productsContext.Provider;

const useProductContext = () => useContext(productsContext);

export default useProductContext;
```

Create context values, wrap everything within Context Provider: App.jsx or any

```
import { ProductsContextProvider } from "./context/productsContext";

function App() {
  const [products, setProducts] = useState([]);
  function alterProducts(data) {
    setProducts(data);
  }
  return (
    <ProductsContextProvider value={{ products, alterProducts }}>
        <Layout />
    </ProductsContextProvider>
  );
}
```

Use context: in any file

```
import useProductContext from "../../context/productsContext";

function Products() {
  const { products } = useProductContext();
  return <div>{products}</div>;
}
```

Redux Toolkit

Prevented Prop-Drilling. A centralized space (store) for data.

Store: centralized space for data.
Reducer: object of actions/functionalities/controllers (add, update, etc).
useSelector(): gets data from store using functions and reducers.
useDispatch(): updates or adds data into store using function and reducers.

Start - Installation:

npm i @reduxjs/toolkit
npm i react-redux

Create store: store.js

```
import { configureStore } from "@reduxjs/toolkit";
import dataReducer from "../features/dataSlice";

export const store = configureStore({
  reducer: dataReducer,
});
```

Create Slice: features/dataSlice.js
initial state and collection of reducers

```
import { createSlice, nanoid } from "@reduxjs/toolkit";

const dataSlice = createSlice({
  name: "slice name",
  initialState: { data: [] },
  reducers: {
    addData: (state, action) => {
      let data = { id: nanoid(), text: action.payload };
      state.data.push(data);
    },
    updateData: (state, action) => {
      state.data = state.data.map((elem) =>
        elem.id == action.payload.id ? {...elem, text: action.payload.text} : elem
      );
    },
  },
});

export const { addData, updateData } = dataSlice.actions;
export default dataSlice.reducer;
```

Provide store (wrap everything in provider): main.jsx

```
import { Provider } from "react-redux";
import { store } from "./store/store.js";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Add/Update data:

```
import { useDispatch } from "react-redux";
import { addData } from "../../features/dataSlice";

function Add() {
  const [input, setInput] = useState("");
  const dispatch = useDispatch();
  const add = (e) => {
    e.preventDefault();
    dispatch(addData(input));
    setInput("");
  };
  return ( <> ... </> );
}
```

Get data:

```
import { useSelector } from "react-redux";

function AllData() {
  const data = useSelector((state) => state.data);
  return (
    <div>
      {data.map((d) => <li key={d.id}> {d.text} </li>)}
    </div>
  );
}
```