**Fullstack Developer Interview Questions & Answers**

**Frontend Technologies**

**React.js**

**Basic Level Questions**

**Q1: What is React and what are its key features?**

**Answer:** React is an open-source JavaScript library developed by Facebook for building user interfaces, particularly single-page applications. It follows a component-based architecture.

**Key Features:**

* **Component-Based Architecture**: Build encapsulated components that manage their own state
* **Virtual DOM**: Creates a virtual representation of the real DOM for efficient updates
* **JSX**: JavaScript syntax extension that allows writing HTML-like code in JavaScript
* **One-way Data Binding**: Data flows down from parent to child components
* **Declarative**: Describe what the UI should look like for any given state

**Example:**

// Simple React component

function Welcome({ name }) {

return <h1>Hello, {name}!</h1>;

}

// Usage

<Welcome name="John" />

**Q2: Explain the difference between functional and class components.**

**Answer:**

**Class Components:**

* ES6 classes that extend React.Component
* Have lifecycle methods
* Use this.state for state management
* More verbose syntax

**Functional Components:**

* Simple JavaScript functions that return JSX
* Use React Hooks for state and lifecycle management
* Cleaner, more concise syntax
* Better performance (no class instantiation overhead)

**Examples:**

// Class Component

class ClassComponent extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

componentDidMount() {

console.log('Component mounted');

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.setState({ count: this.state.count + 1 })}>

Increment

</button>

</div>

);

}

}

// Functional Component (Modern approach)

import { useState, useEffect } from 'react';

function FunctionalComponent() {

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

useEffect(() => {

console.log('Component mounted');

}, []);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>

Increment

</button>

</div>

);

}

**Q3: What are props and how do you pass data between components?**

**Answer:** Props (properties) are read-only data passed from parent components to child components. They enable data flow and component communication.

**Key Points:**

* Props are immutable within the receiving component
* Data flows one-way (parent to child)
* Can pass any data type: strings, numbers, objects, functions, etc.

**Example:**

// Parent Component

function App() {

const user = {

name: 'John Doe',

email: 'john@example.com',

age: 30

};

const handleClick = () => {

alert('Button clicked!');

};

return (

<div>

<UserProfile

user={user}

isActive={true}

onButtonClick={handleClick}

/>

</div>

);

}

// Child Component

function UserProfile({ user, isActive, onButtonClick }) {

return (

<div className={isActive ? 'active' : 'inactive'}>

<h2>{user.name}</h2>

<p>Email: {user.email}</p>

<p>Age: {user.age}</p>

<button onClick={onButtonClick}>

Click Me

</button>

</div>

);

}

// Props destructuring with default values

function Button({ text = 'Click', variant = 'primary', onClick }) {

return (

<button

className={`btn btn-${variant}`}

onClick={onClick}

>

{text}

</button>

);

}

**Medium Level Questions**

**Q4: Explain React component lifecycle methods and their equivalents in functional components.**

**Answer:**

**Class Component Lifecycle Methods:**

1. **Mounting Phase:**
   * constructor(): Initialize state and bind methods
   * componentDidMount(): Execute after component is mounted
2. **Updating Phase:**
   * componentDidUpdate(): Execute after component updates
   * getSnapshotBeforeUpdate(): Capture info before DOM update
3. **Unmounting Phase:**
   * componentWillUnmount(): Cleanup before component unmounts

**Functional Component Equivalents using Hooks:**

// Class Component

class LifecycleExample extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0, data: null };

}

async componentDidMount() {

console.log('Component mounted');

// Fetch data

const response = await fetch('/api/data');

const data = await response.json();

this.setState({ data });

}

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

console.log('Count updated:', this.state.count);

}

}

componentWillUnmount() {

console.log('Component will unmount');

// Cleanup subscriptions, timers, etc.

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.setState({ count: this.state.count + 1 })}>

Increment

</button>

</div>

);

}

}

// Functional Component Equivalent

import { useState, useEffect } from 'react';

function LifecycleExample() {

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

const [data, setData] = useState(null);

// componentDidMount equivalent

useEffect(() => {

console.log('Component mounted');

const fetchData = async () => {

const response = await fetch('/api/data');

const result = await response.json();

setData(result);

};

fetchData();

// componentWillUnmount equivalent (cleanup function)

return () => {

console.log('Component will unmount');

// Cleanup subscriptions, timers, etc.

};

}, []); // Empty dependency array = componentDidMount + componentWillUnmount

// componentDidUpdate equivalent

useEffect(() => {

console.log('Count updated:', count);

}, [count]); // Runs when count changes

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>

Increment

</button>

</div>

);

}

**Q5: What are React Hooks? Explain useState, useEffect, and useContext.**

**Answer:**

React Hooks are functions that let you use state and other React features in functional components. They were introduced in React 16.8.

