DAY 21: MORNING ASSESSMENT

1. What is code splitting in React and why is it beneficial for large applications?

Code splitting breaks your bundle into smaller chunks that are loaded on demand. Benefits: faster initial load, lower Time-to-Interactive, and less JS parsed/executed upfront; users only download code for the route they visit.

2. How do you implement route-based code splitting using React.lazy() and Suspense?

import { lazy, Suspense } from "react";

import { Routes, Route } from "react-router-dom";

const Home = lazy(() => import("./pages/Home"));

const Users = lazy(() => import("./pages/Users"));

export default function App() {

return (

<Suspense fallback={<div>Loading…</div>}>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/users" element={<Users />} />

</Routes>

</Suspense>

);

}

3. What is the role of the fallback prop in Suspense?

fallback is the UI shown while the lazy chunk is being fetched

4. How does code splitting improve initial page load time?

It reduces the size of the first JS payload so the browser downloads, parses, and executes less code before rendering.

5. Compare code splitting by route vs code splitting by component.

*By route*: split at page boundaries; simplest and usually largest wins for initial load.

*By component*: split big widgets used across routes so they load only when rendered, more granular but slightly more wiring.

6. What happens if a dynamically imported route component fails to load? How would you

handle that?

A failed dynamic import throws. Use an Error Boundary around the lazy route to show a friendly error and offer retry:

class Boundary extends React.Component {

state = { key: 0 };

static getDerivedStateFromError() { return {}; }

render() {

return this.state.error

? <button onClick={() => this.setState(s => ({ key: s.key + 1 }))}>Retry</button>

: <div key={this.state.key}>{this.props.children}</div>;

}

}

7. How does Webpack handle chunk naming in dynamic imports?

lazy(() => import(/\* webpackChunkName: "users-page" \*/ "./pages/Users"));

Webpack then outputs a file like users-page.[contenthash].js.

8. What is the default chunk naming strategy if you don’t specify webpackChunkName?

Webpack generates deterministic numeric/hashed chunk IDs .In CRA/typical setups you’ll see hashed filenames without readable names.

9. How would you lazy-load multiple components for the same route?

const Chart = lazy(() => import("./Chart"));

const Editor = lazy(() => import("./Editor"));

<Route

path="/dashboard"

element={

<Suspense fallback={<Skeleton/>}>

<DashboardLayout>

<Chart />

<Editor />

</DashboardLayout>

</Suspense>

}

/>

10. Can you apply code splitting in nested routes? If yes, how?

Lazy the component at any nesting level—parent route, child route, or both. Each lazy boundary loads the needed chunk only when that nested route renders.

11. What is Webpack Bundle Analyzer used for in React development?

Visualizes your bundles to see what’s inside-module sizes, vendor chunks, duplicates-so you can optimize load time.

12. How do you install and configure Webpack Bundle Analyzer?

npm i -D webpack-bundle-analyzer

const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');

module.exports = {

plugins: [new BundleAnalyzerPlugin({ analyzerMode: 'server' })],

};

13. What kind of insights can Webpack Bundle Analyzer provide about your build?

Per-chunk and per-module sizes (parsed/gzipped/brotli)

Dependency tree and duplicates

Which files contribute most to bundle size

Which chunks lazy-load and their sizes

14. How do you identify large dependencies in your Webpack bundle?

Open the treemap report, sort by size, and inspect big rectangles . The side panel shows exact sizes and which chunk they’re in.

15. What strategies can you apply after identifying large bundles in Webpack?

Use dynamic imports for heavy screens/widgets

Use splitChunks to vendor-split and dedupe

Enable tree shaking

Replace heavy libs with lighter alternatives

16. How do you run Webpack Bundle Analyzer in development mode vs production mode?

*Dev*: enable plugin with analyzerMode: 'server'—quick iterations, but sizes are not minified.

*Prod*: build with production mode and run analyzer on the output for realistic (minified) sizes.

17. What is the difference between static and server modes in Webpack Bundle Analyzer?

*server*: starts a local server with an interactive UI.

*static*: emits a standalone HTML report you can open/share

18. How do you exclude certain packages from Webpack’s analysis?

Use plugin options like excludeAssets to hide assets. externals or IgnorePlugin.

19. How can tree shaking help after analyzing your Webpack bundle?

Use ES modules, mark side-effect-free code (package.json → "sideEffects": false or array), avoid common shake-blockers

20. How does Webpack’s splitChunks configuration interact with bundle analysis results?

Tune splitChunks.cacheGroups based on what you saw in the report; adjust minSize, maxSize, and maxInitialRequests to balance chunk sizes and load concurrency.

21. What does “lifting state up” mean in React?

Lifting state up moves shared state to the nearest common ancestor so multiple components can read/update the same source.

22. Why might two sibling components need state to be lifted?

When two siblings must stay in sync, keeping state in each causes divergence. Lift it to their parent so both get consistent props.

23. How do you pass data from a child component to a parent when lifting state?

Give the child a callback from the parent

function Parent() {

const [value, setValue] = useState("");

return <Child value={value} onChange={setValue} />;

}

function Child({ value, onChange }) {

return <input value={value} onChange={e => onChange(e.target.value)} />;

}

24. What is the main drawback of lifting too much state?

The parent becomes a “god component” and many children re-render on every change. This hurts performance and increases coupling.

25. How can lifting state up help prevent prop drilling?

Lift to a higher ancestor *or* to context at that level; then pass down only where needed instead of threading props through many layers.

26. Give an example of a form with multiple inputs where state lifting is used.

function Form() {

const [form, setForm] = useState({ name: "", email: "" });

const setField = (k, v) => setForm(f => ({ ...f, [k]: v }));

return (

<>

<NameInput value={form.name} onChange={v => setField("name", v)} />

<EmailInput value={form.email} onChange={v => setField("email", v)} />

<Summary data={form} />

</>

);

}

27. How do you prevent unnecessary re-renders when lifting state up?

Split state so unrelated parts don’t trigger each other

Use React.memo on children

Pass stable callbacks with useCallback and memoized derived data with useMemo

Consider context selectors or libraries like Zustand/Jotai for finer granularity

28. How do you combine state lifting with context to avoid deep prop passing?

const FormCtx = createContext();

function FormProvider({ children }) {

const [form, setForm] = useState({...});

const setField = useCallback((k,v)=>setForm(f=>({...f,[k]:v})),[]);

const value = useMemo(()=>({form,setField}),[form,setField]);

return <FormCtx.Provider value={value}>{children}</FormCtx.Provider>;

}

29. When lifting state, why might you use useCallback in the parent?

Stable function identities prevent memoized children from re-rendering due solely to new callback references every render.

30. How can you lift state without breaking controlled form elements?

Always pass value and onChange from the parent. In the child, never keep a competing internal state for that same input