Next.js Interview Questions and Answers

Beginner Level Questions (1-20)

1. What is Next.js and why should we use it?

Answer: Next.js is a React framework that provides production-ready features like server-side rendering (SSR), static site generation (SSG), API routes, and automatic code splitting.

Benefits:

- Built-in performance optimizations
- SEO-friendly with SSR/SSG
- File-based routing
- API routes for backend functionality
- Image optimization
- TypeScript support out of the box

```jsx
// Example of a simple Next.js page
export default function HomePage() {
  return <h1>Welcome to Next.js!</h1>
}
```

2. Explain the difference between pages router and app router in Next.js

Answer: Next.js offers two routing systems:

Pages Router (Traditional):

- Files in `/pages` directory become routes
- Uses `getServerSideProps`, `getStaticProps`
- Component-based approach

App Router (New):

- Files in `/app` directory
- Uses React Server Components
- Layout-based approach with better performance

```jsx
// Pages Router: pages/about.js
export default function About() {
  return <h1>About Page</h1>
}

// App Router: app/about/page.js
export default function About() {
  return <h1>About Page</h1>
}
```

3. How does file-based routing work in Next.js?

Answer: Next.js automatically creates routes based on the file structure in the `pages` or `app` directory.

```
pages/
  index.js        → /
  about.js        → /about
  blog/
    index.js      → /blog
    [id].js       → /blog/:id
    [...slug].js  → /blog/* (catch-all)
```

4. What are the different rendering methods in Next.js?

Answer: Next.js supports four main rendering methods:

1. **Static Site Generation (SSG)** - Pages built at build time
2. **Server-Side Rendering (SSR)** - Pages rendered on each request
3. **Incremental Static Regeneration (ISR)** - Static pages updated after deployment
4. **Client-Side Rendering (CSR)** - Traditional React rendering

jsx

```jsx
// SSG
export async function getStaticProps() {
  const data = await fetchData()
  return { props: { data } }
}

// SSR
export async function getServerSideProps() {
  const data = await fetchData()
  return { props: { data } }
}
```

5. What is the purpose of _app.js file?

Answer: The `_app.js` file is used to initialize pages and allows you to:

- Persist layout between page changes
- Keep state when navigating
- Add global CSS
- Pass additional data to pages

jsx

```jsx
// pages/_app.js
import '../styles/globals.css'

function MyApp({ Component, pageProps }) {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  )
}

export default MyApp
```

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

Answer: Dynamic routes are created using square brackets in filenames:

jsx

```jsx
// pages/posts/[id].js
import { useRouter } from 'next/router'

export default function Post() {
  const router = useRouter()
  const { id } = router.query

  return <h1>Post: {id}</h1>
}

// To generate paths for SSG
export async function getStaticPaths() {
  return {
    paths: [
      { params: { id: '1' } },
      { params: { id: '2' } },
    ],
    fallback: false
  }
}
```

7. What is the difference between Link and a tag in Next.js?

Answer: The `Link` component provides client-side navigation with prefetching, while `<a>` causes full page reload.

jsx

```jsx
import Link from 'next/link'

// Client-side navigation with prefetching
<Link href="/about">
  <a>About Us</a>
</Link>

// Full page reload
<a href="/about">About Us</a>
```

8. How do you add global CSS in Next.js?

Answer: Global CSS can only be imported in `_app.js`:

```jsx
// pages/_app.js
import '../styles/globals.css'

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

For component-level styles, use CSS Modules:

```jsx
// components/Button.module.css
.button {
  background: blue;
  color: white;
}

// components/Button.js
import styles from './Button.module.css'

export default function Button() {
  return <button className={styles.button}>Click me</button>
}
```

9. What is next/image and why should we use it?

Answer: The `next/image` component is an optimized image component that provides:

- Automatic lazy loading
- Image optimization
- Responsive images
- Prevents layout shift

```jsx
import Image from 'next/image'

export default function Profile() {
  return (
    <Image
      src="/profile.jpg"
      alt="Profile"
      width={500}
      height={500}
      priority // Load image with high priority
    />
  )
}
```

10. How do you handle environment variables in Next.js?

Answer: Next.js supports environment variables through `.env` files:

```bash
# .env.local
DATABASE_URL=postgresql://localhost:5432/mydb
NEXT_PUBLIC_API_URL=https://api.example.com
```

- Variables prefixed with `NEXT_PUBLIC_` are exposed to the browser
- Others are only available server-side

```jsx
// Server-side only
const dbUrl = process.env.DATABASE_URL

// Available in browser
const apiUrl = process.env.NEXT_PUBLIC_API_URL
```

11. What are API routes in Next.js?

Answer: API routes allow you to build backend endpoints within your Next.js app:

jsx

```jsx
// pages/api/users.js
export default function handler(req, res) {
  if (req.method === 'GET') {
    res.status(200).json({ users: ['John', 'Jane'] })
  } else if (req.method === 'POST') {
    const { name } = req.body
    res.status(201).json({ message: `User ${name} created` })
  }
}
```

12. How do you redirect in Next.js?

Answer: There are multiple ways to redirect:

```jsx
// 1. In getServerSideProps
export async function getServerSideProps() {
  return {
    redirect: {
      destination: '/login',
      permanent: false,
    },
  }
}

// 2. Using router
import { useRouter } from 'next/router'

function Profile() {
  const router = useRouter()

  const handleClick = () => {
    router.push('/dashboard')
  }
}

// 3. In next.config.js
module.exports = {
  async redirects() {
    return [
      {
        source: '/old-page',
        destination: '/new-page',
        permanent: true,
      },
    ]
  },
}
```

13. What is the purpose of next.config.js?

Answer: `next.config.js` is used to customize Next.js configuration:

jsx

```jsx
module.exports = {
  reactStrictMode: true,
  images: {
    domains: ['example.com'],
  },
  env: {
    CUSTOM_KEY: 'value',
  },
  async rewrites() {
    return [
      {
        source: '/api/:path*',
        destination: 'https://api.example.com/:path*',
      },
    ]
  },
}
```

14. How do you handle 404 pages in Next.js?

Answer: Create a `404.js` file in the pages directory:

jsx

```jsx
// pages/404.js
export default function Custom404() {
  return (
    <div>
      <h1>404 - Page Not Found</h1>
      <Link href="/">
        <a>Go back home</a>
      </Link>
    </div>
  )
}
```

15. What is shallow routing in Next.js?

Answer: Shallow routing allows URL changes without running data fetching methods:

```jsx
import { useRouter } from 'next/router'

function FilteredList() {
  const router = useRouter()

  const updateFilter = (filter) => {
    router.push(`/?filter=${filter}`, undefined, { shallow: true })
  }

  return (
    <button onClick={() => updateFilter('active')}>
      Show Active
    </button>
  )
}
```

16. How do you prefetch pages in Next.js?

Answer: Next.js automatically prefetches pages linked with the Link component:

```jsx
import Link from 'next/link'

// Automatic prefetching (default)
<Link href="/about">
  <a>About</a>
</Link>

// Disable prefetching
<Link href="/about" prefetch={false}>
  <a>About</a>
</Link>

// Programmatic prefetching
router.prefetch('/dashboard')
```

17. What is the difference between getStaticProps and getServerSideProps?

Answer:

getStaticProps:

- Runs at build time
- Used for Static Site Generation

- Page is pre-rendered and cached

getServerSideProps:

- Runs on every request
- Used for Server-Side Rendering
- Fresh data on each request

```jsx
// Static Generation
export async function getStaticProps() {
  const posts = await fetchPosts()
  return {
    props: { posts },
    revalidate: 60 // ISR: regenerate every 60 seconds
  }
}

// Server-Side Rendering
export async function getServerSideProps(context) {
  const posts = await fetchUserPosts(context.query.userId)
  return {
    props: { posts }
  }
}
```

18. How do you deploy a Next.js application?

Answer: Next.js can be deployed in multiple ways:

1. **Vercel** (Recommended) - Zero configuration
2. **Static Export** - `next export` for static hosting
3. **Node.js Server** - Custom server deployment
4. **Docker** - Containerized deployment

json

