**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(add);

console.log(curriedAdd(1)(2)(3)); // 6

console.log(curriedAdd(1, 2)(3)); // 6

console.log(curriedAdd(1)(2, 3)); // 6

// Memory Management with Closures

function createLargeDataHandler() {

const largeData = new Array(1000000).fill('data');

return function processData() {

// This closure keeps largeData in memory

return largeData.length;

};

}

// Be careful - largeData won't be garbage collected

const handler = createLargeDataHandler();

**Common Pitfalls:**

// Problem: Loop with var

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

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

}

// Solution 1: Use let

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

setTimeout(() => console.log(i), 100); // Prints: 0, 1, 2

}

// Solution 2: IIFE (Immediately Invoked Function Expression)

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

(function(index) {

setTimeout(() => console.log(index), 100); // Prints: 0, 1, 2

})(i);

}

**Medium Level Questions**

**Q11: Explain prototypal inheritance in JavaScript.**

**Answer:**

JavaScript uses prototypal inheritance where objects can inherit directly from other objects. Every object has a hidden [[Prototype]] property that points to another object (its prototype).

**How Prototypal Inheritance Works:**

// Creating objects with prototypal inheritance

// Method 1: Object.create()

const animalPrototype = {

makeSound() {

console.log(`${this.name} makes a sound`);

},

eat() {

console.log(`${this.name} is eating`);

}

};

const dog = Object.create(animalPrototype);

dog.name = 'Buddy';

dog.breed = 'Golden Retriever';

dog.bark = function() {

console.log(`${this.name} barks`);

};

dog.makeSound(); // "Buddy makes a sound"

dog.bark(); // "Buddy barks"

// Method 2: Constructor Functions

function Animal(name, species) {

this.name = name;

this.species = species;

}

Animal.prototype.makeSound = function() {

console.log(`${this.name} makes a sound`);

};

Animal.prototype.getInfo = function() {

return `${this.name} is a ${this.species}`;

};

function Dog(name, breed) {

// Call parent constructor

Animal.call(this, name, 'Canine');

this.breed = breed;

}

// Set up inheritance

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog;

// Add methods specific to Dog

Dog.prototype.bark = function() {

console.log(`${this.name} barks`);

};

Dog.prototype.fetch = function() {

console.log(`${this.name} fetches the ball`);

};

const myDog = new Dog('Max', 'Labrador');

myDog.makeSound(); // "Max makes a sound" (inherited)

myDog.bark(); // "Max barks" (own method)

console.log(myDog.getInfo()); // "Max is a Canine" (inherited)

// Method 3: ES6 Classes (syntactic sugar over prototypal inheritance)

class Vehicle {

constructor(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

}

start() {

console.log(`${this.make} ${this.model} is starting`);

}

getAge() {

return new Date().getFullYear() - this.year;

}

}

class Car extends Vehicle {

constructor(make, model, year, doors) {

super(make, model, year); // Call parent constructor

this.doors = doors;

}

honk() {

console.log(`${this.make} ${this.model} honks`);

}

// Override parent method

start() {

console.log('Turning key...');

super.start(); // Call parent method

console.log('Engine running');

}

}

const myCar = new Car('Toyota', 'Camry', 2020, 4);

myCar.start(); // Uses overridden method

myCar.honk(); // "Toyota Camry honks"

console.log(myCar.getAge()); // Inherited method

// Prototype Chain Exploration

function Person(name) {

this.name = name;

}

Person.prototype.greet = function() {

return `Hello, I'm ${this.name}`;

};

const john = new Person('John');

console.log(john.greet()); // "Hello, I'm John"

// Checking the prototype chain

console.log(john.hasOwnProperty('name')); // true

console.log(john.hasOwnProperty('greet')); // false

console.log(john.\_\_proto\_\_ === Person.prototype); // true

console.log(Person.prototype.\_\_proto\_\_ === Object.prototype); // true

console.log(Object.prototype.\_\_proto\_\_); // null

// Dynamic prototype modification

Person.prototype.sayGoodbye = function() {

return `Goodbye from ${this.name}`;

