Next.js Comprehensive Notes & Interview Guide

What is Next.js?

Next.js is a **React-based full-stack framework** that provides powerful features like **Server-Side Rendering (SSR)**, **Static Site Generation (SSG)**, **API Routes**, and **File-based Routing** out of the box. It's built on top of React to create production-ready web applications with enhanced performance, SEO, and developer experience.

Key Features:

- **Zero Configuration**: Works out of the box with sensible defaults
- **File-based Routing**: Pages are created by adding files to the `pages` or `app` directory
- **API Routes**: Built-in API endpoints without separate backend
- **Automatic Code Splitting**: Bundles are split automatically for optimal loading
- **Built-in CSS Support**: Support for CSS Modules, Sass, and CSS-in-JS
- **Image Optimization**: Automatic image optimization with `next/image`
- **TypeScript Support**: First-class TypeScript support

1. Project Setup & Structure

Creating a Next.js Application:

```
# Create new Next.js app
npx create-next-app@latest my-app

# With TypeScript
npx create-next-app@latest my-app --typescript

# Navigate to project
cd my-app

# Start development server
npm run dev
```

Project Structure:

```
my-app/
├── app/                    # App Router (Next.js 13+)
│   ├── api/               # API routes
```

```
│   ├── globals.css        # Global styles
│   ├── layout.tsx         # Root layout
│   └── page.tsx          # Home page
├── public/               # Static assets
├── components/           # Reusable components
├── lib/                 # Utility functions
├── next.config.js       # Next.js configuration
└── package.json         # Dependencies
```

2. Routing System

File-based Routing

What it does: Next.js uses the file system for routing. Files in the `app` directory automatically become routes based on their folder structure.

Basic Routes:

```
// app/page.tsx - Homepage (/)
export default function Home() {
    return <h1>Welcome to Homepage</h1>;
}

// app/about/page.tsx - About page (/about)
export default function About() {
    return <h1>About Us</h1>;
}

// app/contact/page.tsx - Contact page (/contact)
export default function Contact() {
    return <h1>Contact Us</h1>;
}
```

Nested Routes:

```
// app/blog/page.tsx - Blog listing (/blog)
export default function Blog() {
    return <h1>Blog Posts</h1>;
}

// app/blog/[slug]/page.tsx - Individual blog post (/blog/my-first-post)
export default function BlogPost({ params }: { params: { slug: string } }) {
    return <h1>Post: {params.slug}</h1>;
}

// app/blog/category/[category]/page.tsx - Category pages (/blog/category/tech)
export default function BlogCategory({ params }: { params: { category: string } }) {
    return <h1>Category: {params.category}</h1>;
}
```

Dynamic Routes:

```
// app/users/[id]/page.tsx - Dynamic user pages (/users/123)
export default function UserProfile({ params }: { params: { id: string } }) {
    return (
        <div>
            <h1>User Profile</h1>
            <p>User ID: {params.id}</p>
        </div>
    );
}

// app/products/[...slug]/page.tsx - Catch-all routes (/products/electronics/phones/iphor
export default function ProductPage({ params }: { params: { slug: string[] } }) {
    return (
        <div>
            <h1>Product Page</h1>
            <p>Path segments: {params.slug.join(' / ')}</p>
        </div>
    );
}
```

3. API Routes

What it does: API Routes allow you to build API endpoints as part of your Next.js application. They run on the server and can handle database operations, authentication, and external API calls.

Basic API Routes

Simple API Route:

```
// app/api/route.ts - Handles requests to /api
import { NextResponse } from 'next/server';

export async function GET() {
    return NextResponse.json({
        message: "Hello from API",
        timestamp: new Date().toISOString()
    });
}

export async function POST(request: Request) {
    const body = await request.json();
    return NextResponse.json({
        message: "Data received",
        data: body
    });
}
```

