□ Frontend Mock Interview Script & Answers

□ Interview Structure

Duration: 60-90 minutes

Format: Technical + Behavioral

Focus Areas: React/Next.js, Tailwind, State Management, Testing, AWS, Performance

---

□ Section 1: React.js / Next.js Core Questions

□ Conceptual Questions

Q1: What is the difference between `useEffect`, `useLayoutEffect`, and `useInsertionEffect`?

Answer:

- **useEffect**: Runs after DOM updates, asynchronous, doesn't block browser painting
- **useLayoutEffect**: Runs synchronously after DOM mutations but before browser repaints, blocks visual updates
- **useInsertionEffect**: Runs before DOM mutations, used for CSS-in-JS libraries to inject styles

Use cases:

- useEffect: API calls, subscriptions, side effects
- useLayoutEffect: DOM measurements, preventing layout shifts
- useInsertionEffect: Style injection, CSS-in-JS

Q2: How does Next.js handle server-side rendering vs static generation?

Answer:

- **SSR (getServerSideProps)**: Renders on each request, dynamic content, SEO-friendly
- **SSG (getStaticProps)**: Pre-renders at build time, faster, cached, good for static content
- **ISR**: Combines both - static generation with revalidation periods

When to use:

- SSR: E-commerce, user-specific content
- SSG: Blog posts, documentation, marketing pages

Q3: Explain hydration in Next.js and why it's important.

Answer: Hydration is the process where React attaches event listeners to server-rendered HTML, making it interactive.

Process:

1. Server renders HTML
2. Client downloads JavaScript
3. React "hydrates" the HTML with interactivity

Why important:

- Enables interactivity without full re-render
- Improves perceived performance
- Maintains SEO benefits of SSR

Q4: What are React Suspense and Concurrent features?

Answer: Suspense: Component that shows fallback while content is loading **Concurrent Features**: React 18's ability to interrupt and prioritize updates

Key features:

- Suspense for data fetching
- Concurrent rendering
- Automatic batching
- Transitions (useTransition)

Q5: How do you handle large component trees to avoid performance issues?

Answer:

- **React.memo**: Prevent unnecessary re-renders
- **useMemo/useCallback**: Memoize expensive calculations/functions
- **Code splitting**: Lazy load components
- **Virtualization**: For long lists (react-window)
- **Profiler**: Identify bottlenecks

Q6: Compare SSR vs CSR vs ISR in Next.js with use cases.

Answer:

- **SSR**: Server renders on each request (e-commerce, dashboards)
- **CSR**: Client renders everything (SPAs, admin panels)
- **ISR**: Static generation with revalidation (blogs, product catalogs)

□ Practical Questions

Q7: How would you implement infinite scrolling in React?

Answer:

```
const useInfiniteScroll = (callback) => {
  const observer = useRef();

  const lastElementRef = useCallback(node => {
    if (observer.current) observer.current.disconnect();
    observer.current = new IntersectionObserver(entries => {
      if (entries[0].isIntersecting) {
        callback();
      }
    });
    if (node) observer.current.observe(node);
  }, [callback]);

  return lastElementRef;
};
```

Q8: How do you dynamically load components in Next.js?

Answer:

```
// Using dynamic imports
const DynamicComponent = dynamic(() => import('./Component'), {
  loading: () => <Spinner />,
  ssr: false // if component uses browser APIs
});

// With preloading
const preloadComponent = () => {
  import('./Component');
};
```

Q9: How do you persist global state between page reloads?

Answer:

- **localStorage/sessionStorage**: Simple key-value storage
- **Redux Persist**: For Redux state

- **Zustand persist**: Built-in persistence
- **Next.js getServerSideProps**: Server-side state hydration

Q10: What are React Server Components (RSCs) and how are they different?

Answer: RSCs are components that run on the server, reducing client bundle size.