```json
// package.json scripts
{
  "scripts": {
    "build": "next build",
    "start": "next start",
    "export": "next export"
  }
}
```

19. What are CSS Modules in Next.js?

Answer: CSS Modules locally scope CSS by automatically generating unique class names:

css

```css
/* Button.module.css */
.button {
  background-color: blue;
  color: white;
  padding: 10px 20px;
}

.button:hover {
  background-color: darkblue;
}
```

jsx

```jsx
// Button.js
import styles from './Button.module.css'

export default function Button() {
  return (
    <button className={styles.button}>
      Click me
    </button>
  )
}
```

20. How do you handle metadata in Next.js pages?

Answer: Use the `Head` component to manage metadata:

```jsx
import Head from 'next/head'

export default function About() {
  return (
    <>
      <Head>
        <title>About Us - My Site</title>
        <meta name="description" content="Learn more about our company" />
        <meta property="og:title" content="About Us" />
        <link rel="canonical" href="https://mysite.com/about" />
      </Head>
      <h1>About Us</h1>
    </>
  )
}
```

Intermediate Level Questions (21-40)

21. Explain Incremental Static Regeneration (ISR) in Next.js

Answer: ISR allows you to update static pages after deployment without rebuilding the entire site:

```jsx
export async function getStaticProps() {
  const posts = await fetchPosts()

  return {
    props: { posts },
    revalidate: 60, // Regenerate page every 60 seconds
  }
}

// With on-demand revalidation
export default async function handler(req, res) {
  try {
    await res.revalidate('/path-to-revalidate')
    return res.json({ revalidated: true })
  } catch (err) {
    return res.status(500).send('Error revalidating')
  }
}
```

22. How do you implement middleware in Next.js?

Answer: Middleware runs before a request is completed:

```jsx
// middleware.js (at root level)
import { NextResponse } from 'next/server'

export function middleware(request) {
  // Check authentication
  const token = request.cookies.get('token')

  if (!token && request.nextUrl.pathname.startsWith('/admin')) {
    return NextResponse.redirect(new URL('/login', request.url))
  }

  // Add custom headers
  const response = NextResponse.next()
  response.headers.set('x-custom-header', 'value')

  return response
}

export const config = {
  matcher: ['/admin/:path*', '/api/:path*']
}
```

23. How do you optimize performance in Next.js applications?

Answer: Key optimization strategies:

1. **Image Optimization:**

```jsx
import Image from 'next/image'

<Image
  src="/hero.jpg"
  alt="Hero"
  width={1200}
  height={600}
  priority
  placeholder="blur"
  blurDataURL={blurDataUrl}
/>
```

2. **Code Splitting:**

```jsx
import dynamic from 'next/dynamic'

const DynamicComponent = dynamic(() => import('../components/Heavy'), {
  loading: () => <p>Loading...</p>,
  ssr: false
})
```

3. **Font Optimization:**

```jsx
// app/layout.js
import { Inter } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})
```

4. **Bundle Analysis:**

```json
// package.json
{
  "scripts": {
    "analyze": "ANALYZE=true next build"
  }
}
```

24. Explain data fetching in the App Router

Answer: App Router uses React Server Components with new patterns:

jsx

```jsx
// app/posts/page.js
// Server Component (default)
async function getPosts() {
  const res = await fetch('https://api.example.com/posts', {
    cache: 'force-cache', // Static data
    // or
    cache: 'no-store', // Dynamic data
    // or
    next: { revalidate: 60 } // ISR
  })
  return res.json()
}

export default async function Posts() {
  const posts = await getPosts()

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}

// Client Component
'use client'

import { useState, useEffect } from 'react'

export default function ClientPosts() {
  const [posts, setPosts] = useState([])

  useEffect(() => {
    fetch('/api/posts')
      .then(res => res.json())
      .then(setPosts)
  }, [])

  return <div>{/* render posts */}</div>
}
```

25. How do you handle authentication in Next.js?

Answer: Common authentication patterns:

```
jsx
```

```javascript
// Using NextAuth.js
// pages/api/auth/[...nextauth].js
import NextAuth from 'next-auth'
import GoogleProvider from 'next-auth/providers/google'

export default NextAuth({
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_ID,
      clientSecret: process.env.GOOGLE_SECRET,
    })
  ],
  callbacks: {
    async jwt({ token, user }) {
      if (user) {
        token.id = user.id
      }
      return token
    },
    async session({ session, token }) {
      session.user.id = token.id
      return session
    }
  }
})

// Protected API Route
import { getSession } from 'next-auth/react'

export default async function handler(req, res) {
  const session = await getSession({ req })

  if (!session) {
    return res.status(401).json({ error: 'Unauthorized' })
  }

  // Handle authenticated request
}

// Protected Page
export async function getServerSideProps(context) {
  const session = await getSession(context)

  if (!session) {
    return {
      redirect: {
          destination: '/login',
          permanent: false,
        },
      }
    }

    return {
      props: { session },
    }
  }
```

26. What are Server Components vs Client Components?

Answer: In App Router, components are Server Components by default:

Server Components:

- Run on server only
- Can fetch data directly
- No JavaScript sent to client
- Cannot use browser APIs or hooks

Client Components:

- Run on client
- Can use hooks and browser APIs
- Marked with 'use client' directive

jsx

```jsx
// Server Component (default)
// app/posts/page.js
async function Posts() {
  const posts = await db.posts.findMany()
  return <PostList posts={posts} />
}

// Client Component
// app/components/LikeButton.js
'use client'

import { useState } from 'react'

export function LikeButton() {
  const [likes, setLikes] = useState(0)
  return (
    <button onClick={() => setLikes(likes + 1)}>
      Likes: {likes}
    </button>
  )
}
```

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

Answer: Next.js has built-in i18n support:

jsx

```jsx
// next.config.js
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'es'],
    defaultLocale: 'en',
    localeDetection: true,
  },
}

// pages/index.js
import { useRouter } from 'next/router'

export default function Home({ content }) {
  const { locale, locales, asPath } = useRouter()

  return (
    <div>
      <h1>{content.title}</h1>
      <select
        value={locale}
        onChange={(e) => router.push(asPath, asPath, { locale: e.target.value })}
      >
        {locales.map((l) => (
          <option key={l} value={l}>{l}</option>
        ))}
      </select>
    </div>
  )
}

export async function getStaticProps({ locale }) {
  const content = await import(`../locales/${locale}.json`)

  return {
    props: {
      content: content.default,
    },
  }
}
```

28. How do you handle errors in Next.js?

Answer: Error handling strategies:

```jsx
// pages/_error.js - Custom error page
function Error({ statusCode, err }) {
  return (
    <p>
      {statusCode
        ? `An error ${statusCode} occurred on server`
        : 'An error occurred on client'}
    </p>
  )
}

Error.getInitialProps = ({ res, err }) => {
  const statusCode = res ? res.statusCode : err ? err.statusCode : 404
  return { statusCode }
}

// app/error.js - App Router error boundary
'use client'

export default function Error({ error, reset }) {
  return (
    <div>
      <h2>Something went wrong!</h2>
      <button onClick={() => reset()}>Try again</button>
    </div>
  )
}

// API Route error handling
export default async function handler(req, res) {
  try {
    const data = await riskyOperation()
    res.status(200).json(data)
  } catch (error) {
    console.error('API Error:', error)
    res.status(500).json({ error: 'Internal Server Error' })
  }
}
```

29. Explain the Next.js build output and optimization

Answer: The build process creates optimized production files:

bash

```bash
# Build output analysis
next build

# Output includes:
# - Page sizes
# - First Load JS
# - Static/SSG/SSR pages
# - API routes

# Optimization techniques:
# 1. Tree shaking
# 2. Code splitting
# 3. Minification
# 4. Compression

# Analyze bundle
npm install @next/bundle-analyzer

// next.config.js
const withBundleAnalyzer = require('@next/bundle-analyzer')({
  enabled: process.env.ANALYZE === 'true',
})

module.exports = withBundleAnalyzer({
  // config
})
```

30. How do you implement streaming in Next.js 13+?

Answer: Streaming allows progressive rendering:

```jsx
// app/page.js
import { Suspense } from 'react'

async function SlowComponent() {
  const data = await fetch('https://slow-api.com/data')
  return <div>{data}</div>
}

export default function Page() {
  return (
    <div>
      <h1>Instant Header</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <SlowComponent />
      </Suspense>
    </div>
  )
}

