**React Framework Fundamentals**

**1. Compare React and Angular**

**Core Architectural Differences:**

* **React:**
  + Library for building UI components.
  + Focuses on "View" in the MVC (Model-View-Controller) pattern.
  + Virtual DOM for efficient updates.
  + Unidirectional data flow.
  + Requires additional libraries (e.g., Redux, React Router) for full functionality.
* **Angular:**
  + Full-fledged framework.
  + Two-way data binding.
  + Built-in features like routing, HTTP services, and form handling.
  + Uses a real DOM and change detection.
  + Includes dependency injection for better modularity.

**Learning Curve:**

* **React:** Easier to learn due to its simplicity and focus on UI. Requires learning additional libraries for complete functionality.
* **Angular:** Steeper learning curve due to its comprehensive feature set and use of TypeScript.

**Scenarios:**

* **React:**
  + Lightweight applications.
  + Projects requiring high flexibility.
  + When a small team is involved.
* **Angular:**
  + Large-scale enterprise applications.
  + Projects requiring built-in solutions.
  + Teams familiar with TypeScript and MVC patterns.

**2. React Project Structure**

**Landing Page/Component in Create React App:**

* App.js is the default landing component.

**Entry Point of a React Application:**

* index.js (renders the root component using ReactDOM.createRoot).

**Initial Rendering Configuration:**

* Configured in index.js by rendering the App component:

ReactDOM.createRoot(document.getElementById('root')).render(<App />);

**3. Package.json Deep Dive**

**Essential Fields:**

* name: Project name.
* version: Project version.
* scripts: Commands to run the project.
* dependencies: Required for the application to run.
* devDependencies: Required only during development.
* browserslist: Specifies browser compatibility.

**Dependencies vs DevDependencies:**

* **Dependencies:** Required in production (e.g., React, React Router).
* **DevDependencies:** Required in development (e.g., Webpack, Babel).

**Version Numbers:**

* ^: Accepts minor updates (e.g., ^1.2.3 allows 1.3.0).
* ~: Accepts patch updates (e.g., ~1.2.3 allows 1.2.4).

**Scripts:**

* Define custom commands, e.g.,

"start": "react-scripts start"

Run with npm start.

**Purpose of "browserslist":**

* Ensures compatibility with specified browsers for building and transpiling.

**Routing**

**4. React Router Implementation**

**Setting up React Router DOM:**

1. Install: npm install react-router-dom
2. Import and wrap the app:
3. import { BrowserRouter } from 'react-router-dom';
4. ReactDOM.render(
5. <BrowserRouter>
6. <App />
7. </BrowserRouter>,
8. document.getElementById('root')

);

**BrowserRouter:**

* Provides routing context and should be placed at the root level.

**Defining Routes:**

* **Parent routes:**

<Route path="/dashboard" element={<Dashboard />} />

* **Child routes:**
* <Route path="/dashboard">
* <Route path="profile" element={<Profile />} />

</Route>

* **Nested routes:** Combine parent and child routes in Outlet.

**Route Parameters:**

* Define:

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

* Access:

const { id } = useParams();

**Protected Routes:**

* Example:
* const ProtectedRoute = ({ element }) => {
* const isAuth = useAuth();
* return isAuth ? element : <Navigate to="/login" />;

};

**React Hooks and State Management**

**5. React Hooks Overview**

**Problem Solved by Hooks:**

* Simplified state and lifecycle management.
* Avoided "wrapper hell" and improved readability.

**Why Replace Class Components:**

* Simplified syntax.
* Eliminated the need for this keyword.
* Easier to manage state and side effects.

**Rules of Hooks:**

1. Only call hooks at the top level.
2. Only call hooks inside React functions.

**6. State Management**

**Defining State Variable:**

const [number, setNumber] = useState(0);

**Why Not Update State Directly:**

* Direct updates won’t trigger re-renders.

**Correct Way to Update State:**

setNumber(prev => prev + 1);

**useState vs useReducer:**

* **useState:** Simple state management.
* **useReducer:** Complex state logic with multiple actions.

**7. useEffect Deep Dive**

**Purpose:**

* Handle side effects like API calls and subscriptions.