};

console.log(john.sayGoodbye()); // "Goodbye from John"

// Mixin Pattern (Multiple Inheritance Simulation)

const CanFly = {

fly() {

console.log(`${this.name} is flying`);

}

};

const CanSwim = {

swim() {

console.log(`${this.name} is swimming`);

}

};

function Duck(name) {

this.name = name;

}

// Combine multiple behaviors

Object.assign(Duck.prototype, CanFly, CanSwim);

const donald = new Duck('Donald');

donald.fly(); // "Donald is flying"

donald.swim(); // "Donald is swimming"

**Advanced Prototype Patterns:**

// Factory Pattern with Prototypes

function createPerson(name, age) {

const person = Object.create({

greet() {

return `Hi, I'm ${this.name}, ${this.age} years old`;

},

haveBirthday() {

this.age++;

return `Happy birthday! Now I'm ${this.age}`;

}

});

person.name = name;

person.age = age;

return person;

}

const alice = createPerson('Alice', 25);

console.log(alice.greet()); // "Hi, I'm Alice, 25 years old"

console.log(alice.haveBirthday()); // "Happy birthday! Now I'm 26"

// Prototype-based Composition

const Eventful = {

on(event, callback) {

this.events = this.events || {};

this.events[event] = this.events[event] || [];

this.events[event].push(callback);

},

emit(event, data) {

this.events = this.events || {};

if (this.events[event]) {

this.events[event].forEach(callback => callback(data));

}

}

};

function Button(label) {

this.label = label;

}

Object.assign(Button.prototype, Eventful);

Button.prototype.click = function() {

this.emit('click', { button: this.label });

};

const button = new Button('Submit');

button.on('click', (data) => console.log(`Button ${data.button} clicked!`));

button.click(); // "Button Submit clicked!"

**Q12: What is the event loop and how does it work?**

**Answer:**

The Event Loop is the mechanism that handles asynchronous operations in JavaScript. Since JavaScript is single-threaded, the event loop allows it to perform non-blocking operations by offloading operations to the browser or Node.js APIs and handling their completion.

**Event Loop Components:**

1. **Call Stack**: Where function executions are tracked
2. **Web APIs/Node APIs**: Handle async operations (setTimeout, HTTP requests, etc.)
3. **Callback Queue**: Holds completed async operation callbacks
4. **Microtask Queue**: Holds Promise callbacks and other microtasks
5. **Event Loop**: Manages the execution order

**How it Works:**

// Event Loop Example

console.log('1'); // Synchronous - goes to call stack immediately

setTimeout(() => {

console.log('2'); // Asynchronous - goes to Web API, then callback queue

}, 0);

Promise.resolve().then(() => {

console.log('3'); // Microtask - goes to microtask queue

});

console.log('4'); // Synchronous - goes to call stack immediately

// Output: 1, 4, 3, 2

// Detailed breakdown:

// 1. console.log('1') - call stack, executes immediately

// 2. setTimeout - goes to Web API, timer starts

// 3. Promise.resolve().then() - creates microtask

// 4. console.log('4') - call stack, executes immediately

// 5. Call stack empty, event loop checks microtask queue first

// 6. console.log('3') executes (microtask)

// 7. Microtask queue empty, event loop checks callback queue

// 8. console.log('2') executes (callback)

**Practical Examples:**

// Example 1: Understanding execution order

function example1() {

console.log('Start');

setTimeout(() => console.log('Timeout 1'), 0);

setTimeout(() => console.log('Timeout 2'), 0);

Promise.resolve()

.then(() => console.log('Promise 1'))

.then(() => console.log('Promise 2'));

console.log('End');

}

example1();

// Output: Start, End, Promise 1, Promise 2, Timeout 1, Timeout 2

// Example 2: Nested setTimeout vs Promise

function example2() {

setTimeout(() => {

console.log('Timeout 1');

Promise.resolve().then(() => console.log('Promise in timeout'));

}, 0);

Promise.resolve().then(() => {

console.log('Promise 1');

setTimeout(() => console.log('Timeout in promise'), 0);

});

}

example2();