Key differences:

- No useState, useEffect, event handlers
- Can directly access databases, file systems
- Smaller client bundles
- Better SEO and performance

□ Section 2: Tailwind CSS & Frontend Architecture

□ Conceptual Questions

Q11: How does utility-first CSS differ from traditional CSS?

Answer: Traditional CSS:

- Custom class names
- Separate CSS files
- Specificity issues
- Larger bundle sizes

Utility-first:

- Predefined utility classes
- Inline styling approach
- Consistent design system
- Smaller production bundles (PurgeCSS)

Q12: What are Tailwind's performance implications in large apps?

Answer: Pros:

- PurgeCSS removes unused styles
- Smaller production CSS
- No CSS-in-JS runtime overhead

Cons:

- Larger development CSS
- HTML becomes verbose
- Learning curve for team

Q13: How do you build a design system using Tailwind?

Answer:

- **Base components**: Buttons, inputs, cards
- **Custom utilities**: @apply directive
- **Theme configuration**: tailwind.config.js
- **Component library**: Storybook integration
- **Design tokens**: Colors, spacing, typography

Q14: Explain Atomic Design and how it can be applied in React.

Answer: Atomic Design levels:

1. **Atoms**: Basic building blocks (buttons, inputs)
2. **Molecules**: Simple combinations (search bar)
3. **Organisms**: Complex UI sections (header, footer)
4. **Templates**: Page layouts
5. **Pages**: Specific instances

React implementation:

- Component hierarchy
- Props for customization
- Composition over inheritance

□ Practical Questions

Q15: How would you structure components to maximize reusability with Tailwind?

Answer:

```
// Base component with variants
const Button = ({ variant = 'primary', size = 'md', children, ...props }) => {
  const baseClasses = 'font-medium rounded-lg transition-colors';
  const variants = {
    primary: 'bg-blue-600 hover:bg-blue-700 text-white',
    secondary: 'bg-gray-200 hover:bg-gray-300 text-gray-900'
  };
  const sizes = {
    sm: 'px-3 py-1.5 text-sm',
    md: 'px-4 py-2 text-base',
    lg: 'px-6 py-3 text-lg'
  };

  return (
    <button
      className={`${baseClasses} ${variants[variant]} ${sizes[size]}`}
      {...props}
    >
      {children}
    </button>
  );
};
```

Q16: How do you handle theme switching (light/dark) with Tailwind + Next.js?

Answer:

```
// tailwind.config.js
module.exports = {
  darkMode: 'class',
  // ... rest of config
};

// Theme context
const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  useEffect(() => {
    document.documentElement.classList.toggle('dark', theme === 'dark');
  }, [theme]);

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

□ Section 3: State Management (Redux, Context API, Zustand)

□ Conceptual Questions

Q17: Compare Redux Toolkit, Zustand, and Context API for managing state.

Answer: Redux Toolkit:

- Pros: Predictable, dev tools, middleware ecosystem
- Cons: Boilerplate, learning curve, overkill for simple apps

Zustand:

- Pros: Simple API, small bundle, TypeScript friendly
- Cons: Less ecosystem, fewer dev tools

Context API:

- Pros: Built-in, simple for small apps
- Cons: Performance issues with frequent updates, prop drilling

Q18: What is the role of middleware in Redux?

Answer: Middleware sits between dispatching an action and the reducer, allowing side effects.

Common middleware:

- **redux-thunk**: Async actions
- **redux-saga**: Complex async flows
- **redux-logger**: Debugging
- **redux-persist**: State persistence

Q19: How do you handle derived state in Zustand?

Answer:

```
const useStore = create((set, get) => ({
  items: [],
  getCompletedItems: () => get().items.filter(item => item.completed),
  getTotalPrice: () => get().items.reduce((sum, item) => sum + item.price, 0)
}));
```

□ Practical Questions

Q20: How would you implement a multi-step form using Zustand?

Answer:

```
const useFormStore = create((set, get) => ({
  currentStep: 0,
  formData: {},

  nextStep: () => set(state => ({
    currentStep: Math.min(state.currentStep + 1, 2)
  })),

  prevStep: () => set(state => ({
    currentStep: Math.max(state.currentStep - 1, 0)
  })),

  updateFormData: (data) => set(state => ({
    formData: { ...state.formData, ...data }
  }))
}));
```

Q21: How do you persist state between page reloads in a Next.js app?

Answer:

```
// With Zustand persist
import { persist } from 'zustand/middleware';

const useStore = create(
  persist(
    (set) => ({
      // store implementation
    }),
    {
      name: 'app-storage',
      getStorage: () => localStorage
    }
  )
);

// With Next.js getServerSideProps
export async function getServerSideProps(context) {
  const savedState = context.req.cookies['app-state'];
  return {
    props: {
      initialState: savedState ? JSON.parse(savedState) : {}
    }
  };
}
```

□ Section 4: Testing, CI/CD, and Production Readiness

□ Conceptual Questions

Q22: How would you structure your React codebase for testability?

Answer:

- **Separation of concerns**: Business logic separate from UI
- **Custom hooks**: Testable logic extraction
- **Component composition**: Smaller, focused components
- **Dependency injection**: Mock external dependencies
- **Test utilities**: Reusable test helpers

Q23: What is the difference between unit, integration, and E2E tests?

Answer: Unit tests:

- Test individual functions/components
- Fast, isolated, mock dependencies
- Tools: Jest, React Testing Library

Integration tests:

- Test component interactions
- Test API integrations
- Tools: React Testing Library, MSW

E2E tests:

- Test complete user workflows
- Real browser, real APIs
- Tools: Cypress, Playwright

Q24: How would you optimize Core Web Vitals in a Next.js application?

Answer: LCP (Largest Contentful Paint):

- Optimize images (next/image)
- Preload critical resources
- Use CDN for static assets

FID (First Input Delay):

- Code splitting
- Reduce JavaScript bundle size
- Optimize third-party scripts

CLS (Cumulative Layout Shift):

- Set image dimensions
- Reserve space for dynamic content
- Avoid inserting content above existing content

□ Practical Questions

Q25: Write a test using React Testing Library for a controlled form input.

Answer:

```
import { render, screen, fireEvent } from '@testing-library/react';
import { FormInput } from './FormInput';

test('updates value when user types', () => {
  const mockOnChange = jest.fn();

  render(
    <FormInput
      value=""
      onChange={mockOnChange}
      placeholder="Enter name"
    />
  );

  const input = screen.getByPlaceholderText('Enter name');
  fireEvent.change(input, { target: { value: 'John' } });

  expect(mockOnChange).toHaveBeenCalledWith('John');
});
```

Q26: What tools do you use in your CI/CD pipeline for frontend deployment?

Answer:

- **GitHub Actions**: CI/CD automation
- **ESLint/Prettier**: Code quality
- **Jest**: Unit testing
- **Cypress**: E2E testing
- **Lighthouse CI**: Performance testing
- **Vercel/Netlify**: Deployment platforms
- **AWS S3 + CloudFront**: Static hosting

Q27: How do you handle feature flagging in production?

Answer:

```
// Feature flag implementation
const useFeatureFlag = (flagName) => {
  const [isEnabled, setIsEnabled] = useState(false);

  useEffect(() => {
    // Fetch from API or environment variable
    const checkFlag = async () => {
      const response = await fetch('/api/feature-flags');
      const flags = await response.json();
      setIsEnabled(flags[flagName] || false);
    };
    checkFlag();
  }, [flagName]);

  return isEnabled;
};

// Usage
const MyComponent = () => {
  const isNewFeatureEnabled = useFeatureFlag('new-feature');

  return (
    <div>
      {isNewFeatureEnabled ? <NewFeature /> : <OldFeature />}
    </div>
  );
};
```

□ Section 5: DevOps, AWS, Monitoring

AWS Services

Q28: How would you integrate Cognito for authentication in a Next.js app?

Answer:

```
// Using AWS Amplify
import { Amplify, Auth } from 'aws-amplify';

Amplify.configure({
  Auth: {
    region: 'us-east-1',
    userPoolId: 'your-user-pool-id',
    userPoolWebClientId: 'your-client-id'
  }
});

// Sign in
const signIn = async (username, password) => {
  try {
    const user = await Auth.signIn(username, password);
    return user;
  } catch (error) {
    console.error('Error signing in:', error);
  }
};
```

Q29: What's the best way to store and access static assets via S3?

Answer:

- **S3 bucket**: Store assets with proper permissions
- **CloudFront**: CDN for global distribution
- **Next.js configuration**: Use custom domain
- **Image optimization**: Use S3 + CloudFront + next/image

Q30: How would you host a serverless frontend using AWS S3 + CloudFront?

Answer:

1. **Build the app**: `npm run build`
2. **Upload to S3**: Sync build folder to S3 bucket
3. **Configure CloudFront**: Point to S3 origin
4. **Set up routing**: Handle SPA routing with error pages
5. **Custom domain**: Route 53 + SSL certificate

Q31: How do you use DynamoDB for low-latency reads in a frontend-heavy app?

Answer:

- **API Gateway**: RESTful API endpoints

- **Lambda functions**: Serverless backend logic
- **DynamoDB**: NoSQL database for data storage
- **Caching**: CloudFront or ElastiCache for frequently accessed data

Monitoring & Observability

Q32: How do you track frontend errors in production?

Answer:

- **Sentry**: Error tracking and monitoring
- **LogRocket**: Session replay and error tracking
- **Google Analytics**: User behavior and errors
- **Custom error boundaries**: React error boundaries with logging

Q33: What's your approach for logging and performance monitoring?

Answer:

- **Application logs**: Console.log with structured logging
- **Performance monitoring**: Web Vitals, custom metrics
- **APM tools**: New Relic, DataDog
- **Real User Monitoring (RUM)**: Track actual user experience

Q34: How do you monitor and alert when Core Web Vitals degrade?

Answer:

- **Lighthouse CI**: Automated performance testing
- **Web Vitals API**: Real-time monitoring
- **Alerting**: Set up thresholds for LCP, FID, CLS
- **Dashboards**: Grafana or similar for visualization

□ Section 6: Bonus/Advanced: Visualization, Micro-frontends, SEO

Data Visualization

Q35: What are the pros/cons of Recharts vs D3.js?

Answer: Recharts:

- Pros: React-friendly, easy to use, good defaults
- Cons: Less customization, larger bundle size

D3.js:

- Pros: Highly customizable, powerful, industry standard
- Cons: Steep learning curve, more code required

Q36: How do you render real-time charts in React using WebSockets?

Answer:

```
const useRealTimeChart = () => {
  const [data, setData] = useState([]);

  useEffect(() => {
    const ws = new WebSocket('wss://api.example.com/chart-data');

    ws.onmessage = (event) => {
      const newData = JSON.parse(event.data);
      setData(prev => [...prev.slice(-50), newData]); // Keep last 50 points
    };

    return () => ws.close();
  }, []);

  return data;
};
```

Micro-Frontends

Q37: What is a micro frontend and when would you use it?

Answer: Micro frontend is an architectural pattern where a frontend app is composed of semi-independent applications.

When to use:

- Large teams working independently
- Different tech stacks for different parts
- Independent deployment cycles
- Legacy system integration

Q38: How do you handle shared state between micro frontends?

Answer:

- **Module Federation**: Webpack 5 feature for sharing code
- **Custom events**: Browser events for communication

- **Shared state management**: Redux or similar across apps
- **URL-based state**: Use URL parameters for shared state

SEO / Performance

Q39: How do you handle canonical tags, sitemap, and metadata in Next.js?

Answer:

```
// next.config.js
module.exports = {
  async headers() {
    return [
      {
        source: '/sitemap.xml',
        headers: [
          {
            key: 'Content-Type',
            value: 'application/xml',
          },
        ],
      },
    ];
  },
};