**Dependency Array Scenarios:**

1. **Empty array:** Runs once on mount.
2. **Specific dependencies:** Runs when dependencies change.
3. **No array:** Runs on every render.

**Cleanup:**

* Use for subscriptions or timers:
* useEffect(() => {
* const timer = setInterval(() => console.log('tick'), 1000);
* return () => clearInterval(timer);

}, []);

**Common Pitfalls:**

* Missing dependencies in the array.
* Infinite re-renders from incorrect dependencies.

**8. Immutability**

**Definition:**

* Prevents direct modification of objects/arrays.

**Importance:**

* Ensures predictable state updates.
* Avoids accidental mutations.

**Maintaining Immutability:**

* **Arrays:**

const newArray = [...oldArray, newItem];

* **Objects:**

const newObject = { ...oldObject, newProp: value };

**JSX and Components**

**9. JSX Understanding**

**What is JSX:**

* Syntax extension for JavaScript to write HTML-like code.

**JSX Limitations:**

* Must return a single parent element.
* Cannot use reserved keywords like class (use className).

**Differences from HTML:**

* className instead of class.
* htmlFor instead of for.

**Rules for JSX:**

* Close all tags.
* Wrap multiple elements in a single parent.

**API Integration**

**10. API Implementation**

**Common Packages:**

* axios
* fetch

**Axios vs Fetch:**

* **Axios:** Simplified syntax, automatic JSON conversion.
* **Fetch:** Native but requires manual configuration.

**Handling API Errors:**

try {

const response = await axios.get('/api');

} catch (error) {

console.error(error);

}

**API Caching:**

* Use react-query or cache results in state.

**11. Lazy Loading**

**Definition:**

* Loading components or data on demand.

**Differences from Eager Loading:**

* **Lazy:** On demand.
* **Eager:** Immediately.

**Default Behavior:**

* Eager loading.

**Lazy Loading and File Handling in React**

**11. Lazy Loading**

**What is lazy loading?** Lazy loading is a design pattern that delays the loading of resources until they are required. In React, it’s often used to load components or images only when they come into view, improving initial load time.

**How does it differ from eager loading?**

* **Lazy loading:** Resources are loaded only when needed.
* **Eager loading:** All resources are loaded upfront, which can increase initial load time.

**Which is the default and why?** Eager loading is the default since it ensures all required resources are available immediately after the page loads. However, lazy loading can be configured to improve performance.

**When should you implement lazy loading?** Use lazy loading for:

* Large components or modules
* Heavy images or media files
* Routes not immediately visible

**How do you implement code splitting with lazy loading?** React provides React.lazy and Suspense to implement lazy loading:

import React, { Suspense } from 'react';

const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {

return (

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

<LazyComponent />

</Suspense>

);

}

**File Upload Implementation**

**12. Basic File Upload Flow**

**How do you handle file input in React?** Use the <input> element with type="file":

function FileUpload() {

const handleFileChange = (event) => {

const file = event.target.files[0];

console.log(file);

};

return <input type="file" onChange={handleFileChange} />;

}

**How do you validate file types?** Check the file’s type or extension:

const handleFileChange = (event) => {

const file = event.target.files[0];

const allowedTypes = ['image/png', 'image/jpeg'];

if (!allowedTypes.includes(file.type)) {

alert('Only PNG and JPEG files are allowed.');

return;

}

};

**How do you handle multiple file uploads?** Set the multiple attribute on the input:

function MultiFileUpload() {

const handleFilesChange = (event) => {

const files = event.target.files;

console.log(files);

};

return <input type="file" multiple onChange={handleFilesChange} />;

}

**13. File Upload Architecture**

**Scenario: Implement a profile photo upload feature**

**Frontend implementation steps:**

1. Create a form with a file input.
2. Handle file selection and preview.
3. Send the file to the backend using axios or fetch.

import React, { useState } from 'react';

import axios from 'axios';

function ProfilePhotoUpload() {

const [file, setFile] = useState(null);

const handleFileChange = (e) => setFile(e.target.files[0]);

const handleUpload = async () => {

const formData = new FormData();

formData.append('file', file);

try {

await axios.post('/upload', formData);

alert('Upload successful');