**1. useState Hook:** Manages local component state.

import { useState } from 'react';

function Counter() {

// useState returns [currentState, setterFunction]

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

const [name, setName] = useState('');

const [user, setUser] = useState({ name: '', email: '' });

const handleIncrement = () => {

setCount(count + 1);

// Or use functional update for complex state updates

setCount(prevCount => prevCount + 1);

};

const handleUserUpdate = (field, value) => {

setUser(prevUser => ({

...prevUser,

[field]: value

}));

};

return (

<div>

<p>Count: {count}</p>

<button onClick={handleIncrement}>Increment</button>

<input

value={name}

onChange={(e) => setName(e.target.value)}

placeholder="Enter name"

/>

</div>

);

}

**2. useEffect Hook:** Handles side effects (data fetching, subscriptions, DOM manipulation).

import { useState, useEffect } from 'react';

function UserProfile({ userId }) {

const [user, setUser] = useState(null);

const [loading, setLoading] = useState(true);

const [windowWidth, setWindowWidth] = useState(window.innerWidth);

// Effect with cleanup (componentDidMount + componentWillUnmount)

useEffect(() => {

const handleResize = () => setWindowWidth(window.innerWidth);

window.addEventListener('resize', handleResize);

// Cleanup function

return () => window.removeEventListener('resize', handleResize);

}, []); // Empty deps = run once

// Effect with dependency (componentDidUpdate)

useEffect(() => {

if (!userId) return;

const fetchUser = async () => {

setLoading(true);

try {

const response = await fetch(`/api/users/${userId}`);

const userData = await response.json();

setUser(userData);

} catch (error) {

console.error('Failed to fetch user:', error);

} finally {

setLoading(false);

}

};

fetchUser();

}, [userId]); // Runs when userId changes

// Effect without cleanup (componentDidUpdate)

useEffect(() => {

document.title = user ? `Profile: ${user.name}` : 'Profile';

}, [user]);

if (loading) return <div>Loading...</div>;

return (

<div>

<h1>{user?.name}</h1>

<p>Window width: {windowWidth}px</p>

</div>

);

}

**3. useContext Hook:** Consumes context values without nesting Consumer components.

import { createContext, useContext, useState } from 'react';

// Create Context

const ThemeContext = createContext();

const UserContext = createContext();

// Provider Component

function App() {

const [theme, setTheme] = useState('light');

const [user, setUser] = useState({ name: 'John', role: 'admin' });

return (

<ThemeContext.Provider value={{ theme, setTheme }}>

<UserContext.Provider value={{ user, setUser }}>

<Header />

<MainContent />

</UserContext.Provider>

</ThemeContext.Provider>

);

}

// Consumer Components

function Header() {

const { theme, setTheme } = useContext(ThemeContext);

const { user } = useContext(UserContext);

return (

<header className={`header ${theme}`}>

<h1>Welcome, {user.name}</h1>

<button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>

Toggle Theme

</button>

</header>

);

}

function MainContent() {

const { theme } = useContext(ThemeContext);

return (

<main className={`main ${theme}`}>

<p>Main content goes here</p>

</main>

);

}

**High Level Questions**

**Q6: Design a custom hook for managing API calls with loading, error, and data states.**

**Answer:**

Custom hooks encapsulate stateful logic that can be reused across components. Here's a comprehensive API management hook:

import { useState, useEffect, useCallback, useRef } from 'react';

// Custom hook for API calls

function useApi(url, options = {}) {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(false);

const [error, setError] = useState(null);

const abortControllerRef = useRef(null);

const fetchData = useCallback(async (customUrl, customOptions = {}) => {

const finalUrl = customUrl || url;

const finalOptions = { ...options, ...customOptions };

if (!finalUrl) {

setError(new Error('URL is required'));

return;

}

// Cancel previous request

if (abortControllerRef.current) {

abortControllerRef.current.abort();

}

// Create new abort controller

abortControllerRef.current = new AbortController();

setLoading(true);

setError(null);

try {

const response = await fetch(finalUrl, {

...finalOptions,

signal: abortControllerRef.current.signal,

headers: {

'Content-Type': 'application/json',

...finalOptions.headers,

},

});

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

const result = await response.json();

setData(result);

return result;

} catch (err) {

if (err.name !== 'AbortError') {

setError(err);

throw err;

}

} finally {

setLoading(false);

}

}, [url, options]);

