Next.js Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a custom Server-Side Rendering (SSR) cache control mechanism in Next.js

Solution:

Here's how to implement custom cache control for SSR pages:

```
export async function getServerSideProps({ req, res }) {
  res.setHeader('Cache-Control', 'public, s-maxage=10, stale-while-revalidate=59');
  const data = await fetchData();
  return { props: { data } };
}
```

Key points:

- s-maxage=10 sets CDN cache duration to 10 seconds
- stale-while-revalidate allows serving stale content while revalidating
- Useful for dynamic pages with semi-frequent updates

2. Create a custom middleware to handle authentication and role-based access control

Solution:

```
export function middleware(req) {
  const token = req.cookies.get('auth-token');
  if (!isValidToken(token)) {
    return NextResponse.redirect(new URL('/login', req.url));
  }
  return NextResponse.next();
}
```

Configuration:

- Place in middleware.ts at project root
- Handles authentication before route processing
- Supports path matching and conditional execution

3. Implement an optimized image loading strategy with next/image

Solution:

```
const ImageComponent = () => (

);
```

Optimization features:

- Automatic WebP conversion
- Responsive sizing
- Lazy loading with blur placeholder
- Priority loading for above-the-fold images

4. Create a custom error boundary with fallback UI in Next.js

Solution:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError(error) {
    return { hasError: true };
  }
  render() {
    return this.state.hasError ?  : this.props.children;
  }
}
```

Implementation details:

- Catches render-phase errors
- Prevents entire app crash
- Supports custom error reporting

5. Implement dynamic routing with catch-all routes and optional parameters

Solution:

```
// pages/posts/[...slug].js
export async function getStaticPaths() {
  return {
    paths: [{ params: { slug: ['2023', 'nextjs'] } }],
    fallback: 'blocking'
  };
}
```

Features:

- Handles nested dynamic routes
- Supports optional catch-all segments
- Enables incremental static regeneration

6. Create a custom hook for SWR data fetching with optimistic updates

Solution:

```
function useOptimisticData(key) {
  const { data, mutate } = useSWR(key, fetcher);
  const optimisticUpdate = async (newData) => {
    await mutate(newData, false);
    await sendToAPI(newData);
  };
  return { data, optimisticUpdate };
}
```

Benefits:

- Immediate UI updates
- Background revalidation
- Automatic error handling

7. Implement a custom Layout component with persistent navigation state

Solution:

```
const PersistentLayout = ({ children }) => {
  const [navState, setNavState] = useState(initialState);
  useEffect(() => {
    Router.events.on('routeChangeComplete', persistState);
    return () => Router.events.off('routeChangeComplete', persistState);
  }, []);
```

Features:

- Maintains state across route changes
- Handles navigation events
- Supports nested layouts

8. Create a custom API route with rate limiting and request validation

Solution:

```
export default async function handler(req, res) {
  const rateLimit = await checkRateLimit(req);
  if (!rateLimit.success) {
    return res.status(429).json({ error: 'Too many requests' });
  }
  // Handle request
}
```

Implementation:

- Redis-based rate limiting
- Request validation middleware
- Error handling patterns

9. Implement incremental static regeneration with on-demand revalidation

Solution:

```
export async function getStaticProps() {
  return {
    props: { data: await fetchData() },
    revalidate: 60,
    notFound: false
  };
}
```

Key features:

- Background regeneration
- On-demand invalidation
- Fallback handling

10. Create a custom performance monitoring solution using Web Vitals

Solution:

```
export function reportWebVitals(metric) {
  const { id, name, value } = metric;
  analytics.send({
    metric: name,
    value: Math.round(name === 'CLS' ? value * 1000 : value)
  });
}
```

Metrics tracked:

- First Contentful Paint (FCP)
- Largest Contentful Paint (LCP)
- Cumulative Layout Shift (CLS)

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU Cache in JavaScript with a specified capacity

Solution:

A Least Recently Used (LRU) cache can be implemented using a Map and maintaining capacity:

```
class LRUCache {
  constructor(capacity) {
    this.cache = new Map()
    this.capacity = capacity
  }
  get(key) {
    if (!this.cache.has(key)) return -1
    const val = this.cache.get(key)
    this.cache.delete(key)
    this.cache.set(key, val)
    return val
  }
  put(key, value) {
    this.cache.delete(key)
    if (this.cache.size === this.capacity) {
      this.cache.delete(this.cache.keys().next().value)
    }
    this.cache.set(key, value)
  }
}
```

Time Complexity: O(1) for both get and put operations

2. Implement a function that finds all pairs of integers in an array that sum to a target value

Solution:

Using a Set for O(n) time complexity:

```
function findPairs(arr, target) {
  const seen = new Set()
  const pairs = []
  for (const num of arr) {
    if (seen.has(target - num)) {
      pairs.push([num, target - num])
    }
    seen.add(num)
  }
  return pairs
}
```

Time Complexity: O(n) where n is the array length **Space Complexity:** O(n) for the Set

3. Implement a sliding window maximum algorithm for an array

Solution:

Using a deque approach for optimal performance:

```
function maxSlidingWindow(nums, k) {
  const result = []
  const deque = []
  for (let i = 0; i < nums.length; i++) {
    while (deque.length && nums[deque[deque.length-1]] <= nums[i]) {
      deque.pop()
    }
    deque.push(i)
    if (deque[0] <= i - k) deque.shift()
    if (i >= k - 1) result.push(nums[deque[0]])
  }
  return result
}
```

Time Complexity: O(n) where n is the array length

4. Implement a Stack data structure with O(1) access to the minimum element

Solution:

Using an auxiliary stack to track minimums:

```
class MinStack {
  constructor() {
    this.stack = []
    this.minStack = []
  }
  push(val) {
    this.stack.push(val)
    if (!this.minStack.length || val <= this.minStack[this.minStack.length-1]) {
      this.minStack.push(val)
    }
  }
  pop() {
    if (this.stack.pop() === this.minStack[this.minStack.length-1]) {
      this.minStack.pop()
    }
  }
  getMin() {
    return this.minStack[this.minStack.length-1]
  }
}
```

5. Implement a function to detect a cycle in a linked list

Solution:

Using Floyd's Cycle-Finding Algorithm (Tortoise and Hare):

```
function hasCycle(head) {
  let slow = head
  let fast = head
  while (fast && fast.next) {
    slow = slow.next
    fast = fast.next.next
    if (slow === fast) return true
  }
  return false
}
```

Time Complexity: O(n) **Space Complexity:** O(1)

6. Implement a Trie (Prefix Tree) data structure

Solution:

```
class TrieNode {
  constructor() {
    this.children = {}
    this.isEndOfWord = false
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode()
  }
  insert(word) {
    let node = this.root
    for (const char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode()
      }
      node = node.children[char]
    }
    node.isEndOfWord = true
  }
}
```

7. Implement a function to serialize and deserialize a binary tree

Solution:

```
function serialize(root) {
  if (!root) return 'null'
  return `${root.val},${serialize(root.left)},${serialize(root.right)}`
}

function deserialize(data) {
  const list = data.split(',')
  function dfs() {
    const val = list.shift()
    if (val === 'null') return null
    const node = new TreeNode(parseInt(val))
    node.left = dfs()
    node.right = dfs()
    return node
  }
  return dfs()
}
```

8. Implement a Queue using two Stacks

Solution:

```
class Queue {
  constructor() {
    this.stack1 = []
    this.stack2 = []
  }
  enqueue(val) {
    this.stack1.push(val)
  }
  dequeue() {
    if (!this.stack2.length) {
      while (this.stack1.length) {
        this.stack2.push(this.stack1.pop())
      }
    }
    return this.stack2.pop()
  }
}
```

9. Implement a function to find the k most frequent elements in an array

Solution:

Using a Map and sorting:

```
function topKFrequent(nums, k) {
  const freq = new Map()
  nums.forEach(n => freq.set(n, (freq.get(n) || 0) + 1))
  return Array.from(freq.entries())
    .sort((a, b) => b[1] - a[1])
    .slice(0, k)
    .map(entry => entry[0])
}
```

Time Complexity: O(n log n) **Space Complexity:** O(n)

10. Implement a function to perform binary search on a rotated sorted array

Solution:

```
function search(nums, target) {
  let left = 0, right = nums.length - 1
  while (left <= right) {
    const mid = Math.floor((left + right) / 2)
    if (nums[mid] === target) return mid
    if (nums[left] <= nums[mid]) {
      if (nums[left] <= target && target < nums[mid]) right = mid - 1
      else left = mid + 1
    } else {
      if (nums[mid] < target && target <= nums[right]) left = mid + 1
      else right = mid - 1
    }
  }
  return -1
}
```

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly using Next.js and supporting infrastructure

Key Components & Architecture:

- **Frontend:** Next.js app with server-side rendering for better SEO
- **Backend API:** Serverless functions via Next.js API routes
- **Database:** Distributed NoSQL (like DynamoDB) for URL mappings
- **Cache Layer:** Redis for frequently accessed URLs

Technical Implementation:

```
// URL shortening logic
const generateShortId = () => {
  return crypto.randomBytes(4).toString('base64')
    .replace(/[+/=]/g, '')
    .substring(0, 6);
}
```

Scalability Considerations:

- Use consistent hashing for database sharding
- Implement rate limiting on API routes
- CDN for static assets and cached redirects
- Analytics events stream to separate data warehouse

2. Design a real-time chat application using Next.js and WebSockets

Architecture Components:

- **WebSocket Server:** Socket.io with Next.js API routes
- **Message Queue:** Redis pub/sub for scaling
- **Database:** MongoDB for message persistence
- **State Management:** React Context + SWR for real-time updates

Implementation Example:

```
// WebSocket setup in Next.js API route
const io = new Server(res.socket.server);
io.on('connection', (socket) => {
  socket.on('message', async (msg) => {
    await saveToDb(msg);
    io.emit('newMessage', msg);
  });
});
```

Scaling Considerations:

- Sticky sessions for WebSocket connections
- Message deduplication
- Horizontal scaling with Redis pub/sub

3. Design a social media feed with infinite scroll and real-time updates

Core Components:

- **Frontend:** Next.js with ISR for feed pages
- **Data Fetching:** SWR + Cursor-based pagination
- **Push Updates:** Server-Sent Events or WebSocket
- **Cache Strategy:** Stale-while-revalidate

Feed Implementation:

```
// Infinite scroll hook
const useFeed = (pageSize = 10) => {
  const {data, setSize} = useSWRInfinite(
    (index) => `/api/feed?cursor=${index}&limit=${pageSize}`,
    fetcher
  );
};
```

Performance Optimizations:

- Content-based image optimization
- Virtual scrolling for large lists
- Selective real-time updates
- Edge caching for static content

4. Design a distributed job queue system for handling background tasks in Next.js

System Components:

- **Queue Service:** Bull with Redis backend
- **Workers:** Serverless functions or dedicated servers
- **Monitoring:** Bull Dashboard or custom metrics
- **Storage:** S3 for job artifacts

Queue Implementation:

```
// Job queue setup
const videoQueue = new Queue('video-processing', {
  redis: process.env.REDIS_URL,
  defaultJobOptions: { attempts: 3, backoff: { type: 'exponential' } }
});
```

Reliability Features:

- Dead letter queues
- Job retry policies
- Circuit breakers
- Distributed locks for unique jobs

5. Design a content management system (CMS) with Next.js and headless architecture

Architecture Overview:

- **Content API:** GraphQL endpoint with type safety
- **Preview Mode:** Next.js preview deployments
- **Asset Pipeline:** CDN + image optimization
- **Auth:** JWT + RBAC for content editors

Content Fetching:

```
// Static page generation with ISR
export async function getStaticProps({ params }) {
  const post = await cms.getPost(params.slug);
  return { props: { post }, revalidate: 60 };
}
```

Advanced Features:

- Content versioning
- Scheduled publishing

- Asset transformation pipeline
- Multi-language support

6. Design a real-time analytics dashboard with Next.js

System Components:

- **Event Collection:** Server-side tracking
- **Processing Pipeline:** Stream processing (Kafka)
- **Storage:** Time-series database (InfluxDB)
- **Visualization:** D3.js or Chart.js

Real-time Updates:

```
// WebSocket subscription for live metrics
const useMetrics = () => {
  const {data} = useSWR('/api/metrics', {
    refreshInterval: 1000,
    refreshWhenHidden: true
  });
```

Performance Considerations:

- Data aggregation strategies
- Time-window caching
- Downsampling for historical data
- Efficient client-side updates

7. Design a distributed caching system for Next.js applications

Cache Layers:

- **Browser Cache:** Service Worker + localStorage
- **CDN Cache:** Edge caching with Vercel
- **Application Cache:** Redis cluster
- **Database Cache:** Query result caching

Implementation:

```
// Multi-layer cache implementation
async function getData(key) {
  const cache = await caches.open('app-cache');
  return cache.match(key) || fetchAndCache(key);
}
```

Cache Strategies:

- Cache invalidation patterns
- Write-through vs Write-behind
- Cache coherence protocols
- Hot key handling

8. Design a search system with autocomplete for Next.js

Components:

- **Search Engine:** Elasticsearch or Algolia
- **Index Updates:** Real-time or batch indexing
- **Query Processing:** Query parsing and analysis
- **Results Ranking:** TF-IDF and custom scoring

Autocomplete Implementation:

```
// Debounced search with highlighting
const useSearch = (query) => {
  const {data} = useSWR(
    query ? `/api/search?q=${query}` : null,
    { dedupingInterval: 300 }
  );
```

Optimization Techniques:

- Prefix indexing
- Fuzzy matching
- Result caching
- Query suggestions

9. Design a multi-tenant architecture for Next.js applications

Key Components:

- **Tenant Isolation:** Separate databases/schemas
- **Routing:** Dynamic middleware for tenant resolution
- **Asset Storage:** Isolated S3 buckets
- **Authentication:** Tenant-specific JWT issuing

Middleware Example:

```
// Tenant resolution middleware
export function middleware(req) {
  const tenant = getTenantFromHostname(req.headers.host);
  return NextResponse.rewrite(new URL(`/${tenant}${req.url}`));
}
```

Security Considerations:

- Data isolation
- Resource quotas
- Rate limiting per tenant
- Audit logging

10. Design a deployment pipeline for Next.js applications with zero-downtime updates

Pipeline Components:

- **CI/CD:** GitHub Actions or Jenkins
- **Infrastructure:** Terraform + AWS/Vercel
- **Monitoring:** Prometheus + Grafana
- **Testing:** Jest + Cypress

Deployment Strategy:

```
// Health check endpoint
export default function handler(req, res) {
  const health = checkDependencies();
  res.status(health.ok ? 200 : 503).json(health);
}
```

Rollout Features:

- Blue-green deployment
- Canary releases
- Automated rollbacks
- Feature flags

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Implement a custom useLocalStorage hook in Next.js that handles JSON serialization and error cases

Solution:

```
const useLocalStorage = (key, initialValue) => {
  const [value, setValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      return initialValue;
    }
  });
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);
  return [value, setValue];
}
```

Key points:

- Handles JSON parsing errors gracefully
- Lazy initialization with useState callback
- Automatically serializes data on updates

2. Debug why getStaticProps is not updating data on production deployment

Common Issues and Solutions:

- Check if revalidate is properly set in getStaticProps
- Verify build output for static pages
- Ensure ISR is working correctly

```
export async function getStaticProps() {
  const data = await fetchData();
  return {
    props: { data },
    revalidate: 60 // Revalidate every minute
  };
}
```

Debugging steps:

- Check build logs for static generation
- Verify CDN cache headers
- Monitor revalidation triggers

3. Implement a custom middleware to handle API rate limiting in Next.js

Implementation:

```
export function middleware(req) {
  const key = req.ip;
  const limit = store.get(key) || 0;
  if (limit > 100) return new Response('Rate limit exceeded', {
    status: 429
  });
  store.set(key, limit + 1);
  return NextResponse.next();
}
```

Features:

- IP-based rate limiting
- Configurable limits
- Response headers for limit tracking

4. Create a custom error boundary component that handles client-side runtime errors

Solution:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false, error: null };
  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }
  render() {
    return this.state.hasError ?  : this.props.children;
  }
}
```

Important considerations:

- Only catches client-side errors
- Implement componentDidCatch for logging
- Reset error state on route changes

5. Optimize image loading performance using Next.js Image component and implement a custom loader

Custom Loader Implementation:

```
const customLoader = ({ src, width, quality }) => {
  return `https://cdn.example.com/${src}?w=${width}&q=${quality || 75}`;
}

const OptimizedImage = ({ src, ...props }) => (

);
```

Optimization techniques:

- Implement WebP support
- Configure proper sizing
- Use priority prop for LCP images

6. Implement a custom server-side caching mechanism for API routes

Implementation:

```
const cache = new Map();

export default async function handler(req, res) {
  const key = req.url;
  if (cache.has(key)) return res.json(cache.get(key));
  const data = await fetchData();
  cache.set(key, data);
  res.json(data);
}
```

Features:

- In-memory caching
- Configurable TTL
- Cache invalidation strategy

7. Debug and fix hydration errors in a Next.js application using dynamic imports

Solution:

```
const DynamicComponent = dynamic(() => import('./Component'), {
  ssr: false,
  loading: () =>
});

function Page() {
  return typeof window !== 'undefined' ?  : null;
}
```

Common issues:

- Mismatched server/client HTML
- useEffect dependencies
- Window object access

8. Implement a custom authentication middleware with role-based access control

Implementation:

```
export function withAuth(handler, roles = []) {
  return async (req, res) => {
    const token = req.headers.authorization;
    const user = await verifyToken(token);
    if (!roles.includes(user.role)) {
      return res.status(403).json({ error: 'Unauthorized' });
    }
    return handler(req, res);
  };
}
```

Features:

- JWT verification
- Role checking
- Error handling

9. Create a custom hook for handling infinite scroll with SSR support

Implementation:

```
const useInfiniteScroll = (fetchMore) => {
  const [loading, setLoading] = useState(false);
  useEffect(() => {
    const observer = new IntersectionObserver(entries => {
      if (entries[0].isIntersecting && !loading) {
        setLoading(true);
        fetchMore().finally(() => setLoading(false));
      }
    });
    observer.observe(document.querySelector('#sentinel'));
  }, []);
}
```

Features:

- Intersection Observer
- Loading states
- Error handling

10. Implement a custom page transition system with loading states

Solution:

```
const PageTransition = ({ children }) => {
  const router = useRouter();
  const [loading, setLoading] = useState(false);
  useEffect(() => {
    router.events.on('routeChangeStart', () => setLoading(true));
    router.events.on('routeChangeComplete', () => setLoading(false));
  }, []);
  return loading ?  : children;
}
```

Features:

- Route change detection
- Smooth transitions
- Loading indicators

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a challenging Next.js performance optimization problem you solved.

Situation: Our e-commerce app's product listing page had poor performance metrics with high LCP and FID scores.

Task: I needed to optimize the page to improve Core Web Vitals and reduce load times by 50%.

Action: I implemented several optimizations:

- Converted static product cards to use Image component with priority loading
- Implemented dynamic imports for non-critical components
- Added ISR with a 1-hour revalidation period
- Set up a CDN for static assets

Result: Page load time decreased by 65%, LCP improved from 4.2s to 1.8s, and conversion rate increased by 12%.