} catch (err) {

console.error(err);

alert('Upload failed');

}

};

return (

<div>

<input type="file" onChange={handleFileChange} />

<button onClick={handleUpload}>Upload</button>

</div>

);

}

**Backend handling:** Use a Node.js server with multer:

const express = require('express');

const multer = require('multer');

const upload = multer({ dest: 'uploads/' });

const app = express();

app.post('/upload', upload.single('file'), (req, res) => {

res.send('File uploaded successfully');

});

app.listen(3000, () => console.log('Server started on port 3000'));

**Storage solutions (S3 vs local storage):**

* **S3:** Highly scalable, secure, and accessible from anywhere.
* **Local storage:** Suitable for small-scale or local applications but limited in scalability.

**Database design considerations:** Store file metadata (filename, size, type, and URL) instead of the file itself.

**14. Database Design for Files**

**What information should be stored in the database?**

* File name
* File size
* File type
* Upload date
* File URL or path

**How do you store file metadata?** Create a files table or collection:

{

"id": 1,

"filename": "profile.jpg",

"type": "image/jpeg",

"size": 204800,

"url": "https://s3.amazonaws.com/bucketname/profile.jpg",

"uploaded\_at": "2025-01-15T12:00:00Z"

}

**How do you handle file relationships?** Establish foreign key relationships. For example, link files to users:

CREATE TABLE users (

id INT PRIMARY KEY,

name VARCHAR(255)

);

CREATE TABLE files (

id INT PRIMARY KEY,

user\_id INT,

filename VARCHAR(255),

FOREIGN KEY (user\_id) REFERENCES users(id)

);

**15. AWS S3 Integration**

**How do you upload files to S3?** Use the AWS SDK:

const AWS = require('aws-sdk');

const s3 = new AWS.S3();

const uploadToS3 = (file) => {

const params = {

Bucket: 'your-bucket-name',

Key: file.name,

Body: file,

};

s3.upload(params, (err, data) => {

if (err) console.error(err);

else console.log(`File uploaded: ${data.Location}`);

});

};

**What is a pre-signed URL?** A pre-signed URL allows clients to upload files directly to S3 without exposing your AWS credentials.

**What is an unsigned URL?** An unsigned URL is a public URL for accessing files in S3 without any restrictions.

**When would you use each approach?**

* **Pre-signed URL:** For secure, temporary access to upload/download files.
* **Unsigned URL:** For publicly accessible files like images on a website.

**16. Large File Handling**

**Scenario: Uploading a 3GB movie file**

**How would you handle large file uploads?**

* Use chunked uploads.
* Implement resumable uploads.

**What are the API limitations?** APIs may have timeouts or size limits (e.g., 5GB for a single S3 PUT request).

**How would you implement chunked upload?** Split the file into smaller parts and upload each chunk:

async function uploadFile(file) {

const chunkSize = 10 \* 1024 \* 1024; // 10MB

const totalChunks = Math.ceil(file.size / chunkSize);

for (let i = 0; i < totalChunks; i++) {

const start = i \* chunkSize;

const end = Math.min(start + chunkSize, file.size);

const chunk = file.slice(start, end);

await uploadChunk(chunk, i);

}

}

**How would you show upload progress?** Use XMLHttpRequest or axios with progress tracking:

axios.post('/upload', formData, {

onUploadProgress: (progressEvent) => {

const percentCompleted = Math.round((progressEvent.loaded \* 100) / progressEvent.total);

console.log(`${percentCompleted}% uploaded`);

},

});

**17. File Transfer Methods**

**How do you send raw image files via API?** Send the file as multipart/form-data:

const formData = new FormData();

formData.append('file', file);

axios.post('/upload', formData);

**What are the different approaches to file transfer?**

* **multipart/form-data:** Encodes files for uploading.
* **base64 encoding:** Converts files to a string format.
* **binary transfer:** Transfers raw binary data.

**Compare:**

* **multipart/form-data:** Best for forms and file uploads.
* **base64 encoding:** Increases file size; use for small files.
* **binary transfer:** Efficient for large files.

**When would you use each approach?**

* Use multipart/form-data for typical uploads.
* Use base64 for embedding small images.
* Use binary transfer for large or high-performance scenarios.