// With loading.js
// app/dashboard/loading.js
export default function Loading() {
  return <div>Loading dashboard...</div>
}
```

31. What are Route Handlers in App Router?

Answer: Route Handlers replace API Routes in App Router:

jsx

```jsx
// app/api/posts/route.js
import { NextResponse } from 'next/server'

export async function GET(request) {
  const { searchParams } = new URL(request.url)
  const id = searchParams.get('id')

  const posts = await fetchPosts(id)

  return NextResponse.json(posts)
}

export async function POST(request) {
  const body = await request.json()
  const post = await createPost(body)

  return NextResponse.json(post, { status: 201 })
}

// Dynamic route handler
// app/api/posts/[id]/route.js
export async function GET(request, { params }) {
  const post = await getPost(params.id)
  return NextResponse.json(post)
}
```

32. How do you implement rate limiting in Next.js API routes?

Answer: Rate limiting protects APIs from abuse:

```jsx
// Using a simple in-memory store
const rateLimit = new Map()

function rateLimiter(req) {
  const ip = req.headers['x-forwarded-for'] || req.connection.remoteAddress
  const now = Date.now()
  const windowMs = 60 * 1000 // 1 minute
  const maxRequests = 10

  if (!rateLimit.has(ip)) {
    rateLimit.set(ip, [])
  }

  const requests = rateLimit.get(ip)
  const recentRequests = requests.filter(time => now - time < windowMs)

  if (recentRequests.length >= maxRequests) {
    return false
  }

  recentRequests.push(now)
  rateLimit.set(ip, recentRequests)

  return true
}

// API Route
export default function handler(req, res) {
  if (!rateLimiter(req)) {
    return res.status(429).json({ error: 'Too many requests' })
  }

  // Handle request
}
```

33. How do you handle database connections in Next.js?

Answer: Efficient database connection management:

```jsx
// lib/prisma.js - Using Prisma
import { PrismaClient } from '@prisma/client'

let prisma

if (process.env.NODE_ENV === 'production') {
  prisma = new PrismaClient()
} else {
  // Prevent multiple instances in development
  if (!global.prisma) {
    global.prisma = new PrismaClient()
  }
  prisma = global.prisma
}

export default prisma

// Usage in API route
import prisma from '../../lib/prisma'

export default async function handler(req, res) {
  const posts = await prisma.post.findMany()
  res.json(posts)
}
```

34. What are Parallel Routes in Next.js App Router?

Answer: Parallel routes allow rendering multiple pages in the same layout:

```jsx
// app/layout.js
export default function Layout({ children, team, analytics }) {
  return (
    <>
      {children}
      <div>{team}</div>
      <div>{analytics}</div>
    </>
  )
}

// app/@team/page.js
export default function Team() {
  return <div>Team Dashboard</div>
}

// app/@analytics/page.js
export default function Analytics() {
  return <div>Analytics Dashboard</div>
}
```

35. How do you implement WebSockets in Next.js?

Answer: WebSockets require a custom server:

```
jsx
```

```javascript
// server.js
const { createServer } = require('http')
const { parse } = require('url')
const next = require('next')
const { Server } = require('socket.io')

const dev = process.env.NODE_ENV !== 'production'
const app = next({ dev })
const handle = app.getRequestHandler()

app.prepare().then(() => {
  const server = createServer((req, res) => {
    const parsedUrl = parse(req.url, true)
    handle(req, res, parsedUrl)
  })

  const io = new Server(server)

  io.on('connection', (socket) => {
    console.log('Client connected')

    socket.on('message', (msg) => {
      io.emit('message', msg)
    })
  })

  server.listen(3000, (err) => {
    if (err) throw err
    console.log('> Ready on http://localhost:3000')
  })
})

// Client-side
import { useEffect } from 'react'
import io from 'socket.io-client'

export default function Chat() {
  useEffect(() => {
    const socket = io()

    socket.on('message', (msg) => {
      console.log('Received:', msg)
    })

    return () => socket.disconnect()
  }, [])
}
```

36. How do you optimize SEO in Next.js?

Answer: SEO optimization strategies:

jsx

```
// Using Next.js Head
import Head from 'next/head'

export default function SEOPage({ post }) {
  return (
    <>
      <Head>
        <title>{post.title} | My Site</title>
        <meta name="description" content={post.excerpt} />
        <meta property="og:title" content={post.title} />
        <meta property="og:description" content={post.excerpt} />
        <meta property="og:image" content={post.image} />
        <meta property="og:type" content="article" />
        <meta name="twitter:card" content="summary_large_image" />
        <link rel="canonical" href={`https://mysite.com/posts/${post.slug}`} />
      </Head>
      {/* Page content */}
    </>
  )
}

// Structured data
export default function Product({ product }) {
  const structuredData = {
    '@context': 'https://schema.org',
    '@type': 'Product',
    name: product.name,
    description: product.description,
    price: product.price,
  }

  return (
    <>
      <Head>
        <script
          type="application/ld+json"
          dangerouslySetInnerHTML={{ __html: JSON.stringify(structuredData) }}
        />
      </Head>
      {/* Product content */}
    </>
  )
}

// Sitemap generation
// pages/sitemap.xml.js
```

```
export async function getServerSideProps({ res }) {
  const posts = await getAllPosts()

  const sitemap = `<?xml version="1.0" encoding="UTF-8"?>
    <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
      ${posts.map(post => `
        <url>
          <loc>https://mysite.com/posts/${post.slug}</loc>
          <lastmod>${post.updatedAt}</lastmod>
          <priority>0.8</priority>
        </url>
      `).join('')}
    </urlset>`

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

  return { props: {} }
}
```

37. How do you implement Progressive Web App (PWA) features?

Answer: Adding PWA capabilities to Next.js:

jsx

```
// next.config.js
const withPWA = require('next-pwa')({
  dest: 'public',
  register: true,
  skipWaiting: true,
})

module.exports = withPWA({
  // Next.js config
})

// public/manifest.json
{
  "name": "My Next.js PWA",
  "short_name": "NextPWA",
  "description": "A Progressive Web App built with Next.js",
  "start_url": "/",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff",
  "icons": [
    {
      "src": "/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}

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

export default function Document() {
  return (
    <Html>
      <Head>
        <link rel="manifest" href="/manifest.json" />
        <meta name="theme-color" content="#000000" />
        <link rel="apple-touch-icon" href="/icon-192x192.png" />
      </Head>
      <body>
```

```
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

38. How do you handle file uploads in Next.js?

Answer: File upload implementation:

jsx

```js
// pages/api/upload.js
import formidable from 'formidable'
import fs from 'fs'

export const config = {
  api: {
    bodyParser: false,
  },
}

export default async function handler(req, res) {
  if (req.method !== 'POST') {
    return res.status(405).json({ error: 'Method not allowed' })
  }

  const form = new formidable.IncomingForm({
    uploadDir: './public/uploads',
    keepExtensions: true,
    maxFileSize: 10 * 1024 * 1024, // 10MB
  })

  form.parse(req, (err, fields, files) => {
    if (err) {
      return res.status(500).json({ error: 'Upload failed' })
    }

    const file = files.file
    const newPath = `./public/uploads/${Date.now()}-${file.originalFilename}`

    fs.rename(file.filepath, newPath, (err) => {
      if (err) {
        return res.status(500).json({ error: 'File save failed' })
      }

      res.status(200).json({
        message: 'Upload successful',
        path: newPath.replace('./public', '')
      })
    })
  })
}

// Client-side
function UploadForm() {
  const handleUpload = async (e) => {
    e.preventDefault()
```

```
      const formData = new FormData()
      formData.append('file', e.target.files[0])

      const res = await fetch('/api/upload', {
        method: 'POST',
        body: formData,
      })

      const data = await res.json()
      console.log('Uploaded:', data.path)
    }

    return (
      <form>
        <input type="file" onChange={handleUpload} />
      </form>
    )
  }
```