// pages/sitemap.xml.js
const Sitemap = () => {};

export const getServerSideProps = async ({ res }) => {
  const sitemap = `<?xml version="1.0" encoding="UTF-8"?>
    <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
      <url>
        <loc>https://example.com</loc>
        <lastmod>${new Date().toISOString()}</lastmod>
      </url>
    </urlset>`;

  res.setHeader('Content-Type', 'text/xml');
  res.write(sitemap);
  res.end();

  return {
    props: {},
  };
};

export default Sitemap;
```

Q40: How do you handle preloading fonts, lazy loading images, and optimizing LCP?

Answer:

```
// next.config.js
module.exports = {
  images: {
    domains: ['example.com'],
    formats: ['image/webp', 'image/avif'],
  },
};

// _document.js
import { Html, Head, Main, NextScript } from 'next/document';

export default function Document() {
  return (
    <Html>
      <Head>
        <link
          rel="preload"
          href="/fonts/inter-var.woff2"
          as="font"
          type="font/woff2"
          crossOrigin="anonymous"
        />
      </Head>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  );
}
```

□□ Behavioral + Team Fit Questions

Q41: How do you handle performance bugs reported from production?

Answer:

1. **Reproduce**: Try to replicate the issue
2. **Profile**: Use browser dev tools, Lighthouse
3. **Prioritize**: Assess impact and urgency
4. **Fix**: Implement solution with testing
5. **Monitor**: Track metrics after deployment

6. **Document**: Share learnings with team

Q42: Tell me about a time you had to rewrite a legacy component.

Answer: "I had to rewrite a complex form component that was using class components and jQuery. I broke it down into smaller functional components, used React hooks for state management, and implemented proper form validation. The new version was more maintainable, testable, and had better performance."

Q43: How do you approach working with designers or product teams?

Answer:

- **Early collaboration**: Involve designers in technical decisions
- **Clear communication**: Explain technical constraints
- **Prototyping**: Build quick prototypes for feedback
- **Design system**: Establish shared components and patterns
- **Regular sync**: Weekly meetings to align on priorities

Q44: Describe a time you introduced a performance optimization in production.

Answer: "I identified that our image loading was causing layout shifts. I implemented lazy loading with proper aspect ratios, used next/image for optimization, and added skeleton loaders. This improved our CLS score by 0.15 and overall page load time by 30%."

Q45: How do you stay updated with frontend technologies?

Answer:

- **Newsletters**: JavaScript Weekly, React Status
- **Blogs**: React blog, Next.js blog, CSS-Tricks
- **Conferences**: React Conf, JSConf
- **Open source**: Contributing to projects
- **Experimentation**: Building side projects with new tech

□ Mock Interview Tips

Before the Interview:

1. **Review your projects**: Be ready to discuss technical decisions
2. **Practice coding**: Whiteboard coding exercises
3. **Prepare questions**: Ask about team, tech stack, challenges
4. **Test your setup**: Ensure your development environment works

During the Interview:

1. **Think aloud**: Explain your thought process
2. **Ask clarifying questions**: Understand requirements fully
3. **Start simple**: Begin with basic solutions, then optimize
4. **Show enthusiasm**: Demonstrate passion for the work

After the Interview:

1. **Follow up**: Send thank you email
2. **Reflect**: Note what went well and what to improve
3. **Practice**: Work on areas that were challenging

□ Additional Resources

Key Concepts to Review:

- React 18 Concurrent Features
- Next.js 13+ App Router
- TypeScript fundamentals
- CSS Grid and Flexbox
- Web APIs (Intersection Observer, etc.)
- Performance optimization techniques

Practice Platforms:

- LeetCode (JavaScript problems)
- Frontend Mentor (UI challenges)
- CodeSandbox (React/Next.js demos)
- GitHub (Open source contributions)

Good luck with your interview! Remember to stay calm, think clearly, and show your problem-solving approach.