**Advanced File Transfer and Best Practices**

**17. File Transfer Methods:**

**How do you send raw image files via API?**

* Use the multipart/form-data content type.
* Send the file in the request body using tools like FormData in JavaScript.

**What are the different approaches to file transfer?**

1. multipart/form-data
2. Base64 encoding
3. Binary transfer

**Compare:**

* **multipart/form-data:** Efficient for large files, works well with REST APIs.
* **Base64 encoding:** Increases file size by ~33%, suitable for embedding small files.
* **Binary transfer:** Fast and efficient but requires proper API configuration.

**When would you use each approach?**

* Use multipart/form-data for standard file uploads.
* Use Base64 for embedding images in JSON responses.
* Use binary for streaming large files.

**18. Performance Optimization:**

**How would you optimize file uploads for:**

* **Multiple concurrent users:**
  + Use a load balancer to distribute traffic.
  + Implement horizontal scaling with multiple servers.
* **Large files:**
  + Use chunked uploads.
  + Enable compression and server-side caching.
* **Different file types:**
  + Validate and compress files based on type.
  + Use Content Delivery Networks (CDNs).

**19. Security Considerations:**

**How do you implement secure file uploads?**

* Use HTTPS to encrypt data in transit.
* Validate file types and sizes on both client and server.
* Store files in a secure location, like an isolated server or S3 bucket.

**How do you prevent malicious file uploads?**

* Use file type whitelisting.
* Scan files for viruses.
* Limit upload sizes and restrict executable file types.

**How do you handle file access permissions?**

* Use role-based access control (RBAC).
* Implement signed URLs for temporary access.

**20. Error Handling:**

**How do you handle upload failures?**

* Retry failed uploads using exponential backoff.
* Notify users of failure and provide retry options.

**How do you implement retry mechanisms?**

* Use libraries like axios with retry functionality.
* Implement retry logic with setTimeout or recursive functions.

**How do you handle network interruptions?**

* Pause the upload and resume from the last successful chunk.
* Use service workers for offline support.

**21. Scale Considerations:**

**Scenario: System needs to handle 1000+ concurrent users:**

* Use cloud-based services for scalability (AWS, Azure, GCP).
* Optimize database queries and indexes.
* Implement horizontal scaling and load balancing.

**What caching strategies would you implement?**

* Use CDNs for static content.
* Implement server-side caching with Redis or Memcached.

**How would you handle rate limiting?**

* Use tools like Nginx or API Gateway to throttle requests.
* Implement user-specific rate limits in the backend.

**22. File Management System:**

**Design a complete file management system with:**

* **Upload capabilities:**
  + Use multipart/form-data for uploading.
  + Validate files on both client and server.
* **File organization:**
  + Use directories or tags for categorization.
  + Maintain a structured database schema.
* **Search functionality:**
  + Implement full-text search using tools like Elasticsearch.
  + Use metadata for advanced filtering.
* **Access control:**
  + Implement RBAC.
  + Use signed URLs for secure access.
* **Version control:**
  + Store versions in separate folders or use version numbers in filenames.
  + Maintain a database record of version history.

**23. Development Guidelines:**

**What are React best practices for:**

* **Component structure:**
  + Keep components small and reusable.
  + Organize files by feature or domain.
* **State management:**
  + Use useState and useReducer for local state.
  + Use libraries like Redux or Zustand for global state.
* **Performance optimization:**
  + Use React.memo to prevent unnecessary re-renders.
  + Implement lazy loading and code splitting.
* **Error handling:**
  + Use error boundaries for UI errors.
  + Catch and handle API errors in the code.
* **Testing:**
  + Write unit tests with Jest.
  + Use React Testing Library for component tests.

**24. Production Considerations:**

**What factors should be considered before deploying to production?**

* Optimize the build with minification and tree-shaking.
* Set proper environment variables.
* Configure a secure HTTPS connection.

**How do you handle environment variables?**

* Store them in a .env file.
* Use process.env to access them in the code.

**How do you implement monitoring and logging?**

* Use tools like Sentry or LogRocket for error monitoring.
* Implement server-side logging with tools like Winston or Bunyan.