2. Describe a time when you had to refactor a large Next.js codebase.

Situation: Inherited a monolithic Next.js application with mixed server/client code and poor separation of concerns.

Task: Restructure the codebase to improve maintainability and reduce bundle size.

Action: I:

- Created clear server/client component boundaries
- Implemented proper data fetching patterns
- Set up module aliases for better imports
- Added strict TypeScript configurations

Result: Reduced bundle size by 40%, improved developer onboarding time from 2 weeks to 3 days, and decreased bug reports by 30%.

3. Tell me about a time you implemented a complex authentication system in Next.js.

Situation: Client needed a multi-tenant SaaS platform with SSO support.

Task: Implement secure authentication with multiple providers and role-based access.

Action: I:

- Integrated NextAuth.js with custom providers
- Implemented JWT with refresh token rotation
- Created middleware for route protection
- Set up role-based access control

Result: Successfully launched with zero security incidents, supporting 5000+ users across 50 organizations with 99.9% uptime.

4. Describe a situation where you had to debug a critical production issue in Next.js.

Situation: Production app experienced 500 errors during peak hours.

Task: Identify and fix the issue while minimizing downtime.

Action: I:

- Analyzed error logs and monitoring data
- Identified memory leaks in getStaticProps
- Implemented proper error boundaries
- Added comprehensive error logging

Result: Resolved issue within 2 hours, implemented preventive measures, and created a incident response playbook that reduced future MTTR by 60%.

5. Share an experience where you had to mentor junior developers in Next.js.

Situation: Team expanded with 3 junior developers new to Next.js.

Task: Get them productive with Next.js while maintaining code quality.

Action: I:

- Created a starter template with best practices
- Conducted weekly workshops on key concepts
- Implemented pair programming sessions
- Developed coding guidelines

Result: All juniors became independent contributors within 2 months, reducing PR review cycles by 40% and maintaining 90% test coverage.

6. Tell me about a time you had to optimize SEO for a Next.js application.

Situation: Content-heavy marketplace site had poor search rankings.

Task: Improve SEO metrics and search engine visibility.

Action: I:

- Implemented dynamic metadata generation
- Added structured data for products
- Set up proper canonical URLs
- Optimized for Core Web Vitals

Result: Organic traffic increased by 85% in 3 months, achieved 90+ Lighthouse SEO score, and improved SERP rankings for key terms by 40%.

7. Describe a situation where you had to integrate complex third-party services with Next.js.

Situation: Needed to integrate payment processing and analytics services.

Task: Implement secure, performant integrations without compromising UX.

Action: I:

- Created abstraction layers for services
- Implemented proper error handling
- Set up retry mechanisms
- Added monitoring and logging

Result: Achieved 99.9% payment processing success rate, reduced integration-related issues by 70%, and maintained sub-2-second page loads.

8. Share an experience with implementing internationalization in Next.js.

Situation: Company expanded to 5 new markets requiring localization.

Task: Implement full i18n support with minimal performance impact.

Action: I:

- Set up next-i18next with SSR support
- Implemented language detection
- Created translation management system
- Optimized bundle sizes per locale

Result: Successfully launched in all markets, maintained performance metrics, and achieved 98% translation coverage with automated workflows.

9. Tell me about a time you had to scale a Next.js application.

Situation: E-commerce platform experiencing performance issues at 100k daily users.

Task: Scale infrastructure to handle 5x growth.

Action: I:

- Implemented distributed caching
- Set up edge functions for API routes
- Optimized database queries
- Added load balancing

Result: Successfully handled Black Friday traffic of 500k users with 99.99% uptime and average response time under 200ms.

10. Describe a situation where you had to improve the testing strategy for a Next.js project.

Situation: Project had low test coverage and frequent regressions.

Task: Implement comprehensive testing strategy.

Action: I:

- Set up Jest and React Testing Library
- Implemented E2E tests with Cypress
- Added API integration tests
- Created CI/CD pipeline

Result: Achieved 90% test coverage, reduced production bugs by 75%, and decreased deployment rollbacks to near zero.