39. What are Server Actions in Next.js?

Answer: Server Actions allow direct server mutations from components:

jsx

```jsx
// app/actions.js
'use server'

import { revalidatePath } from 'next/cache'
import { redirect } from 'next/navigation'

export async function createPost(formData) {
  const title = formData.get('title')
  const content = formData.get('content')

  await db.post.create({
    data: { title, content }
  })

  revalidatePath('/posts')
  redirect('/posts')
}

// app/new-post/page.js
import { createPost } from '../actions'

export default function NewPost() {
  return (
    <form action={createPost}>
      <input name="title" type="text" required />
      <textarea name="content" required />
      <button type="submit">Create Post</button>
    </form>
  )
}

// With useFormStatus
'use client'
import { useFormStatus } from 'react-dom'

function SubmitButton() {
  const { pending } = useFormStatus()

  return (
    <button type="submit" disabled={pending}>
      {pending ? 'Creating...' : 'Create Post'}
    </button>
  )
}
```

40. How do you implement caching strategies in Next.js?

Answer: Various caching approaches:

```jsx
// 1. Data Cache (fetch)
fetch('https://api.example.com/data', {
  cache: 'force-cache', // Default, cache indefinitely
  // or
  cache: 'no-store', // Never cache
  // or
  next: {
    revalidate: 3600, // Cache for 1 hour
    tags: ['posts'] // For on-demand revalidation
  }
})

// 2. Full Route Cache
export const dynamic = 'force-static' // Force static rendering
export const revalidate = 3600 // Revalidate every hour

// 3. Router Cache (client-side)
// Automatically handled by Next.js

// 4. On-demand revalidation
import { revalidateTag, revalidatePath } from 'next/cache'

export async function POST() {
  revalidateTag('posts') // Revalidate all with 'posts' tag
  revalidatePath('/posts') // Revalidate specific path

  return Response.json({ revalidated: true })
}

// 5. Custom cache headers
export async function GET() {
  return new Response('Hello', {
    headers: {
      'Cache-Control': 'public, s-maxage=3600, stale-while-revalidate',
    },
  })
}
```

Advanced Level Questions (41-60)

41. How do you implement custom webpack configuration in Next.js?

Answer: Advanced webpack customization:

```jsx
// next.config.js
module.exports = {
  webpack: (config, { buildId, dev, isServer, defaultLoaders, webpack }) => {
    // Add custom webpack plugins
    config.plugins.push(
      new webpack.DefinePlugin({
        'process.env.BUILD_ID': JSON.stringify(buildId),
      })
    )

    // Modify module rules
    config.module.rules.push({
      test: /\.svg$/,
      use: ['@svgr/webpack'],
    })

    // Add aliases
    config.resolve.alias = {
      ...config.resolve.alias,
      '@components': path.resolve(__dirname, 'components'),
      '@utils': path.resolve(__dirname, 'utils'),
    }

    // Optimize bundle splitting
    if (!isServer) {
      config.optimization.splitChunks.cacheGroups = {
        ...config.optimization.splitChunks.cacheGroups,
        commons: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all',
        },
      }
    }

    return config
  },
}
```

42. Explain the Next.js rendering pipeline and hydration process

Answer: The complete rendering flow:

```jsx
// 1. Server-side rendering process
// - React components render to HTML
// - Data fetching occurs on server
// - HTML sent to client

// 2. Hydration process
// - React attaches event handlers
// - Components become interactive
// - Reconciliation with server HTML

// Hydration errors and solutions
export default function Component() {
  // Avoid hydration mismatches
  const [mounted, setMounted] = useState(false)

  useEffect(() => {
    setMounted(true)
  }, [])

  // Only render client-specific content after mount
  if (!mounted) {
    return <div>Loading...</div>
  }

  return <div>{new Date().toLocaleTimeString()}</div>
}

// Selective hydration with Suspense
import { Suspense } from 'react'

export default function Page() {
  return (
    <Suspense fallback={<Loading />}>
      <HeavyComponent />
    </Suspense>
  )
}
```

43. How do you implement advanced security measures in Next.js?

Answer: Comprehensive security implementation:

jsx

```javascript
// 1. Content Security Policy
// middleware.js
import { NextResponse } from 'next/server'

export function middleware(request) {
  const nonce = Buffer.from(crypto.randomUUID()).toString('base64')
  const cspHeader = `
    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strict-dynamic';
    style-src 'self' 'unsafe-inline';
    img-src 'self' blob: data:;
    font-src 'self';
    object-src 'none';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    upgrade-insecure-requests;
  `.replace(/\s{2,}/g, ' ').trim()

  const response = NextResponse.next()
  response.headers.set('Content-Security-Policy', cspHeader)
  response.headers.set('X-Nonce', nonce)

  return response
}

// 2. CSRF Protection
// pages/api/csrf.js
import csrf from 'csrf'

const tokens = new csrf()

export default function handler(req, res) {
  if (req.method === 'GET') {
    const secret = tokens.secretSync()
    const token = tokens.create(secret)

    res.setHeader('Set-Cookie', `csrf-secret=${secret}; HttpOnly; Secure; SameSite=Str
    res.json({ token })
  }
}

// 3. Input validation and sanitization
import { z } from 'zod'
import DOMPurify from 'isomorphic-dompurify'
```

```
const userSchema = z.object({
  email: z.string().email(),
  name: z.string().min(2).max(50),
  bio: z.string().max(500).transform(val => DOMPurify.sanitize(val))
})

export default async function handler(req, res) {
  try {
    const validated = userSchema.parse(req.body)
    // Process validated data
  } catch (error) {
    res.status(400).json({ error: error.errors })
  }
}
```

44. How do you implement micro-frontends with Next.js?

Answer: Micro-frontend architecture:

```
jsx
```

```js
// 1. Module Federation approach
// next.config.js
const { NextFederationPlugin } = require('@module-federation/nextjs-mf')

module.exports = {
  webpack(config, options) {
    config.plugins.push(
      new NextFederationPlugin({
        name: 'host',
        remotes: {
          shop: `shop@http://localhost:3001/_next/static/chunks/remoteEntry.js`,
          checkout: `checkout@http://localhost:3002/_next/static/chunks/remoteEntry.js
        },
        filename: 'static/chunks/remoteEntry.js',
        exposes: {
          './nav': './components/Navigation',
        },
        shared: {
          react: { singleton: true, requiredVersion: false },
          'react-dom': { singleton: true, requiredVersion: false },
        },
      })
    )
    return config
  },
}

// 2. Zone-based approach
// next.config.js
module.exports = {
  async rewrites() {
    return [
      {
        source: '/shop/:path*',
        destination: 'http://localhost:3001/shop/:path*',
      },
      {
        source: '/checkout/:path*',
        destination: 'http://localhost:3002/checkout/:path*',
      },
    ]
  },
}

// 3. Runtime integration
import dynamic from 'next/dynamic'
```

```
const RemoteComponent = dynamic(
  () => import('shop/ProductList').catch(() => {
    return () => <div>Failed to load remote component</div>
  }),
  { ssr: false }
)
```

45. How do you optimize Next.js for Core Web Vitals?

Answer: Core Web Vitals optimization:

```
jsx
```

```
// 1. Largest Contentful Paint (LCP)
import Image from 'next/image'

export default function Hero() {
  return (
    <Image
      src="/hero.jpg"
      alt="Hero"
      width={1200}
      height={600}
      priority // Load with high priority
      placeholder="blur"
      blurDataURL={blurDataUrl}
      sizes="(max-width: 768px) 100vw, 1200px"
    />
  )
}

// 2. First Input Delay (FID) / Interaction to Next Paint (INP)
// Code splitting for better interactivity
const HeavyComponent = dynamic(() => import('./HeavyComponent'), {
  loading: () => <Skeleton />,
  ssr: false
})

// 3. Cumulative Layout Shift (CLS)
// Reserve space for dynamic content
<div style={{ minHeight: '500px' }}>
  <Suspense fallback={<Skeleton height={500} />}>
    <DynamicContent />
  </Suspense>
</div>

// 4. Monitoring
export function reportWebVitals(metric) {
  if (metric.label === 'web-vital') {
    console.log(metric) // Send to analytics

    // Send to analytics endpoint
    fetch('/api/vitals', {
      method: 'POST',
      body: JSON.stringify(metric),
    })
  }
}
```