Nested API Routes:

```
// app/api/users/route.ts - Handles /api/users
import { NextResponse } from 'next/server';

// Sample users data (in real app, this would come from database)
const users = [
    { id: 1, name: "John Doe", email: "john@example.com" },
    { id: 2, name: "Jane Smith", email: "jane@example.com" },
    { id: 3, name: "Bob Johnson", email: "bob@example.com" }
];

export async function GET() {
    return NextResponse.json({
        success: true,
        users: users,
        total: users.length
    });
}

export async function POST(request: Request) {
    try {
        const { name, email } = await request.json();

        // Validate required fields
        if (!name || !email) {
            return NextResponse.json(
                { error: "Name and email are required" },
                { status: 400 }
            );
        }

        // Create new user (in real app, save to database)
        const newUser = {
            id: users.length + 1,
            name,
            email
        };

        users.push(newUser);

        return NextResponse.json({
            success: true,
            message: "User created successfully",
            user: newUser
        }, { status: 201 });

    } catch (error) {
        return NextResponse.json(
            { error: "Invalid request body" },
            { status: 400 }
        );
    }
}
```

Dynamic API Routes

```
// app/api/users/[id]/route.ts - Handles /api/users/123
import { NextResponse } from 'next/server';

const users = [
    { id: 1, name: "John Doe", email: "john@example.com" },
    { id: 2, name: "Jane Smith", email: "jane@example.com" },
    { id: 3, name: "Bob Johnson", email: "bob@example.com" }
];

export async function GET(
    request: Request,
    { params }: { params: { id: string } }
) {
    try {
        const userId = parseInt(params.id);
        const user = users.find(u => u.id === userId);

        if (!user) {
            return NextResponse.json(
                { error: "User not found" },
                { status: 404 }
            );
        }

        return NextResponse.json({
            success: true,
            user: user
        });

    } catch (error) {
        return NextResponse.json(
            { error: "Invalid user ID" },
            { status: 400 }
        );
    }
}

export async function PUT(
    request: Request,
    { params }: { params: { id: string } }
) {
    try {
        const userId = parseInt(params.id);
        const { name, email } = await request.json();

        const userIndex = users.findIndex(u => u.id === userId);

        if (userIndex === -1) {
            return NextResponse.json(
                { error: "User not found" },
                { status: 404 }
            );
        }
```

```
        // Update user (in real app, update in database)
        users[userIndex] = { ...users[userIndex], name, email };

        return NextResponse.json({
            success: true,
            message: "User updated successfully",
            user: users[userIndex]
        });

    } catch (error) {
        return NextResponse.json(
            { error: "Internal Server Error" },
            { status: 500 }
        );
    }
}

export async function DELETE(
    request: Request,
    { params }: { params: { id: string } }
) {
    try {
        const userId = parseInt(params.id);
        const userIndex = users.findIndex(u => u.id === userId);

        if (userIndex === -1) {
            return NextResponse.json(
                { error: "User not found" },
                { status: 404 }
            );
        }

        // Remove user (in real app, delete from database)
        const deletedUser = users.splice(userIndex, 1)[^0];

        return NextResponse.json({
            success: true,
            message: "User deleted successfully",
            user: deletedUser
        });

    } catch (error) {
        return NextResponse.json(
            { error: "Internal Server Error" },
            { status: 500 }
        );
    }
}
```

Advanced API Route Features

Handling Query Parameters:

```
// app/api/search/route.ts
import { NextResponse } from 'next/server';

export async function GET(request: Request) {
    // Extract query parameters from URL
    const { searchParams } = new URL(request.url);
    const query = searchParams.get('q');
    const page = searchParams.get('page') || '1';
    const limit = searchParams.get('limit') || '10';

    // Convert query parameters to object for easier handling
    const params = Object.fromEntries(searchParams.entries());

    return NextResponse.json({
        searchQuery: query,
        pagination: {
            page: parseInt(page),
            limit: parseInt(limit)
        },
        allParams: params
    });
}

// Usage: GET /api/search?q=nextjs&page=2&limit=5
// Returns: { "searchQuery": "nextjs", "pagination": { "page": 2, "limit": 5 }, "allParam
```

Error Handling and Middleware Pattern:

```
// app/api/protected/route.ts
import { NextResponse } from 'next/server';

// Simulated authentication middleware
async function authenticate(request: Request) {
    const authorization = request.headers.get('authorization');

    if (!authorization || !authorization.startsWith('Bearer ')) {
        return null;
    }

    // In real app, verify JWT token
    const token = authorization.split(' ')[^1];
    return token === 'valid-token' ? { id: 1, name: 'John Doe' } : null;
}

export async function GET(request: Request) {
    try {
        // Check authentication
        const user = await authenticate(request);

        if (!user) {
```

```
            return NextResponse.json(
                { error: "Unauthorized access" },
                { status: 401 }
            );
        }

        return NextResponse.json({
            message: "Protected data accessed successfully",
            user: user,
            data: {
                secretInfo: "This is protected information"
            }
        });

    } catch (error) {
        console.error('API Error:', error);
        return NextResponse.json(
            { error: "Internal Server Error" },
            { status: 500 }
        );
    }
}
```

4. Data Fetching Methods

What it does: Next.js provides multiple ways to fetch data depending on when and where you need it - at build time, request time, or on the client side.