// Auto-fetch on mount if URL is provided

useEffect(() => {

if (url) {

fetchData();

}

// Cleanup on unmount

return () => {

if (abortControllerRef.current) {

abortControllerRef.current.abort();

}

};

}, [fetchData]);

const refetch = useCallback(() => fetchData(), [fetchData]);

const reset = useCallback(() => {

setData(null);

setError(null);

setLoading(false);

}, []);

return {

data,

loading,

error,

refetch,

fetchData,

reset,

};

}

// Advanced version with caching

function useApiWithCache(url, options = {}) {

const cacheRef = useRef(new Map());

const [data, setData] = useState(null);

const [loading, setLoading] = useState(false);

const [error, setError] = useState(null);

const getCacheKey = (url, options) => {

return `${url}-${JSON.stringify(options)}`;

};

const fetchData = useCallback(async (customUrl, customOptions = {}) => {

const finalUrl = customUrl || url;

const finalOptions = { ...options, ...customOptions };

const cacheKey = getCacheKey(finalUrl, finalOptions);

// Check cache first

if (cacheRef.current.has(cacheKey)) {

const cachedData = cacheRef.current.get(cacheKey);

setData(cachedData);

return cachedData;

}

setLoading(true);

setError(null);

try {

const response = await fetch(finalUrl, {

...finalOptions,

headers: {

'Content-Type': 'application/json',

...finalOptions.headers,

},

});

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

const result = await response.json();

// Cache the result

cacheRef.current.set(cacheKey, result);

setData(result);

return result;

} catch (err) {

setError(err);

throw err;

} finally {

setLoading(false);

}

}, [url, options]);

useEffect(() => {

if (url) {

fetchData();

}

}, [fetchData]);

const clearCache = useCallback(() => {

cacheRef.current.clear();

}, []);

return {

data,

loading,

error,

refetch: fetchData,

clearCache,

};

}

// Usage examples

function UserList() {

const { data: users, loading, error, refetch } = useApi('/api/users');

if (loading) return <div>Loading users...</div>;

if (error) return <div>Error: {error.message}</div>;

return (

<div>

<button onClick={refetch}>Refresh</button>

<ul>

{users?.map(user => (

<li key={user.id}>{user.name}</li>

))}

</ul>

</div>

);

}

function UserProfile({ userId }) {

const {

data: user,

loading,

error,

fetchData

} = useApi(); // No initial URL

useEffect(() => {

if (userId) {

fetchData(`/api/users/${userId}`);

}

}, [userId, fetchData]);

const updateUser = async (updates) => {

try {

await fetchData(`/api/users/${userId}`, {

method: 'PUT',

body: JSON.stringify(updates),

});

} catch (error) {

console.error('Update failed:', error);

}

};

return (

<div>

{loading && <p>Loading...</p>}

{error && <p>Error: {error.message}</p>}

{user && (

<div>

<h2>{user.name}</h2>

<button onClick={() => updateUser({ status: 'active' })}>

Activate User

</button>

</div>

)}

</div>

);

}

**Next.js**

**Basic Level Questions**

**Q7: What is Next.js and what problems does it solve?**

**Answer:**

Next.js is a React framework that provides additional structure, features, and optimizations for building production-ready applications.

**Problems Next.js Solves:**

1. **SEO and Performance**: Server-side rendering improves SEO and initial page load
2. **Routing**: File-based routing system eliminates need for router configuration
3. **Bundle Optimization**: Automatic code splitting and optimization
4. **Development Experience**: Hot reload, built-in CSS support, TypeScript support
5. **Deployment**: Easy deployment with Vercel platform
6. **API Routes**: Built-in API functionality

**Key Features:**

// File-based routing

// pages/index.js → /

// pages/about.js → /about

// pages/blog/[slug].js → /blog/post-1, /blog/post-2, etc.

// pages/index.js

import { GetServerSideProps } from 'next';

export default function HomePage({ posts }) {

return (

<div>

<h1>Welcome to My Blog</h1>

{posts.map(post => (

<div key={post.id}>

<h2>{post.title}</h2>

<p>{post.excerpt}</p>

</div>

))}

</div>

);

}

// Server-side rendering

export const getServerSideProps: GetServerSideProps = async () => {

const res = await fetch('https://api.example.com/posts');

const posts = await res.json();

return {

props: {

posts,

},

};

};

// API Routes

// pages/api/users/[id].js

export default function handler(req, res) {

const { id } = req.query;

if (req.method === 'GET') {

// Fetch user by ID

res.status(200).json({ id, name: 'John Doe' });