46. How do you implement advanced state management patterns?

Answer: Complex state management solutions:

jsx

```
// 1. Zustand with SSR
import { create } from 'zustand'
import { devtools, persist } from 'zustand/middleware'

const useStore = create(
  devtools(
    persist(
      (set) => ({
        user: null,
        setUser: (user) => set({ user }),
        logout: () => set({ user: null }),
      }),
      {
        name: 'user-storage',
      }
    )
  )
)

// SSR-safe implementation
export default function Component() {
  const [mounted, setMounted] = useState(false)
  const user = useStore((state) => state.user)

  useEffect(() => {
    setMounted(true)
  }, [])

  if (!mounted) return null

  return <div>{user?.name}</div>
}

// 2. Server state with React Query/TanStack Query
import { QueryClient, dehydrate } from '@tanstack/react-query'

export async function getServerSideProps() {
  const queryClient = new QueryClient()

  await queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: fetchPosts,
  })

  return {
    props: {
        dehydratedState: dehydrate(queryClient),
      },
    }
  }

  // 3. Optimistic updates
  const mutation = useMutation({
    mutationFn: updatePost,
    onMutate: async (newPost) => {
      await queryClient.cancelQueries({ queryKey: ['posts'] })

      const previousPosts = queryClient.getQueryData(['posts'])

      queryClient.setQueryData(['posts'], (old) => [...old, newPost])

      return { previousPosts }
    },
    onError: (err, newPost, context) => {
      queryClient.setQueryData(['posts'], context.previousPosts)
    },
    onSettled: () => {
      queryClient.invalidateQueries({ queryKey: ['posts'] })
    },
  })
```

47. How do you implement complex routing patterns with intercepting routes?

Answer: Advanced routing with interception:

```jsx
// Intercepting routes structure
app/
  feed/
    page.js
  @modal/
    (.)photo/[id]/
      page.js
  photo/[id]/
    page.js
  layout.js

// app/layout.js
export default function Layout({ children, modal }) {
  return (
    <>
      {children}
      {modal}
    </>
  )
}

// app/@modal/(.)photo/[id]/page.js
import { Modal } from '@/components/Modal'

export default function PhotoModal({ params }) {
  return (
    <Modal>
      <Photo id={params.id} />
    </Modal>
  )
}

// Complex route matching
app/
  shop/
    [...categories]/
      page.js // Matches /shop/electronics/phones/samsung
  blog/
    [[...slug]]/
      page.js // Optional catch-all
```

48. How do you implement edge computing with Next.js?

Answer: Edge runtime implementation:

jsx

```javascript
// 1. Edge API Routes
// app/api/edge/route.js
export const runtime = 'edge'

export async function GET(request) {
  const { searchParams } = new URL(request.url)

  // Use Web APIs available at edge
  const response = await fetch('https://api.example.com/data', {
    headers: {
      'X-Custom-Header': searchParams.get('token'),
    },
  })

  // Stream response
  return new Response(response.body, {
    headers: {
      'Content-Type': 'application/json',
      'Cache-Control': 'public, s-maxage=60',
    },
  })
}

// 2. Edge Middleware with geolocation
export function middleware(request) {
  const country = request.geo?.country || 'US'
  const response = NextResponse.next()

  response.headers.set('X-User-Country', country)

  // Redirect based on location
  if (country === 'GB' && request.nextUrl.pathname === '/') {
    return NextResponse.redirect(new URL('/uk', request.url))
  }

  return response
}

// 3. Edge-rendered pages
export const runtime = 'edge'
export const preferredRegion = 'iad1' // Specific region

export default async function Page() {
  const data = await fetch('https://edge-api.example.com/data', {
    cache: 'no-store',
  })
```

```
    return <div>{/* Render data */}</div>
  }
```

49. How do you implement advanced testing strategies in Next.js?

Answer: Comprehensive testing approach:

jsx

```
// 1. Integration testing with Playwright
// e2e/app.spec.ts
import { test, expect } from '@playwright/test'

test.describe('App E2E Tests', () => {
  test('should navigate and interact', async ({ page }) => {
    await page.goto('/')

    // Test navigation
    await page.click('text=About')
    await expect(page).toHaveURL('/about')

    // Test form submission
    await page.fill('input[name="email"]', 'test@example.com')
    await page.click('button[type="submit"]')

    await expect(page.locator('.success')).toBeVisible()
  })
})

// 2. Component testing with React Testing Library
// __tests__/components/Form.test.tsx
import { render, screen, waitFor } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { rest } from 'msw'
import { setupServer } from 'msw/node'

const server = setupServer(
  rest.post('/api/submit', (req, res, ctx) => {
    return res(ctx.json({ success: true }))
  })
)

beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

test('submits form data', async () => {
  render(<Form />)

  const user = userEvent.setup()

  await user.type(screen.getByLabelText('Email'), 'test@example.com')
  await user.click(screen.getByRole('button', { name: 'Submit' }))

  await waitFor(() => {
    expect(screen.getByText('Success!')).toBeInTheDocument()
  })
})

// 3. API route testing
// __tests__/api/users.test.ts
import { createMocks } from 'node-mocks-http'
import handler from '@/pages/api/users'

describe('/api/users', () => {
  test('returns users list', async () => {
    const { req, res } = createMocks({
      method: 'GET',
    })

    await handler(req, res)

    expect(res._getStatusCode()).toBe(200)
    expect(JSON.parse(res._getData())).toEqual(
      expect.arrayContaining([
        expect.objectContaining({
          id: expect.any(Number),
          name: expect.any(String),
        })
      ])
    )
  })
})
```

50. How do you implement real-time features with Server-Sent Events?

Answer: SSE implementation for real-time updates:

jsx

```javascript
// app/api/events/route.js
export async function GET(request) {
  const encoder = new TextEncoder()

  const stream = new ReadableStream({
    async start(controller) {
      const sendEvent = (data) => {
        controller.enqueue(
          encoder.encode(`data: ${JSON.stringify(data)}\n\n`)
        )
      }

      // Send initial event
      sendEvent({ type: 'connected', timestamp: Date.now() })

      // Set up interval for periodic updates
      const interval = setInterval(() => {
        sendEvent({
          type: 'update',
          data: Math.random(),
          timestamp: Date.now(),
        })
      }, 1000)

      // Clean up on close
      request.signal.addEventListener('abort', () => {
        clearInterval(interval)
        controller.close()
      })
    },
  })

  return new Response(stream, {
    headers: {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      'Connection': 'keep-alive',
    },
  })
}

// Client component
'use client'

export default function RealTimeData() {
  const [data, setData] = useState([])
```

```
    useEffect(() => {
      const eventSource = new EventSource('/api/events')

      eventSource.onmessage = (event) => {
        const newData = JSON.parse(event.data)
        setData(prev => [...prev, newData].slice(-10)) // Keep last 10
      }

      eventSource.onerror = () => {
        console.error('SSE connection error')
        eventSource.close()
      }

      return () => eventSource.close()
    }, [])

    return (
      <div>
        {data.map((item, i) => (
          <div key={i}>{JSON.stringify(item)}</div>
        ))}
      </div>
    )
  }
```

51. How do you implement advanced image optimization strategies?

Answer: Comprehensive image optimization:

```
jsx
```

```
// 1. Responsive images with art direction
import Image from 'next/image'

export function ResponsiveHero() {
  return (
    <picture>
      <source
        media="(min-width: 1024px)"
        srcSet="/hero-desktop.webp"
      />
      <source
        media="(min-width: 768px)"
        srcSet="/hero-tablet.webp"
      />
      <Image
        src="/hero-mobile.webp"
        alt="Hero"
        width={768}
        height={400}
        sizes="(min-width: 1024px) 1200px, (min-width: 768px) 768px, 100vw"
        priority
      />
    </picture>
  )
}

// 2. Dynamic image optimization API
// app/api/optimize/route.js
import sharp from 'sharp'

export async function GET(request) {
  const { searchParams } = new URL(request.url)
  const url = searchParams.get('url')
  const width = parseInt(searchParams.get('w') || '800')
  const quality = parseInt(searchParams.get('q') || '80')

  const response = await fetch(url)
  const buffer = await response.arrayBuffer()

  const optimized = await sharp(buffer)
    .resize(width)
    .webp({ quality })
    .toBuffer()

  return new Response(optimized, {
    headers: {
        'Content-Type': 'image/webp',
        'Cache-Control': 'public, max-age=31536000',
      },
    })
  }

  // 3. Placeholder generation
  import { getPlaiceholder } from 'plaiceholder'

  export async function getStaticProps() {
    const { base64, img } = await getPlaiceholder('/hero.jpg')

    return {
      props: {
        imageProps: {
          ...img,
          blurDataURL: base64,
        },
      },
    }
  }
```