Server-Side Rendering (SSR)

When to use: When you need fresh data on every request, real-time content, or user-specific data.

```
// app/posts/page.tsx
import { Suspense } from 'react';

// This component will render on the server for each request
async function PostsList() {
    // Fetch data on the server
    const response = await fetch('https://jsonplaceholder.typicode.com/posts', {
        cache: 'no-store' // Ensure fresh data on each request
    });

    const posts = await response.json();

    return (
        <div>
            <h1>Latest Posts (SSR)</h1>
            <p>Rendered on: {new Date().toISOString()}</p>
            <ul>
                {posts.slice(0, 5).map((post: any) => (
                    <li key={post.id}>
                        <h3>{post.title}</h3>
```

```
                        <p>{post.body.substring(0, 100)}...</p>
                    </li>
                ))}
            </ul>
        </div>
    );
}

export default function PostsPage() {
    return (
        <Suspense fallback={<div>Loading posts...</div>}>
            <PostsList />
        </Suspense>
    );
}
```

Static Site Generation (SSG)

When to use: For content that doesn't change often, like blog posts, product pages, or documentation.

```
// app/blog/page.tsx
async function BlogList() {
    // This data will be fetched at build time
    const response = await fetch('https://jsonplaceholder.typicode.com/posts', {
        cache: 'force-cache' // Cache the data
    });

    const posts = await response.json();

    return (
        <div>
            <h1>Blog Posts (SSG)</h1>
            <p>Generated at build time</p>
            <div className="grid">
                {posts.map((post: any) => (
                    <article key={post.id} className="post-card">
                        <h2>{post.title}</h2>
                        <p>{post.body}</p>
                        <a href={`/blog/${post.id}`}>Read more →</a>
                    </article>
                ))}
            </div>
        </div>
    );
}

export default function BlogPage() {
    return <BlogList />;
}

// Generate static params for dynamic routes
export async function generateStaticParams() {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
```

```
    const posts = await response.json();

    return posts.map((post: any) => ({
        id: post.id.toString(),
    }));
}
```

Incremental Static Regeneration (ISR)

When to use: For content that changes occasionally but you want the benefits of static generation.

```
// app/products/page.tsx
async function ProductsList() {
    // Revalidate data every 60 seconds
    const response = await fetch('https://fakestoreapi.com/products', {
        next: { revalidate: 60 } // ISR: revalidate every 60 seconds
    });

    const products = await response.json();

    return (
        <div>
            <h1>Products (ISR)</h1>
            <p>Data revalidated every 60 seconds</p>
            <p>Last updated: {new Date().toLocaleString()}</p>

            <div className="products-grid">
                {products.map((product: any) => (
                    <div key={product.id} className="product-card">
                        <img src={product.image} alt={product.title} />
                        <h3>{product.title}</h3>
                        <p className="price">${product.price}</p>
                        <p>{product.description.substring(0, 100)}...</p>
                    </div>
                ))}
            </div>
        </div>
    );
}

export default function ProductsPage() {
    return <ProductsList />;
}
```

Client-Side Rendering (CSR)