} else if (req.method === 'PUT') {

// Update user

const updatedUser = { id, ...req.body };

res.status(200).json(updatedUser);

} else {

res.setHeader('Allow', ['GET', 'PUT']);

res.status(405).end(`Method ${req.method} Not Allowed`);

}

}

**Q8: Explain the difference between CSR, SSR, and SSG in Next.js.**

**Answer:**

**1. Client-Side Rendering (CSR):**

* JavaScript executes in the browser
* Initial HTML is minimal, content loaded via JS
* Good for dynamic, interactive applications
* SEO challenges, slower initial paint

// CSR Example - Traditional React approach

import { useState, useEffect } from 'react';

function ClientSideRendering() {

const [posts, setPosts] = useState([]);

const [loading, setLoading] = useState(true);

useEffect(() => {

fetch('/api/posts')

.then(res => res.json())

.then(data => {

setPosts(data);

setLoading(false);

});

}, []);

if (loading) return <div>Loading...</div>;

return (

<div>

{posts.map(post => (

<article key={post.id}>

<h2>{post.title}</h2>

<p>{post.content}</p>

</article>

))}

</div>

);

}

**2. Server-Side Rendering (SSR):**

* HTML generated on server for each request
* Better SEO and initial page load
* Higher server load
* Good for dynamic content that changes frequently

// SSR Example - getServerSideProps

import { GetServerSideProps } from 'next';

interface Post {

id: string;

title: string;

content: string;

author: string;

}

interface Props {

posts: Post[];

user: any;

}

export default function ServerSideRendering({ posts, user }: Props) {

return (

<div>

<h1>Welcome, {user.name}</h1>

{posts.map(post => (

<article key={post.id}>

<h2>{post.title}</h2>

<p>By: {post.author}</p>

<p>{post.content}</p>

</article>

))}

</div>

);

}

// This function runs on every request

export const getServerSideProps: GetServerSideProps = async (context) => {

const { req } = context;

// Access cookies, headers, etc.

const userToken = req.cookies.token;

// Fetch data that requires authentication or real-time data

const [postsRes, userRes] = await Promise.all([

fetch('https://api.example.com/posts'),

fetch('https://api.example.com/user', {

headers: { Authorization: `Bearer ${userToken}` }

})

]);

const posts = await postsRes.json();

const user = await userRes.json();

// If user is not authenticated, redirect

if (!user) {

return {

redirect: {

destination: '/login',

permanent: false,

},

};

}

return {

props: {

posts,

user,

},

};

};

**3. Static Site Generation (SSG):**

* HTML generated at build time
* Best performance and SEO
* Content is static until next build
* Perfect for blogs, documentation, marketing sites

// SSG Example - getStaticProps

import { GetStaticProps, GetStaticPaths } from 'next';

interface Post {

id: string;

title: string;

content: string;

date: string;

}

interface Props {

post: Post;

}

export default function BlogPost({ post }: Props) {

return (

<article>

<h1>{post.title}</h1>

<time>{post.date}</time>

<div dangerouslySetInnerHTML={{ \_\_html: post.content }} />

</article>

);

}

// Generate static paths at build time

export const getStaticPaths: GetStaticPaths = async () => {

const res = await fetch('https://api.example.com/posts');

const posts = await res.json();

const paths = posts.map((post: Post) => ({

params: { id: post.id },

}));

return {

paths,

fallback: 'blocking', // or false, true, 'blocking'

};

};

// Generate static props at build time

export const getStaticProps: GetStaticProps = async ({ params }) => {

const res = await fetch(`https://api.example.com/posts/${params?.id}`);

const post = await res.json();

if (!post) {

return {

notFound: true,

};

}

return {

props: {

post,

},

revalidate: 60, // ISR - revalidate every 60 seconds

};

};

**When to use each:**

* **CSR**: Dashboards, admin panels, apps with lots of user interaction
* **SSR**: E-commerce product pages, user-specific content, real-time data
* **SSG**: Blogs, documentation, marketing sites, content that doesn't change often

**JavaScript & TypeScript**

**Basic Level Questions**

**Q9: Explain the difference between var, let, and const.**

**Answer:**

**1. var:**

* Function-scoped or globally-scoped
* Hoisted and initialized with undefined
* Can be redeclared and reassigned
* Creates property on global object (in browser: window)

**2. let:**

* Block-scoped
* Hoisted but not initialized (Temporal Dead Zone)
* Cannot be redeclared in same scope
* Can be reassigned

**3. const:**

* Block-scoped
* Hoisted but not initialized (Temporal Dead Zone)
* Cannot be redeclared or reassigned
* Must be initialized at declaration
* Object/array contents can still be modified