52. How do you implement complex data fetching patterns with React Suspense?

Answer: Advanced Suspense patterns:

jsx

```
// 1. Parallel data fetching with Suspense
// app/dashboard/page.js
import { Suspense } from 'react'

async function UserData() {
  const user = await fetchUser()
  return <div>{user.name}</div>
}

async function Analytics() {
  const data = await fetchAnalytics()
  return <div>{data.views} views</div>
}

async function Activity() {
  const activities = await fetchActivity()
  return <div>{activities.length} activities</div>
}

export default function Dashboard() {
  return (
    <div>
      <Suspense fallback={<div>Loading user...</div>}>
        <UserData />
      </Suspense>

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

      <Suspense fallback={<div>Loading activity...</div>}>
        <Activity />
      </Suspense>
    </div>
  )
}

// 2. Streaming with nested Suspense
export default function Page() {
  return (
    <Suspense fallback={<HeaderSkeleton />}>
      <Header />
      <Suspense fallback={<ContentSkeleton />}>
        <Content />
        <Suspense fallback={<CommentsSkeleton />}>
          <Comments />
        </Suspense>
      </Suspense>
    </Suspense>
  )
}

// 3. Error boundaries with Suspense
'use client'

import { ErrorBoundary } from 'react-error-boundary'

function ErrorFallback({ error, resetErrorBoundary }) {
  return (
    <div>
      <p>Something went wrong:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>Try again</button>
    </div>
  )
}

export default function SafeComponent() {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <Suspense fallback={<Loading />}>
        <DataComponent />
      </Suspense>
    </ErrorBoundary>
  )
}
```

53. How do you implement advanced monitoring and observability?

Answer: Comprehensive monitoring setup:

```
jsx
```

```
// 1. Custom performance monitoring
// lib/monitoring.js
export class PerformanceMonitor {
  constructor() {
    this.metrics = new Map()
  }

  startTimer(label) {
    this.metrics.set(label, performance.now())
  }

  endTimer(label) {
    const start = this.metrics.get(label)
    if (start) {
      const duration = performance.now() - start
      this.sendMetric(label, duration)
      this.metrics.delete(label)
    }
  }

  sendMetric(label, value) {
    // Send to monitoring service
    fetch('/api/metrics', {
      method: 'POST',
      body: JSON.stringify({ label, value, timestamp: Date.now() }),
    })
  }
}

// 2. Error tracking with Sentry
// app/error.js
'use client'

import * as Sentry from '@sentry/nextjs'
import { useEffect } from 'react'

export default function Error({ error, reset }) {
  useEffect(() => {
    Sentry.captureException(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button onClick={reset}>Try again</button>
    </div>
  )
}

// 3. Custom instrumentation
// instrumentation.ts
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    const { PrometheusExporter } = await import('@opentelemetry/exporter-prometheus')
    const { MeterProvider } = await import('@opentelemetry/sdk-metrics')

    const exporter = new PrometheusExporter({ port: 9090 })
    const meterProvider = new MeterProvider()

    meterProvider.addMetricReader(exporter)

    // Create custom metrics
    const meter = meterProvider.getMeter('nextjs-app')
    const requestCounter = meter.createCounter('http_requests_total')

    // Track requests
    global.trackRequest = (method, path, status) => {
      requestCounter.add(1, { method, path, status })
    }
  }
}
```

54. How do you implement advanced deployment strategies?

Answer: Sophisticated deployment patterns:

```
yaml
```

```
# 1. Blue-Green Deployment with GitHub Actions
name: Blue-Green Deploy

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build and Deploy to Green
        run: |
          docker build -t myapp:${{ github.sha }} .
          docker tag myapp:${{ github.sha }} registry.example.com/myapp:green
          docker push registry.example.com/myapp:green

      - name: Health Check Green
        run: |
          for i in {1..30}; do
            if curl -f https://green.example.com/health; then
              echo "Green deployment healthy"
              break
            fi
            sleep 10
          done

      - name: Switch Traffic
        run: |
          # Update load balancer to point to green
          aws elbv2 modify-target-group \
            --target-group-arn ${{ secrets.TARGET_GROUP_ARN }} \
            --targets Id=green-instance

      - name: Tag Green as Blue
        run: |
          docker tag registry.example.com/myapp:green registry.example.com/myapp:blue

# 2. Canary Deployment
// middleware.js
export function middleware(request) {
  const canaryPercentage = 10
  const isCanary = Math.random() * 100 < canaryPercentage
  if (isCanary) {
    // Route to canary version
    request.headers.set('x-deployment-version', 'canary')
    return NextResponse.rewrite(
      new URL('/canary' + request.nextUrl.pathname, request.url)
    )
  }

  return NextResponse.next()
}

# 3. Feature Flag Deployment
// lib/features.js
import { createClient } from '@vercel/edge-config'

const edgeConfig = createClient(process.env.EDGE_CONFIG)

export async function isFeatureEnabled(feature, userId) {
  const flags = await edgeConfig.get('featureFlags')
  const flag = flags[feature]

  if (!flag) return false

  // Percentage rollout
  if (flag.percentage) {
    const hash = userId.split('').reduce((a, b) => {
      a = ((a << 5) - a) + b.charCodeAt(0)
      return a & a
    }, 0)

    return Math.abs(hash) % 100 < flag.percentage
  }

  // User list
  if (flag.users) {
    return flag.users.includes(userId)
  }

  return flag.enabled
}
```

55. How do you implement advanced API patterns like GraphQL?

Answer: GraphQL integration with Next.js:

```
jsx
```

```
// 1. GraphQL API Route
// pages/api/graphql.js
import { ApolloServer } from '@apollo/server'
import { startServerAndCreateNextHandler } from '@as-integrations/next'
import { makeExecutableSchema } from '@graphql-tools/schema'
import { applyMiddleware } from 'graphql-middleware'
import { shield, rule } from 'graphql-shield'

const typeDefs = `
  type Query {
    posts(limit: Int, offset: Int): [Post!]!
    post(id: ID!): Post
    me: User
  }

  type Mutation {
    createPost(input: CreatePostInput!): Post!
    updatePost(id: ID!, input: UpdatePostInput!): Post!
  }

  type Post {
    id: ID!
    title: String!
    content: String!
    author: User!
    comments: [Comment!]!
  }

  type User {
    id: ID!
    email: String!
    posts: [Post!]!
  }
`

const resolvers = {
  Query: {
    posts: async (_, { limit = 10, offset = 0 }, { prisma }) => {
      return prisma.post.findMany({
        take: limit,
        skip: offset,
        include: { author: true },
      })
    },
    post: async (_, { id }, { prisma }) => {
      return prisma.post.findUnique({
        where: { id },
        include: { author: true, comments: true },
      })
    },
    me: async (_, __, { user, prisma }) => {
      if (!user) return null
      return prisma.user.findUnique({ where: { id: user.id } })
    },
  },
  Mutation: {
    createPost: async (_, { input }, { user, prisma }) => {
      return prisma.post.create({
        data: {
          ...input,
          authorId: user.id,
        },
      })
    },
  },
}

// Authorization rules
const isAuthenticated = rule()(async (parent, args, { user }) => {
  return user !== null
})

const permissions = shield({
  Query: {
    me: isAuthenticated,
  },
  Mutation: {
    createPost: isAuthenticated,
    updatePost: isAuthenticated,
  },
})

const schema = applyMiddleware(
  makeExecutableSchema({ typeDefs, resolvers }),
  permissions
)

const server = new ApolloServer({
  schema,
  introspection: process.env.NODE_ENV !== 'production',
})

export default startServerAndCreateNextHandler(server, {
  context: async (req) => {
    const user = await getUserFromToken(req.headers.authorization)
    return { user, prisma }
  },
})

// 2. Client-side GraphQL with Apollo
// lib/apollo-client.js
import { ApolloClient, InMemoryCache, createHttpLink } from '@apollo/client'
import { setContext } from '@apollo/client/link/context'

const httpLink = createHttpLink({
  uri: '/api/graphql',
})

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem('token')

  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : '',
    },
  }
})

export const apolloClient = new ApolloClient({
  link: authLink.concat(httpLink),
  cache: new InMemoryCache(),
  defaultOptions: {
    watchQuery: {
      fetchPolicy: 'cache-and-network',
    },
  },
})

// 3. SSR/SSG with GraphQL
export async function getStaticProps() {
  const { data } = await apolloClient.query({
    query: gql`
      query GetPosts {
        posts(limit: 10) {
          id
          title
          author {
            name
          }
          }
        }
      `,
    })

    return {
      props: {
        posts: data.posts,
      },
      revalidate: 60,
    }
  }
```