When to use: For user-specific data, real-time updates, or when SEO isn't important.

```
// app/dashboard/page.tsx
'use client';
```

```
import { useState, useEffect } from 'react';

export default function Dashboard() {
    const [userData, setUserData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        // Fetch user data on the client side
        async function fetchUserData() {
            try {
                setLoading(true);
                const response = await fetch('/api/user/profile', {
                    headers: {
                        'Authorization': `Bearer ${localStorage.getItem('token')}`
                    }
                });

                if (!response.ok) {
                    throw new Error('Failed to fetch user data');
                }

                const data = await response.json();
                setUserData(data);
            } catch (err) {
                setError(err.message);
            } finally {
                setLoading(false);
            }
        }

        fetchUserData();
    }, []);

    if (loading) return <div className="loading">Loading dashboard...</div>;
    if (error) return <div className="error">Error: {error}</div>;

    return (
        <div className="dashboard">
            <h1>User Dashboard (CSR)</h1>
            <p>Rendered on client side after page load</p>

            {userData && (
                <div className="user-info">
                    <h2>Welcome, {userData.name}!</h2>
                    <p>Email: {userData.email}</p>
                    <p>Last login: {userData.lastLogin}</p>
                </div>
            )}

            <div className="dashboard-widgets">
                {/* Dashboard content */}
            </div>
        </div>
    );
}
```

5. Layouts and Components

What it does: Layouts provide a way to share UI elements across multiple pages, while components help organize reusable pieces of your application.

Root Layout

```
// app/layout.tsx - Root layout for entire application
import './globals.css';

export const metadata = {
    title: 'My Next.js App',
    description: 'A comprehensive Next.js application',
};

export default function RootLayout({
    children,
}: {
    children: React.ReactNode;
}) {
    return (
        <html lang="en">
            <body>
                <header className="app-header">
                    <nav>
                        <a href="/">Home</a>
                        <a href="/about">About</a>
                        <a href="/blog">Blog</a>
                        <a href="/contact">Contact</a>
                    </nav>
                </header>

                <main className="main-content">
                    {children}
                </main>

                <footer className="app-footer">
                    <p>&copy; 2024 My Next.js App. All rights reserved.</p>
                </footer>
            </body>
        </html>
    );
}
```

Nested Layouts

```
// app/blog/layout.tsx - Layout specific to blog section
export default function BlogLayout({
    children,
}: {
    children: React.ReactNode;
}) {
```

```
    return (
        <div className="blog-layout">
            <aside className="blog-sidebar">
                <h3>Categories</h3>
                <ul>
                    <li><a href="/blog/category/tech">Technology</a></li>
                    <li><a href="/blog/category/design">Design</a></li>
                    <li><a href="/blog/category/business">Business</a></li>
                </ul>

                <h3>Recent Posts</h3>
                <ul>
                    <li><a href="/blog/post-1">Latest Post</a></li>
                    <li><a href="/blog/post-2">Another Post</a></li>
                </ul>
            </aside>

            <div className="blog-content">
                {children}
            </div>
        </div>
    );
}
```

6. Middleware

What it does: Middleware runs before requests are processed, allowing you to modify responses, redirect users, or add authentication logic.

```
// middleware.ts - Root level middleware
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(request: NextRequest) {
    // Get pathname from request
    const { pathname } = request.nextUrl;

    // Authentication check for protected routes
    if (pathname.startsWith('/dashboard')) {
        const token = request.cookies.get('auth-token')?.value;

        if (!token) {
            // Redirect to login if no token
            return NextResponse.redirect(new URL('/login', request.url));
        }
    }

    // API rate limiting example
    if (pathname.startsWith('/api/')) {
        const userIP = request.ip ?? 'unknown';

        // In a real app, implement rate limiting logic here
        console.log(`API request from ${userIP} to ${pathname}`);
```

```
    }

    // Add custom headers
    const response = NextResponse.next();
    response.headers.set('X-Custom-Header', 'Next.js Middleware');

    return response;
}

// Configure which paths middleware should run on
export const config = {
    matcher: [
        '/dashboard/:path*',
        '/api/:path*',
        '/((?!_next/static|_next/image|favicon.ico).*)',
    ],
};
```

7. Environment Variables and Configuration

What it does: Environment variables allow you to configure your application for different environments (development, production) without hardcoding sensitive values.

```
// .env.local - Environment variables (never commit to version control)
DATABASE_URL=mongodb://localhost:27017/myapp
API_SECRET_KEY=your-secret-key-here
NEXTAUTH_SECRET=your-nextauth-secret
NEXT_PUBLIC_API_URL=http://localhost:3000/api

// .env.example - Template for environment variables
DATABASE_URL=your_database_url_here
API_SECRET_KEY=your_secret_key_here
NEXTAUTH_SECRET=your_nextauth_secret_here
NEXT_PUBLIC_API_URL=your_api_url_here
```

