**Code Splitting by Route**

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

Code splitting in React is a technique that divides a large JavaScript bundle into smaller, more manageable chunks, which are then loaded on demand rather than all at once.

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

```
import React, { Suspense, lazy } from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
const Home = lazy(() => import("./pages/Home"));
const About = lazy(() => import("./pages/About"));
const Contact = lazy(() => import("./pages/Contact"));
function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
        </Routes>
      </Suspense>
    </Router>
  );
}
export default App;
```

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

The fallback prop defines what to show while a lazy-loaded component is still being fetched. The fallback prop is a temporary placeholder that improves UX by avoiding a blank screen while React fetches lazy-loaded components.

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

Code splitting improves initial page load time because it divides the application into smaller bundles. Instead of loading the entire code of the app at once, only the code required for the first page is loaded. This makes the initial download smaller and faster. Other parts of the app are loaded only when the user visits them, so the first page becomes interactive quickly.

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

Code splitting by route means dividing the code based on application pages or routes. Each page's code is loaded only when the user navigates to that route.

Code splitting by component means splitting at a smaller level, where individual heavy components inside a page are lazy-loaded only when required.

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

If a dynamically imported route component fails to load, it typically results in a "Failed to fetch dynamically imported module" error or similar.

We should wrap lazy-loaded components in <Suspense> and an Error Boundary.

```
import React, { Suspense, lazy } from "react";

const LazyComponent = lazy(() => import("./NotExist"));

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: false };
  }
  static getDerivedStateFromError() {
    return { error: true };
  }
```

```
  render() {
    if (this.state.error) {
      return <h3>Could not load component</h3>;
    }
    return this.props.children;
  }
}
function App() {
  return (
    <ErrorBoundary>
      <Suspense fallback={<h3>⌛ Loading...</h3>}>
        <LazyComponent />
      </Suspense>
    </ErrorBoundary>
  );
}
export default App;
```

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

Webpack handles chunk naming in dynamic imports primarily through "magic comments" within the import() syntax. Default Behavior (Automatic Naming).

When a dynamic import() is used without any specific naming instruction, Webpack assigns a unique, numerical ID to the generated chunk. This results in output files like 0.bundle.js, 1.bundle.js, etc. Explicit Naming with webpackChunkName.

To provide a more descriptive and human-readable name for a dynamically imported chunk, the /* webpackChunkName: "yourChunkName" */ magic comment is used directly within the import() call.

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

Webpack falls back to its default chunk naming strategy.

Webpack generates id/hash-based chunk filenames derived from module ids, e.g. src_pages_About_jsx.

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

Use React.lazy() to dynamically import each component you want to lazy-load.

Use React.Suspense to wrap all the lazy-loaded components for a given route. This allows us to provide a fallback UI while the components are being loaded.

```
const ComponentA = React.lazy(() => import('./ComponentA'));
const ComponentB = React.lazy(() => import('./ComponentB'));


import React, { Suspense } from 'react';
import { Route, Routes } from 'react-router-dom';
const ComponentA = React.lazy(() => import('./ComponentA'));
const ComponentB = React.lazy(() => import('./ComponentB'));
function MyRouteComponent() {
 return (
  <Suspense fallback={<div>Loading components...</div>}>
   <ComponentA />
   <ComponentB />
  </Suspense>
 );
}
function App() {
 return (
  <Routes>
   <Route path="/my-route" element={<MyRouteComponent />} />
  </Routes>
```

```
  );
}
```

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

Yes, code splitting can be applied to nested routes using React.lazy and Suspense in conjunction with a routing library like React Router.

```
import { Suspense, lazy } from 'react';
import { Routes, Route } from 'react-router-dom';
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Settings = lazy(() => import('./pages/Settings'));
const UserProfile = lazy(() => import('./pages/UserProfile'));
<Routes>
  <Route path="/dashboard" element={
    <Suspense fallback={<div>Loading Dashboard...</div>}>
      <Dashboard />
    </Suspense>
  }>
    {/* Nested Route */}
    <Route path="settings" element={
      <Suspense fallback={<div>Loading Settings...</div>}>
        <Settings />
      </Suspense>
    } />
    <Route path="profile" element={
      <Suspense fallback={<div>Loading User Profile...</div>}>
        <UserProfile />
      </Suspense>
    } />
```

```
    </Route>
  </Routes>
```

**Webpack Bundle Analyzer**

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

Webpack Bundle Analyzer is used to analyze, optimize, and debug bundle size in React applications for faster performance.

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

npm install --save-dev webpack-bundle-analyzer

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

- Size of each module/dependency in the bundle
- Treemap visualization of bundle contents
- Duplicate dependencies detection
- Entry point and chunk analysis
- Bundle growth tracking over builds

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

We can open the analyzer's treemap and see which libraries occupy the most space.

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

- Code Splitting
- Tree Shaking
- Replace Heavy Libraries
- Dynamic Imports
- Bundle Externalization (CDN)
- Optimize Images & Assets
- Deduplicate Dependencies

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

Development Mode

```
const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');
module.exports = {
  mode: "development",
  plugins: [
    new BundleAnalyzerPlugin()
  ]
};
npm run start
```

Production Mode

```
const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');
module.exports = {
  mode: "production",
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: "static",
      reportFilename: "report.html",
      openAnalyzer: true
    })
  ]
};
npm run build
```

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

- static: generates an HTML report you can open later (no server).
- server: spins up a local HTTP UI that updates after each build.

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

We use the excludeAssets option with a regex or function to skip certain packages or files from being analyzed.

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

Tree shaking, after analyzing your Webpack bundle, helps reduce its size by eliminating unused code.

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

Webpack's splitChunks works with bundle analysis by moving large dependencies into separate chunks. This reduces the main bundle size, improves caching, and the analyzer helps confirm the split worked.

**State Lifting (State Up)**

21. What does "lifting state up" mean in React?

It means moving state from a child component into a common parent so multiple components can share it.

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

Because siblings can't directly share state; putting it in the parent lets both access and update it.

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

By passing a callback function as a prop to the child, and the child calls it with data.

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

It can cause unnecessary re-renders of many components and make the parent too complex.

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

Shared state in the parent can be passed only where needed, instead of drilling props through many layers.

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

```
import React, { useState } from "react";
function InputField({ label, value, onChange }) {
  return (
    <div>
      <label>{label}: </label>
      <input value={value} onChange={(e) => onChange(e.target.value)} />
    </div>
  );
}
function Form() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  return (
    <div>
      <h2>Registration Form</h2>
      <InputField label="Name" value={name} onChange={setName} />
      <InputField label="Email" value={email} onChange={setEmail} />
      <p>
        Preview → Name: {name}, Email: {email}
      </p>
    </div>
  );
}
export default Form;
```

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

We can use React.memo, useCallback, or useMemo so child components only re-render when their relevant props change.

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

Lift state to a parent, then provide it with React Context, so children at any depth can consume it without prop chains.

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

To memoize callback functions so they don’t get recreated on every render, reducing child re-renders.

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

We can keep input values in the parent’s state and pass them down as value with onChange handlers ensuring inputs remain controlled.