56. How do you implement database transactions and connection pooling?

Answer: Advanced database patterns:

```
jsx
```

```
// 1. Connection pooling with Prisma
// lib/prisma.js
import { PrismaClient } from '@prisma/client'

const globalForPrisma = global as unknown as {
  prisma: PrismaClient | undefined
}

export const prisma = globalForPrisma.prisma ??
  new PrismaClient({
    datasources: {
      db: {
        url: process.env.DATABASE_URL,
      },
    },
    log: process.env.NODE_ENV === 'development'
      ? ['query', 'error', 'warn']
      : ['error'],
    // Connection pool settings
    connectionLimit: 10,
  })

if (process.env.NODE_ENV !== 'production') {
  globalForPrisma.prisma = prisma
}

// 2. Database transactions
// pages/api/transfer.js
export default async function handler(req, res) {
  const { fromUserId, toUserId, amount } = req.body

  try {
    const result = await prisma.$transaction(async (tx) => {
      // Debit from sender
      const sender = await tx.user.update({
        where: { id: fromUserId },
        data: { balance: { decrement: amount } },
      })

      if (sender.balance < 0) {
        throw new Error('Insufficient funds')
      }

      // Credit to receiver
      const receiver = await tx.user.update({
        where: { id: toUserId },
        data: { balance: { increment: amount } },
      })

      // Create transaction record
      const transaction = await tx.transaction.create({
        data: {
          fromUserId,
          toUserId,
          amount,
          status: 'completed',
        },
      })

      return { sender, receiver, transaction }
    }, {
      maxWait: 5000, // 5 seconds max wait
      timeout: 10000, // 10 seconds timeout
      isolationLevel: 'Serializable',
    })

    res.status(200).json(result)
  } catch (error) {
    console.error('Transaction failed:', error)
    res.status(400).json({ error: error.message })
  }
}

// 3. Raw SQL with parameterized queries
export async function complexQuery(filters) {
  const { category, minPrice, maxPrice } = filters

  const products = await prisma.$queryRaw`
    SELECT
      p.*,
      c.name as category_name,
      COUNT(r.id) as review_count,
      AVG(r.rating) as avg_rating
    FROM products p
    LEFT JOIN categories c ON p.category_id = c.id
    LEFT JOIN reviews r ON p.id = r.product_id
    WHERE
      p.price BETWEEN ${minPrice} AND ${maxPrice}
      ${category ? Prisma.sql`AND c.slug = ${category}` : Prisma.empty}
    GROUP BY p.id, c.id
    ORDER BY avg_rating DESC NULLS LAST
    LIMIT 20
  `
```

```
    return products
  }
```

57. How do you implement advanced caching strategies with Redis?

Answer: Redis caching patterns:

```
jsx
```

```
// 1. Redis client setup with connection pooling
// lib/redis.js
import { createClient } from 'redis'

class RedisClient {
  constructor() {
    this.client = null
    this.isReady = false
  }

  async connect() {
    if (this.client) return this.client

    this.client = createClient({
      url: process.env.REDIS_URL,
      socket: {
        reconnectStrategy: (retries) => Math.min(retries * 50, 1000),
      },
    })

    this.client.on('error', (err) => console.error('Redis error:', err))
    this.client.on('ready', () => { this.isReady = true })

    await this.client.connect()
    return this.client
  }

  async get(key) {
    if (!this.isReady) await this.connect()
    return this.client.get(key)
  }

  async set(key, value, options = {}) {
    if (!this.isReady) await this.connect()
    return this.client.set(key, value, options)
  }

  async invalidate(pattern) {
    if (!this.isReady) await this.connect()
    const keys = await this.client.keys(pattern)
    if (keys.length) {
      await this.client.del(keys)
    }
  }
}
```

```
export const redis = new RedisClient()

// 2. Cache-aside pattern with stale-while-revalidate
export async function getCachedData(key, fetchFn, options = {}) {
  const { ttl = 3600, swr = 300 } = options

  try {
    // Try to get from cache
    const cached = await redis.get(key)

    if (cached) {
      const data = JSON.parse(cached)
      const age = Date.now() - data.timestamp

      // Return cached data if fresh
      if (age < ttl * 1000) {
        return data.value
      }

      // Stale-while-revalidate
      if (age < (ttl + swr) * 1000) {
        // Return stale data and refresh in background
        fetchFn().then(async (fresh) => {
          await redis.set(key, JSON.stringify({
            value: fresh,
            timestamp: Date.now(),
          }), { EX: ttl + swr })
        }).catch(console.error)

        return data.value
      }
    }

    // Fetch fresh data
    const fresh = await fetchFn()

    // Cache the result
    await redis.set(key, JSON.stringify({
      value: fresh,
      timestamp: Date.now(),
    }), { EX: ttl + swr })

    return fresh
  } catch (error) {
    console.error('Cache error:', error)
    // Fallback to direct fetch on cache error
    return fetchFn()
  }
}

// 3. Distributed locking for cache stampede prevention
export async function getWithLock(key, fetchFn, ttl = 3600) {
  const lockKey = `lock:${key}`
  const lockTTL = 30 // 30 seconds lock

  // Try to acquire lock
  const lockAcquired = await redis.set(lockKey, '1', {
    NX: true,
    EX: lockTTL,
  })

  if (!lockAcquired) {
    // Wait for other process to populate cache
    for (let i = 0; i < 10; i++) {
      await new Promise(resolve => setTimeout(resolve, 100))
      const cached = await redis.get(key)
      if (cached) return JSON.parse(cached)
    }

    throw new Error('Failed to get cached data')
  }

  try {
    // Fetch and cache data
    const data = await fetchFn()
    await redis.set(key, JSON.stringify(data), { EX: ttl })
    return data
  } finally {
    // Release lock
    await redis.del(lockKey)
  }
}
```

58. How do you implement multi-tenancy in Next.js?

Answer: Multi-tenant architecture:

jsx

```
// 1. Subdomain-based multi-tenancy
// middleware.js
export function middleware(request) {
  const hostname = request.headers.get('host')
  const subdomain = hostname.split('.')[0]

  // Skip for main domain
  if (subdomain === 'www' || subdomain === 'yourdomain') {
    return NextResponse.next()
  }

  // Rewrite to tenant-specific routes
  const url = request.nextUrl.clone()
  url.pathname = `/tenant/${subdomain}${url.pathname}`

  return NextResponse.rewrite(url)
}

// 2. Database isolation strategy
// lib/tenant.js
export async function getTenantConnection(tenantId) {
  const tenant = await prisma.tenant.findUnique({
    where: { id: tenantId },
    select: { databaseUrl: true },
  })

  if (!tenant) throw new Error('Tenant not found')

  // Create isolated Prisma client for tenant
  return new PrismaClient({
    datasources: {
      db: { url: tenant.databaseUrl },
    },
  })
}

// 3. Tenant context provider
// contexts/TenantContext.js
'use client'

import { createContext, useContext } from 'react'

const TenantContext = createContext()

export function TenantProvider({ tenant, children }) {
  return (
      <TenantContext.Provider value={tenant}>
        {children}
      </TenantContext.Provider>
    )
  }

  export function useTenant() {
    const context = useContext(TenantContext)
    if (!context) {
      throw new Error('useTenant must be used within TenantProvider')
    }
    return context
  }

  // 4. Tenant-aware API routes
  // pages/api/[tenant]/posts.js
  export default async function handler(req, res) {
    const { tenant: tenantSlug } = req.query

    // Get tenant configuration
    const tenant = await getTenantBySlug(tenantSlug)
    if (!tenant) {
      return res.status(404).json({ error: 'Tenant not found' })
    }

    // Use tenant-specific database
    const tenantDb = await getTenantConnection(tenant.id)

    try {
      const posts = await tenantDb.post.findMany({
        where: { tenantId: tenant.id },
      })

      res.status(200).json(posts)
    } finally {
      await tenantDb.$disconnect()
    }
  }
```