// Output: Promise 1, Timeout 1, Promise in timeout, Timeout in promise

// Example 3: Blocking the event loop

function blockingExample() {

console.log('Before blocking');

setTimeout(() => console.log('This will be delayed'), 0);

// Blocking operation

const start = Date.now();

while (Date.now() - start < 2000) {

// Block for 2 seconds

}

console.log('After blocking');

}

blockingExample();

// Output: Before blocking, After blocking (2 sec delay), This will be delayed

// Example 4: Event Loop with fetch

async function fetchExample() {

console.log('1');

fetch('https://jsonplaceholder.typicode.com/posts/1')

.then(response => response.json())

.then(data => console.log('Fetch result'));

setTimeout(() => console.log('2'), 0);

await Promise.resolve();

console.log('3');

Promise.resolve().then(() => console.log('4'));

}

fetchExample();

// Output: 1, 3, 4, 2, Fetch result (network dependent)

**Advanced Event Loop Concepts:**

// Microtask vs Macrotask priority

function priorityExample() {

console.log('=== Start ===');

// Macrotasks (callback queue)

setTimeout(() => console.log('setTimeout 1'), 0);

setImmediate(() => console.log('setImmediate')); // Node.js only

// Microtasks (microtask queue)

Promise.resolve().then(() => {

console.log('Promise 1');

return Promise.resolve();

}).then(() => console.log('Promise 2'));

queueMicrotask(() => console.log('queueMicrotask'));

process.nextTick(() => console.log('nextTick')); // Node.js only - highest priority

console.log('=== End ===');

}

// Event Loop visualization function

function visualizeEventLoop() {

const steps = [];

const log = (message, type = 'sync') => {

steps.push({ message, type, time: Date.now() });

console.log(`[${type.toUpperCase()}] ${message}`);

};

log('Step 1: Synchronous');

setTimeout(() => log('Step 4: setTimeout (macrotask)', 'async'), 0);

Promise.resolve().then(() => log('Step 3: Promise (microtask)', 'async'));

log('Step 2: Synchronous');

// After all synchronous code

setTimeout(() => {

console.log('\n=== Execution Order ===');

steps.forEach((step, index) => {

console.log(`${index + 1}. ${step.message}`);

});

}, 10);

}

visualizeEventLoop();

// Practical application: Task scheduler

class TaskScheduler {

constructor() {

this.tasks = [];

this.running = false;

}

addTask(task, priority = 'normal') {

this.tasks.push({ task, priority });

if (!this.running) {

this.processTasks();

}

}

processTasks() {

this.running = true;

const processNext = () => {

if (this.tasks.length === 0) {

this.running = false;

return;

}

// Sort by priority

this.tasks.sort((a, b) => {

const priorities = { high: 3, normal: 2, low: 1 };

return priorities[b.priority] - priorities[a.priority];

});

const { task } = this.tasks.shift();

// Use microtask for high priority, macrotask for others

if (task.priority === 'high') {

Promise.resolve().then(() => {

task();

processNext();

});

} else {

setTimeout(() => {

task();

processNext();

}, 0);

}

};

processNext();

}

}

const scheduler = new TaskScheduler();

scheduler.addTask(() => console.log('Normal task 1'), 'normal');

scheduler.addTask(() => console.log('High priority task'), 'high');

scheduler.addTask(() => console.log('Low priority task'), 'low');

scheduler.addTask(() => console.log('Normal task 2'), 'normal');

// Performance monitoring

function measureEventLoopLag() {

const start = process.hrtime.bigint();

setImmediate(() => {

const lag = Number(process.hrtime.bigint() - start) / 1000000; // Convert to ms

console.log(`Event loop lag: ${lag.toFixed(2)}ms`);

});

}

setInterval(measureEventLoopLag, 1000);

**TypeScript**

**Basic Level Questions**

**Q13: What is TypeScript and what are its benefits?**

**Answer:**

TypeScript is a statically typed superset of JavaScript developed by Microsoft. It adds optional type annotations and compiles to plain JavaScript.