Using Environment Variables:

```
// lib/config.ts - Configuration management
export const config = {
    // Server-side only (secure)
    databaseUrl: process.env.DATABASE_URL!,
    apiSecretKey: process.env.API_SECRET_KEY!,

    // Client-side accessible (prefix with NEXT_PUBLIC_)
    publicApiUrl: process.env.NEXT_PUBLIC_API_URL!,

    // Environment check
    isDevelopment: process.env.NODE_ENV === 'development',
    isProduction: process.env.NODE_ENV === 'production',
};

// Usage in API route
```

```
// app/api/config/route.ts
import { config } from '@/lib/config';

export async function GET() {
    return Response.json({
        environment: process.env.NODE_ENV,
        publicApiUrl: config.publicApiUrl,
        // Never expose secret keys in API responses
        hasSecretKey: !!config.apiSecretKey,
    });
}
```

8. Next.js Configuration

```
// next.config.js - Next.js configuration
/** @type {import('next').NextConfig} */
const nextConfig = {
    // Enable experimental features
    experimental: {
        appDir: true, // Enable app directory
    },

    // Image domains for next/image optimization
    images: {
        domains: [
            'example.com',
            'cdn.example.com',
            'images.unsplash.com',
        ],
    },

    // Custom redirects
    async redirects() {
        return [
            {
                source: '/old-page',
                destination: '/new-page',
                permanent: true, // 301 redirect
            },
        ];
    },

    // Custom rewrites (internal redirects)
    async rewrites() {
        return [
            {
                source: '/blog/:slug',
                destination: '/posts/:slug',
            },
        ];
    },

    // Custom headers
    async headers() {
```

```
        return [
            {
                source: '/api/:path*',
                headers: [
                    {
                        key: 'Access-Control-Allow-Origin',
                        value: '*',
                    },
                ],
            },
        ];
    },

    // Environment variables
    env: {
        CUSTOM_KEY: process.env.CUSTOM_KEY,
    },

    // Webpack customization
    webpack: (config, { buildId, dev, isServer, defaultLoaders, webpack }) => {
        // Custom webpack configuration
        return config;
    },
};

module.exports = nextConfig;
```

Interview Questions & Answers

Basic Level Questions (1-15)

1. What is Next.js and how does it differ from React?

Answer: Next.js is a React framework that adds features like server-side rendering, static site generation, API routes, and file-based routing on top of React. While React is a library for building user interfaces, Next.js is a full-stack framework that provides production-ready features out of the box, including automatic code splitting, image optimization, and performance optimizations.

2. Explain the concept of Server-Side Rendering (SSR) in Next.js.

Answer: SSR renders pages on the server for each request, generating HTML before sending it to the client. This improves SEO, reduces time to first contentful paint, and ensures users see content immediately. In Next.js 13+, you can create SSR by using Server Components or by fetching data without caching (`cache: 'no-store'`).

3. What are the benefits of using Next.js over plain React?

Answer:

- **Built-in SSR/SSG**: No configuration needed

- **File-based routing**: Automatic route creation
- **API routes**: Full-stack capabilities
- **Image optimization**: Automatic image optimization
- **Performance**: Code splitting, prefetching, and caching
- **SEO-friendly**: Better search engine optimization
- **TypeScript support**: First-class TypeScript integration

4. How do you create a new Next.js project?

Answer: Use `npx create-next-app@latest my-app` for JavaScript or `npx create-next-app@latest my-app --typescript` for TypeScript. Then navigate with `cd my-app` and start development with `npm run dev`.

5. Explain the file-system based routing in Next.js.

Answer: Next.js uses the file structure in the `app` directory (App Router) or `pages` directory (Pages Router) to create routes automatically. Each folder represents a route segment, and `page.tsx` files define the UI for that route. Dynamic routes use brackets like `[id]`, and catch-all routes use `[...slug]`.

6. What is the purpose of the pages directory in Next.js?

Answer: The `pages` directory (in Pages Router) automatically creates routes based on file names. Each file exports a React component that becomes a page. Special files like `_app.js`, `_document.js`, and `_error.js` have specific purposes for customizing the application structure.

7. How do you handle static assets in Next.js?

Answer: Static assets go in the `public` directory and are served from the root URL. Use `next/image` for optimized images, which provides automatic optimization, lazy loading, and responsive images. Reference static files with `/filename.ext`.