59. How do you implement complex authentication flows with OAuth and JWT?

Answer: Advanced authentication implementation:

```
jsx
```

```
// 1. Custom OAuth implementation
// pages/api/auth/oauth/[provider].js
import { OAuth2Client } from 'google-auth-library'
import jwt from 'jsonwebtoken'

const providers = {
  google: {
    clientId: process.env.GOOGLE_CLIENT_ID,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    redirectUri: `${process.env.NEXTAUTH_URL}/api/auth/callback/google`,
  },
  github: {
    clientId: process.env.GITHUB_CLIENT_ID,
    clientSecret: process.env.GITHUB_CLIENT_SECRET,
    redirectUri: `${process.env.NEXTAUTH_URL}/api/auth/callback/github`,
  },
}

export default async function handler(req, res) {
  const { provider } = req.query

  if (req.method === 'GET') {
    // Initiate OAuth flow
    const config = providers[provider]
    if (!config) {
      return res.status(404).json({ error: 'Provider not found' })
    }

    const state = generateState()
    await redis.set(`oauth:state:${state}`, provider, { EX: 600 })

    const authUrl = getAuthUrl(provider, state)
    res.redirect(authUrl)
  }
}

// 2. JWT with refresh tokens
// lib/auth.js
export function generateTokens(user) {
  const accessToken = jwt.sign(
    {
      id: user.id,
      email: user.email,
      role: user.role,
    },
    process.env.JWT_SECRET,
    {
      expiresIn: '15m',
      issuer: 'myapp',
      audience: 'myapp-api',
    }
  )

  const refreshToken = jwt.sign(
    {
      id: user.id,
      tokenFamily: crypto.randomUUID(),
    },
    process.env.JWT_REFRESH_SECRET,
    {
      expiresIn: '30d',
    }
  )

  return { accessToken, refreshToken }
}

// 3. Token refresh with rotation
// pages/api/auth/refresh.js
export default async function handler(req, res) {
  const { refreshToken } = req.body

  try {
    const decoded = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET)

    // Check if token is in blacklist (token rotation)
    const isBlacklisted = await redis.get(`blacklist:${refreshToken}`)
    if (isBlacklisted) {
      // Token reuse detected - invalidate all tokens in family
      await redis.set(`blacklist:family:${decoded.tokenFamily}`, '1', { EX: 86400 * 30
      throw new Error('Token reuse detected')
    }

    // Get user
    const user = await prisma.user.findUnique({
      where: { id: decoded.id },
    })

    if (!user) throw new Error('User not found')

    // Generate new tokens
    const tokens = generateTokens(user)
```

```
      // Blacklist old refresh token
      await redis.set(`blacklist:${refreshToken}`, '1', { EX: 86400 * 30 })

      // Store new refresh token
      await prisma.refreshToken.create({
        data: {
          token: tokens.refreshToken,
          userId: user.id,
          family: decoded.tokenFamily,
          expiresAt: new Date(Date.now() + 30 * 24 * 60 * 60 * 1000),
        },
      })

      res.status(200).json(tokens)
    } catch (error) {
      res.status(401).json({ error: 'Invalid refresh token' })
    }
  }

  // 4. Permission-based authorization
  // middleware/auth.js
  export function requirePermission(permission) {
    return async (req, res, next) => {
      try {
        const token = req.headers.authorization?.split(' ')[1]
        if (!token) throw new Error('No token provided')

        const decoded = jwt.verify(token, process.env.JWT_SECRET)

        // Get user with permissions
        const user = await prisma.user.findUnique({
          where: { id: decoded.id },
          include: {
            role: {
              include: {
                permissions: true,
              },
            },
          },
        })

        const hasPermission = user.role.permissions.some(p => p.name === permission)

        if (!hasPermission) {
          return res.status(403).json({ error: 'Insufficient permissions' })
        }
        req.user = user
        next()
      } catch (error) {
        res.status(401).json({ error: 'Authentication failed' })
      }
    }
  }
```

60. How do you optimize Next.js applications for large-scale production?

Answer: Production optimization strategies:

```
jsx
```

```
// 1. Advanced performance optimizations
// next.config.js
module.exports = {
  experimental: {
    optimizeCss: true,
    optimizePackageImports: ['lodash', 'date-fns'],
  },

  compiler: {
    removeConsole: process.env.NODE_ENV === 'production',
  },

  images: {
    formats: ['image/avif', 'image/webp'],
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },

  // Advanced caching headers
  async headers() {
    return [
      {
        source: '/_next/static/:path*',
        headers: [
          {
            key: 'Cache-Control',
            value: 'public, max-age=31536000, immutable',
          },
        ],
      },
      {
        source: '/api/:path*',
        headers: [
          {
            key: 'Cache-Control',
            value: 'public, s-maxage=60, stale-while-revalidate=300',
          },
        ],
      },
    ]
  },
}

// 2. Database query optimization
// lib/optimized-queries.js
export async function getOptimizedPosts(filters) {
  // Use database views for complex queries
  const posts = await prisma.$queryRaw`
    SELECT * FROM optimized_posts_view
    WHERE published = true
    ${filters.category ? Prisma.sql`AND category_id = ${filters.category}` : Prisma.em
    ORDER BY score DESC
    LIMIT 20
  `

  // Parallel fetch for related data
  const postIds = posts.map(p => p.id)

  const [authors, commentCounts] = await Promise.all([
    prisma.user.findMany({
      where: { id: { in: posts.map(p => p.authorId) } },
      select: { id: true, name: true, avatar: true },
    }),
    prisma.comment.groupBy({
      by: ['postId'],
      where: { postId: { in: postIds } },
      _count: true,
    }),
  ])

  // Efficient data mapping
  const authorMap = new Map(authors.map(a => [a.id, a]))
  const commentMap = new Map(commentCounts.map(c => [c.postId, c._count]))

  return posts.map(post => ({
    ...post,
    author: authorMap.get(post.authorId),
    commentCount: commentMap.get(post.id) || 0,
  }))
}

// 3. Resource optimization
// hooks/useOptimizedResources.js
export function useOptimizedResources() {
  useEffect(() => {
    // Preconnect to external domains
    const link = document.createElement('link')
    link.rel = 'preconnect'
    link.href = 'https://cdn.example.com'
    document.head.appendChild(link)

    // Lazy load non-critical resources
    if ('requestIdleCallback' in window) {
      requestIdleCallback(() => {
        import('../analytics').then(({ init }) => init())
        import('../chat-widget').then(({ init }) => init())
      })
    }

    // Progressive enhancement
    if ('IntersectionObserver' in window) {
      const images = document.querySelectorAll('img[data-src]')
      const imageObserver = new IntersectionObserver((entries) => {
        entries.forEach(entry => {
          if (entry.isIntersecting) {
            const img = entry.target
            img.src = img.dataset.src
            imageObserver.unobserve(img)
          }
        })
      })

      images.forEach(img => imageObserver.observe(img))
    }
  }, [])
}

// 4. Monitoring and alerting
// pages/api/health.js
export default async function handler(req, res) {
  const checks = {
    database: false,
    redis: false,
    storage: false,
  }

  // Parallel health checks
  await Promise.allSettled([
    prisma.$queryRaw`SELECT 1`.then(() => { checks.database = true }),
    redis.ping().then(() => { checks.redis = true }),
    checkStorage().then(() => { checks.storage = true }),
  ])

  const healthy = Object.values(checks).every(v => v)

  res.status(healthy ? 200 : 503).json({
    status: healthy ? 'healthy' : 'unhealthy',
    checks,
    timestamp: new Date().toISOString(),
    version: process.env.APP_VERSION,
  })
}
```