**Key Benefits:**

1. **Type Safety**: Catch errors at compile time
2. **Better IDE Support**: Enhanced autocomplete, refactoring, navigation
3. **Self-Documenting Code**: Types serve as documentation
4. **Improved Maintainability**: Easier to refactor and understand large codebases
5. **Modern JavaScript Features**: Use latest JS features with backwards compatibility
6. **Gradual Adoption**: Can be introduced incrementally

**Basic TypeScript Examples:**

// Basic Types

let name: string = 'John';

let age: number = 30;

let isActive: boolean = true;

let hobbies: string[] = ['reading', 'swimming'];

let tuple: [string, number] = ['John', 30];

// Any type (avoid when possible)

let dynamicValue: any = 'hello';

dynamicValue = 42;

dynamicValue = true;

// Union types

let id: string | number;

id = 'user123';

id = 123;

// Object types

let user: {

name: string;

age: number;

email?: string; // Optional property

} = {

name: 'John',

age: 30

};

// Function types

function greet(name: string): string {

return `Hello, ${name}!`;

}

const add = (a: number, b: number): number => a + b;

// Function with optional parameters

function createUser(name: string, age: number, email?: string): object {

return { name, age, ...(email && { email }) };

}

// Function with default parameters

function multiply(a: number, b: number = 1): number {

return a \* b;

}

// Rest parameters

function sum(...numbers: number[]): number {

return numbers.reduce((total, num) => total + num, 0);

}

// Interface definitions

interface User {

readonly id: number; // Readonly property

name: string;

email: string;

age?: number; // Optional property

isActive: boolean;

}

interface UserMethods {

updateEmail(newEmail: string): void;

deactivate(): void;

}

// Extending interfaces

interface AdminUser extends User {

role: 'admin' | 'super-admin';

permissions: string[];

}

// Implementing interfaces

class UserService implements UserMethods {

private users: User[] = [];

updateEmail(newEmail: string): void {

// Implementation

}

deactivate(): void {

// Implementation

}

addUser(user: User): void {

this.users.push(user);

}

getUser(id: number): User | undefined {

return this.users.find(user => user.id === id);

}

}

// Type aliases

type Status = 'pending' | 'approved' | 'rejected';

type UserRole = 'user' | 'admin' | 'moderator';

interface Task {

id: string;

title: string;

status: Status;

assignedTo: User;

}

// Generic types

function identity<T>(arg: T): T {

return arg;

}

const stringResult = identity<string>('hello'); // Type: string

const numberResult = identity(42); // Type inferred as number

// Generic interfaces

interface Repository<T> {

findById(id: string): T | null;

save(entity: T): T;

delete(id: string): boolean;

}

class UserRepository implements Repository<User> {

private users: User[] = [];

findById(id: string): User | null {

return this.users.find(user => user.id.toString() === id) || null;

}

save(user: User): User {

this.users.push(user);

return user;

}

delete(id: string): boolean {

const index = this.users.findIndex(user => user.id.toString() === id);

if (index > -1) {

this.users.splice(index, 1);

return true;

}

return false;

}

}

// Enum types

enum Color {

Red = 'red',

Green = 'green',

Blue = 'blue'

}

enum HttpStatus {

OK = 200,

NotFound = 404,

InternalServerError = 500

}

// Using enums

function setThemeColor(color: Color): void {

document.body.style.backgroundColor = color;

}

setThemeColor(Color.Blue);

// Type assertions

let someValue: unknown = 'this is a string';

let strLength: number = (someValue as string).length;

// Alternative syntax: let strLength: number = (<string>someValue).length;

// Literal types

type ButtonVariant = 'primary' | 'secondary' | 'danger';

type Size = 'small' | 'medium' | 'large';

interface Button {

variant: ButtonVariant;

size: Size;

disabled?: boolean;

}

function createButton(config: Button): HTMLButtonElement {

const button = document.createElement('button');

button.className = `btn btn-${config.variant} btn-${config.size}`;

button.disabled = config.disabled || false;

return button;

}

**Q14: Explain interfaces vs types in TypeScript.**

**Answer:**

Both interfaces and types can define object shapes, but they have different capabilities and use cases.