**Examples:**

// Hoisting behavior

console.log(varVariable); // undefined (not error)

console.log(letVariable); // ReferenceError: Cannot access before initialization

console.log(constVariable); // ReferenceError: Cannot access before initialization

var varVariable = 'var';

let letVariable = 'let';

const constVariable = 'const';

// Scope differences

function scopeExample() {

if (true) {

var varScoped = 'var'; // Function scoped

let letScoped = 'let'; // Block scoped

const constScoped = 'const'; // Block scoped

}

console.log(varScoped); // 'var' - accessible

console.log(letScoped); // ReferenceError - not accessible

console.log(constScoped); // ReferenceError - not accessible

}

// Redeclaration

var name = 'John';

var name = 'Jane'; // OK

let age = 25;

let age = 30; // SyntaxError: Identifier 'age' has already been declared

const city = 'NYC';

const city = 'LA'; // SyntaxError: Identifier 'city' has already been declared

// Reassignment

var a = 1;

a = 2; // OK

let b = 1;

b = 2; // OK

const c = 1;

c = 2; // TypeError: Assignment to constant variable

// Objects and arrays with const

const user = { name: 'John' };

user.name = 'Jane'; // OK - modifying property

user.age = 30; // OK - adding property

const numbers = [1, 2, 3];

numbers.push(4); // OK - modifying array

numbers[0] = 0; // OK - modifying element

// user = {}; // TypeError - reassigning

// numbers = []; // TypeError - reassigning

// Loop behavior difference

console.log('var in loops:');

for (var i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 100); // Prints: 3, 3, 3

}

console.log('let in loops:');

for (let i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 200); // Prints: 0, 1, 2

}

// Temporal Dead Zone example

function temporalDeadZoneExample() {

console.log(typeof undeclaredVariable); // "undefined"

console.log(typeof letVariable); // ReferenceError

let letVariable = 'Hello';

}

**Q10: What are closures and provide an example.**

**Answer:**

A closure is a function that has access to variables from its outer (enclosing) scope even after the outer function has finished executing. Closures are created when a function is defined inside another function and the inner function references variables from the outer function.

**How Closures Work:**

1. Inner function has access to outer function's variables
2. Outer function's variables remain accessible even after outer function returns
3. Each closure maintains its own copy of the outer variables

**Examples:**

// Basic Closure Example

function outerFunction(x) {

// This is the outer function's scope

function innerFunction(y) {

// Inner function has access to x and y

return x + y;

}

return innerFunction;

}

const addFive = outerFunction(5);

console.log(addFive(3)); // 8 - x (5) is still accessible

// Practical Example: Counter

function createCounter() {

let count = 0; // Private variable

return {

increment: () => ++count,

decrement: () => --count,

getCount: () => count

};

}

const counter1 = createCounter();

const counter2 = createCounter();

console.log(counter1.increment()); // 1

console.log(counter1.increment()); // 2

console.log(counter2.increment()); // 1 - separate closure

console.log(counter1.getCount()); // 2

// Module Pattern using Closures

const calculatorModule = (function() {

let result = 0; // Private variable

function add(x) { result += x; }

function multiply(x) { result \*= x; }

function getResult() { return result; }

function reset() { result = 0; }

// Public interface

return {

add,

multiply,

getResult,

reset

};

})();

calculatorModule.add(5);

calculatorModule.multiply(2);

console.log(calculatorModule.getResult()); // 10

// Function Factory with Closures

function multiplierFactory(multiplier) {

return function(x) {

return x \* multiplier;

};

}

const double = multiplierFactory(2);

const triple = multiplierFactory(3);

console.log(double(5)); // 10

console.log(triple(5)); // 15

// Event Handlers with Closures

function setupButtons() {

const buttons = document.querySelectorAll('.btn');

for (let i = 0; i < buttons.length; i++) {

// Closure captures the current value of i

buttons[i].addEventListener('click', function() {

console.log(`Button ${i} clicked`);

});

}

}

// Closure in setTimeout

function delayedGreeting(name) {

const message = `Hello, ${name}!`;

setTimeout(function() {

// Closure has access to 'message' and 'name'

console.log(message);

}, 1000);

}

delayedGreeting('John'); // Prints "Hello, John!" after 1 second

// Advanced: Currying with Closures

function curry(fn) {

return function curried(...args) {

if (args.length >= fn.length) {

return fn.apply(this, args);

} else {

return function(...nextArgs) {

return curried.apply(this, args.concat(nextArgs));

};

}

};

}

function add(a, b, c) {

return a + b + c;

}

const curriedAdd = curry(