8. What is the _app.js file used for?

Answer: `_app.js` (or `app/layout.tsx` in App Router) wraps all pages and allows you to:

- Add global CSS
- Maintain state between page changes
- Add global providers (Auth, Theme, etc.)
- Handle page initialization
- Add custom error handling

9. How do you create dynamic routes in Next.js?

Answer: Create dynamic routes using square brackets in file/folder names:

- `[id].tsx` for single dynamic segment
- `[...slug].tsx` for catch-all routes

- Access parameters via `params` prop in Server Components or `useRouter` in Client Components

10. Explain the difference between getStaticProps and getServerSideProps.

Answer: (Pages Router only)

- **getStaticProps**: Runs at build time for SSG, generates static HTML
- **getServerSideProps**: Runs on each request for SSR, generates HTML per request
  In App Router, use Server Components with different caching strategies instead.

11. What is Incremental Static Regeneration (ISR) and how does it work?

Answer: ISR allows you to update static content after deployment without rebuilding the entire site. It works by revalidating pages in the background when requests come in after the revalidation period. Use `next: { revalidate: 60 }` in fetch options or `revalidate` in `generateStaticParams`.

12. How do you implement API routes in Next.js?

Answer: Create files in `app/api` directory with named exports for HTTP methods:

```
// app/api/users/route.ts
export async function GET() {
    return Response.json({ users: [] });
}

export async function POST(request: Request) {
    const data = await request.json();
    return Response.json({ success: true });
}
```

13. Explain how to use middleware in Next.js.

Answer: Create `middleware.ts` in the root directory to run code before request completion. Use it for authentication, redirects, and request modification:

```
export function middleware(request: NextRequest) {
    // Middleware logic
    return NextResponse.next();
}

export const config = {
    matcher: ['/dashboard/:path*']
};
```

14. What are environment variables in Next.js and how do you use them?

Answer: Environment variables configure your app for different environments. Create `.env.local` for secrets (server-only) and prefix with `NEXT_PUBLIC_` for client-accessible variables. Access with `process.env.VARIABLE_NAME`.

15. How do you optimize images in Next.js?

Answer: Use the `next/image` component which provides:

- Automatic optimization and format conversion
- Lazy loading and blur placeholders
- Responsive images with `sizes` prop
- Priority loading for above-fold images
  Configure allowed domains in `next.config.js`.

Intermediate Level Questions (16-25)

16. What is shallow routing in Next.js?

Answer: Shallow routing updates the URL without running data fetching methods. Use `router.push(url, as, { shallow: true })` to change the URL without triggering page re-renders or data fetching.

17. How do you implement internationalization (i18n) in Next.js?

Answer: Configure i18n in `next.config.js` with locales and default locale. Next.js automatically handles routing for different languages. Use libraries like `next-i18next` or `react-intl` for translations.

18. What is next/script and how does it improve performance?

Answer: `next/script` optimizes third-party script loading with strategies:

- `beforeInteractive`: Load before page becomes interactive
- `afterInteractive`: Load after page becomes interactive (default)
- `lazyOnload`: Load during idle time
- `worker`: Load in a web worker (experimental)

19. How do you handle SEO in Next.js applications?

Answer: Next.js provides several SEO features:

- `metadata` API for meta tags in App Router
- `next/head` for custom head elements
- Automatic sitemap generation
- Server-side rendering for search engines
- Image optimization for better Core Web Vitals

20. Explain the concept of Next.js rewrites and redirects.

Answer:

- **Redirects**: Send users to different URLs (301/302 status codes)

- **Rewrites**: Internal URL mapping without changing the browser URL
  Both are configured in `next.config.js` and support pattern matching.

21. How does Next.js implement code splitting?

Answer: Next.js automatically splits code by:

- Route-based splitting (each page is a separate bundle)
- Dynamic imports for component-level splitting
- Shared dependencies are optimized into separate chunks
- Automatic prefetching of linked pages

22. What are React Server Components in Next.js 13+ and their benefits?

Answer: Server Components render on the server, reducing bundle size and enabling direct database access. Benefits include:

- Zero impact on bundle size
- Direct backend resource access
- Better security (server-only code)
- Improved performance for data fetching

23. What are Edge Functions in Next.js and when would you use them?

Answer: Edge Functions run closer to users at edge locations, providing lower latency. Use them for:

- Authentication middleware
- A/B testing
- Redirects based on geography
- Simple API operations
  They have runtime limitations but offer better performance.