**Interfaces:**

* Primarily for object shapes
* Can be extended and merged
* Support declaration merging
* Better for OOP patterns
* Can be implemented by classes

**Types:**

* More flexible, can define any type
* Support union, intersection, and complex types
* Cannot be merged
* Better for functional programming patterns
* More concise for complex type manipulations

**Detailed Comparison:**

// INTERFACES

// Basic interface

interface User {

id: number;

name: string;

email: string;

}

// Interface extension

interface AdminUser extends User {

role: string;

permissions: string[];

}

// Multiple inheritance

interface Timestamped {

createdAt: Date;

updatedAt: Date;

}

interface AuditableUser extends User, Timestamped {

lastLoginAt?: Date;

}

// Interface merging (declaration merging)

interface Window {

customProperty: string;

}

interface Window {

anotherProperty: number;

}

// Now Window has both customProperty and anotherProperty

// Implementing interfaces in classes

interface Flyable {

fly(): void;

altitude: number;

}

interface Swimmable {

swim(): void;

depth: number;

}

class Duck implements Flyable, Swimmable {

altitude: number = 0;

depth: number = 0;

fly(): void {

this.altitude = 100;

console.log('Duck is flying');

}

swim(): void {

this.depth = 5;

console.log('Duck is swimming');

}

}

// TYPES

// Basic type

type UserType = {

id: number;

name: string;

email: string;

};

// Type intersection (similar to extends)

type AdminUserType = UserType & {

role: string;

permissions: string[];

};

// Union types

type Status = 'loading' | 'success' | 'error';

type ID = string | number;

// Complex type manipulations

type EventHandler<T> = (event: T) => void;

type AsyncEventHandler<T> = (event: T) => Promise<void>;

// Conditional types

type NonNullable<T> = T extends null | undefined ? never : T;

// Mapped types

type Partial<T> = {

[P in keyof T]?: T[P];

};

type Required<T> = {

[P in keyof T]-?: T[P];

};

// Template literal types

type CSSProperty = 'margin' | 'padding' | 'border';

type CSSDirection = 'top' | 'right' | 'bottom' | 'left';

type CSSPropertyWithDirection = `${CSSProperty}-${CSSDirection}`;

// Results in: 'margin-top' | 'margin-right' | ... etc.

// Function types

type Calculator = (a: number, b: number) => number;

type AsyncDataFetcher<T> = (url: string) => Promise<T>;

// Tuple types

type Coordinates = [number, number];

type NamedCoordinates = [x: number, y: number];

// When to use Interface vs Type

// Use Interface when:

// 1. Defining object shapes

interface ApiResponse {

data: unknown;

status: number;

message: string;

}

// 2. You might need to extend or merge

interface BaseComponent {

render(): void;

}

interface BaseComponent {

destroy(): void; // Declaration merging

}

// 3. Implementing in classes

interface Validator {

validate(value: unknown): boolean;

}

class EmailValidator implements Validator {

validate(value: unknown): boolean {

return typeof value === 'string' && value.includes('@');

}

}

// Use Type when:

// 1. Creating union types

type Theme = 'light' | 'dark' | 'auto';

// 2. Complex type manipulations

type DeepPartial<T> = {

[P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];

};

// 3. Function types

type Middleware = (req: Request, res: Response, next: NextFunction) => void;

// 4. Conditional logic

type ApiResult<T> = T extends string

? { message: T }

: T extends number

? { code: T }

: { data: T };

// Practical examples combining both

// Interface for base structure

interface BaseEntity {

id: string;

createdAt: Date;

updatedAt: Date;

}

// Type for operations

type EntityOperations<T> = {

create: (data: Omit<T, keyof BaseEntity>) => Promise<T>;

update: (id: string, data: Partial<T>) => Promise<T>;

delete: (id: string) => Promise<boolean>;

findById: (id: string) => Promise<T | null>;

};

// Combining interface and type

interface Product extends BaseEntity {

name: string;

price: number;

category: string;

}

type ProductService = EntityOperations<Product> & {

findByCategory: (category: string) => Promise<Product[]>;

updatePrice: (id: string, newPrice: number) => Promise<Product>;

};

// Advanced pattern: Discriminated unions with interfaces

interface LoadingState {

status: 'loading';

}

interface SuccessState {

status: 'success';

data: unknown;

}

interface ErrorState {

status: 'error';

error: string;

}

type AsyncState = LoadingState | SuccessState | ErrorState;

function handleState(state: AsyncState) {

switch (state.status) {

case 'loading':

return 'Loading...';

case 'success':

return `Data: ${JSON.stringify(state.data)}`;

case 'error':

return `Error: ${state.error}`;

default:

// TypeScript ensures exhaustive checking

const exhaustiveCheck: never = state;

return exhaustiveCheck;

}

}

**Backend Technologies**

**Node.js**

**Basic Level Questions**

**Q15: What is Node.js and how does it work?**

**Answer:**

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that allows you to run JavaScript on the server side. It uses an event-driven, non-blocking I/O model that makes it efficient for building scalable network applications.

**Key Characteristics:**

1. **Single-threaded Event Loop**: Handles multiple concurrent operations
2. **Non-blocking I/O**: Operations don't wait for completion
3. **Built-in Modules**: File system, HTTP, crypto, etc.
4. **NPM Ecosystem**: Largest package repository
5. **Cross-platform**: Runs on Windows, macOS, Linux

**How Node.js Works:**

// Basic Node.js server

const http = require('http');

const url = require('url');

const fs = require('fs').promises;

const server = http.createServer(async (req, res) => {

const parsedUrl = url.parse(req.url, true);

const path = parsedUrl.pathname;

const method = req.method;

// Set CORS headers

res.setHeader('Access-Control-Allow-Origin', '\*');

res.setHeader('Content-Type', 'application/json');

try {

if (method === 'GET' && path === '/api/users') {

// Simulate async database operation

const users = await getUsersFromDatabase();

res.statusCode = 200;

res.end(JSON.stringify(users));

} else if (method === 'POST' && path === '/api/users') {

// Handle POST request

let body = '';

req.on('data', chunk => {

body += chunk.toString();

});

req.on('end', async () => {

try {

const userData = JSON.parse(body);

const newUser = await createUser(userData);

res.statusCode = 201;

res.end(JSON.stringify(newUser));

} catch (error) {

res.statusCode = 400;

res.end(JSON.stringify({ error: 'Invalid JSON' }));

}

});

} else if (method === 'GET' && path === '/api/file') {

// File operations (non-blocking)

try {

const fileContent = await fs.readFile('./data.txt', 'utf8');

res.statusCode = 200;

res.end(JSON.stringify({ content: fileContent }));

} catch (error) {

res.statusCode = 404;

res.end(JSON.stringify({ error: 'File not found' }));

}

} else {

res.statusCode = 404;

res.end(JSON.stringify({ error: 'Route not found' }));

}

} catch (error) {

res.statusCode = 500;

res.end(JSON.stringify({ error: 'Internal server error' }));

}

});

// Simulate database operations

async function getUsersFromDatabase() {

// Simulate async operation

return new Promise((resolve) => {

setTimeout(() => {

resolve([

{ id: 1, name: 'John Doe', email: 'john@example.com' },

{ id: 2, name: 'Jane Smith', email: 'jane@example.com' }

]);

}, 100);

});

}

async function createUser(userData) {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (!userData.name || !userData.email) {

reject(new Error('Name and email are required'));

} else {

resolve({

id: Date.now(),

...userData,

createdAt: new Date().toISOString()

});

}

}, 50);

});

}

const PORT = process.env.PORT || 3000;

server.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

// Event-driven architecture examples

const EventEmitter = require('events');

class UserService extends EventEmitter {

constructor() {

super();

this.users = [];

}

async createUser(userData) {

try {

this.emit('user:creating', userData);

// Simulate validation

if (!userData.email) {

throw new Error('Email is required');

}

const user = {

id: Date.now(),

...userData,

createdAt: new Date()

};

this.users.push(user);

this.emit('user:created', user);

return user;

} catch (error) {