24. How do you optimize Core Web Vitals in a Next.js application?

Answer:

- **LCP**: Optimize images with `next/image`, use proper caching
- **FID**: Minimize JavaScript, use code splitting
- **CLS**: Specify image dimensions, avoid layout shifts
- Use Next.js built-in analytics and performance monitoring

25. Explain the App Router vs Pages Router architecture.

Answer:

- **Pages Router**: File-based routing in `pages` directory, uses `getServerSideProps`/`getStaticProps`

- **App Router**: New routing system in `app` directory, uses Server Components, nested layouts, and streaming
  App Router provides better performance and developer experience.

Advanced Level Questions (26-30)

26. How would you handle complex state management in a large Next.js application?

Answer: Use appropriate state management solutions:

- **Local state**: useState, useReducer for component-level state
- **Global state**: Context API, Zustand, or Redux Toolkit
- **Server state**: React Query/TanStack Query or SWR
- **Form state**: React Hook Form or Formik
  Choose based on complexity and requirements.

27. What are Streaming and Suspense in Next.js 13+ and how do they improve UX?

Answer: Streaming sends HTML in chunks as components render, while Suspense shows loading states for pending components. This improves perceived performance by showing content progressively instead of waiting for the entire page.

28. How do you implement custom server-side logic with Next.js?

Answer:

- Use API Routes for custom endpoints
- Server Components for server-side rendering logic
- Middleware for request processing
- Custom server with `server.js` (advanced use cases)
- Edge API Routes for edge computing

29. How would you implement a micro-frontend architecture using Next.js?

Answer: Strategies include:

- **Module Federation**: Share components between applications
- **Subdomain routing**: Separate Next.js apps per subdomain
- **Dynamic imports**: Load micro-frontends as needed
- **API-first architecture**: Shared backend services
  Consider build complexity and deployment strategies.

30. How do you handle authentication in Next.js applications?

Answer: Common patterns:

- **NextAuth.js**: Full-featured auth library
- **JWT tokens**: Store in httpOnly cookies

- **Session management**: Server-side sessions with database
- **OAuth providers**: Social login integration
- **Middleware**: Protect routes automatically
  Choose based on security requirements and complexity.

This comprehensive guide covers all essential Next.js concepts with practical examples and common interview questions. The code snippets demonstrate real-world usage patterns that you can reference and adapt for your projects. Each section builds upon previous concepts, ensuring a complete understanding of Next.js development.

⁂

1. https://mentorcruise.com/questions/nextjs/
2. https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering
3. https://supertokens.com/blog/mastering-nextjs-api-routes
4. https://www.lambdatest.com/learning-hub/next-js-interview-questions
5. https://www.reddit.com/r/nextjs/comments/1fj9vri/in_nextjs_everyones_all_ssr_ssg_rsc_in_their_spa/
6. https://nextjs.org/docs/14/app/building-your-application/data-fetching/patterns
7. https://github.com/mrhrifat/nextjs-interview-questions
8. https://www.freecodecamp.org/news/the-nextjs-15-streaming-handbook/
9. https://nextjs.org/blog/building-apis-with-nextjs
10. https://codeinterview.io/interview-questions/next-js-questions-answers
11. https://strapi.io/blog/ssr-vs-ssg-in-nextjs-differences-advantages-and-use-cases
12. https://www.geeksforgeeks.org/nextjs/nextjs-tutorial/
13. https://nextjs.org/docs/pages/building-your-application/routing/api-routes
14. https://www.finalroundai.com/blog/nextjs-interview-questions
15. https://tech.groww.in/ssr-vs-csr-vs-ssg-in-next-js-510422994741
16. https://engineering.udacity.com/5-steps-to-create-professional-api-routes-in-next-js-201e726ead48
17. https://www.geeksforgeeks.org/reactjs/next-js-interview-questions-answers/
18. https://theodorusclarence.com/blog/nextjs-fetch-method
19. https://makerkit.dev/blog/tutorials/nextjs-api-best-practices