Interview Questions and Answers - Full Stack Development

React & Next.js Questions

"What's the difference between server-side rendering and client-side rendering?"

Answer:

Client-Side Rendering (CSR):

- Initial HTML is minimal, JavaScript loads and renders content in browser
- Process: Browser loads HTML → Downloads JS → Executes JS → Renders content
- Pros: Rich interactions, less server load
- Cons: Slower initial load, poor SEO, performance issues on low-end devices

Server-Side Rendering (SSR):

- Server generates complete HTML for each request
- Process: Server processes request → Renders HTML → Sends complete page to browser
- Pros: Faster initial load, better SEO, works without JavaScript
- Cons: More server resources, slower page transitions, full page reloads

"How would you optimize the performance of a React application?"

Answer:

1. **Code splitting**: Use `React.lazy()` and `Suspense` to load components only when needed
2. **Memoization**: Use `React.memo`, `useMemo`, and `useCallback` to prevent unnecessary re-renders
3. **Virtualization**: Implement windowing for long lists using libraries like `react-window`
4. **Image optimization**: Lazy load images, use proper formats (WebP), and responsive sizes
5. **State management optimization**: Keep state as local as possible, avoid unnecessary global state
6. **Bundle optimization**: Remove unused dependencies, use tree shaking, configure proper code splitting
7. **Use production builds**: Ensure minification and optimization flags are enabled
8. **Implement proper keys**: Use stable, unique keys for list items
9. **Avoid inline function definitions**: Define event handlers outside render method
10. **Use Chrome DevTools and React Profiler**: Identify and fix performance bottlenecks

"Explain the difference between useState and useReducer"

Answer:

useState:

- Simpler hook for managing single values or simple state
- Returns current state and setter function
- Best for independent pieces of state
- Example: const [count, setCount] = useState(0)

useReducer:

- More complex hook for managing related state transitions
- Returns current state and dispatch function
- Best for complex state logic with multiple sub-values or when next state depends on previous
- Uses reducer pattern similar to Redux
- Example:
- const [state, dispatch] = useReducer(reducer, initialState);dispatch({ type: 'INCREMENT' });

When to use which:

- Use useState for simple state
- Use useReducer when state logic is complex, involves multiple sub-values, or when next state depends on previous

"When would you use Next.js instead of plain React?"

Answer:

I would choose Next.js over plain React when:

1. **SEO is important**: Next.js provides server-side rendering and static generation, improving SEO
2. **Performance is critical**: Automatic code splitting and server-side rendering improve load times
3. **Routing needs are complex**: Next.js has built-in file-based routing system
4. **API routes are needed**: Next.js allows creating API endpoints within the same project
5. **Static site generation benefits the project**: Pre-rendering pages at build time improves performance
6. **Image optimization is required**: Built-in Image component optimizes images automatically
7. **Full-stack capabilities are desired**: Can build both frontend and API in one project
8. **Incremental Static Regeneration**: Need to update static content without full rebuilds

MERN Stack Questions

"How does the MERN stack communicate between frontend and backend?"

Answer:

In the MERN stack, communication between frontend (React) and backend (Express/Node.js) typically occurs through:

1. RESTful API endpoints:

- Frontend makes HTTP requests (GET, POST, PUT, DELETE) to backend API endpoints
- Express handles these requests and performs operations on MongoDB
- Server responds with JSON data or status codes

2. Example flow:

```
// Frontend (React)
const fetchUsers = async () => {
  const response = await fetch('http://localhost:5000/api/users');
  const data = await response.json();
  setUsers(data);
};

// Backend (Express)
app.get('/api/users', async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

3. Common libraries used:

- Axios or fetch API for making requests
- Express for handling requests
- Middleware like CORS to handle cross-origin requests

"How would you handle user authentication in a MERN application?"

Answer:

For authentication in a MERN application, I would implement:

1. JWT (JSON Web Tokens) based authentication:

- User logs in with credentials
- Server validates and returns a JWT
- Client stores token (localStorage/HTTP-only cookie)
- Token sent with subsequent requests in Authorization header

2. Implementation steps:

- Create user model in MongoDB with hashed passwords
- Implement register/login endpoints in Express
- Use bcrypt for password hashing and comparison

- Generate JWT tokens upon successful authentication
- Create middleware to verify tokens on protected routes

3. Security measures:

- Store passwords with bcrypt hashing (never plain text)
- Use HTTP-only cookies for better security against XSS
- Implement token expiration and refresh tokens
- Add rate limiting to prevent brute force attacks

4. Code example:

```
// Login endpoint
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });

  if (!user || !await bcrypt.compare(password, user.password)) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {
    expiresIn: '1h'
  });

  res.json({ token });
});

// Auth middleware
const auth = (req, res, next) => {
  try {
    const token = req.header('Authorization').replace('Bearer ', '');
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.id;
    next();
  } catch (e) {
    res.status(401).json({ message: 'Please authenticate' });
  }
};
```

"Describe your process for creating a RESTful API with Express"

Answer:

My process for creating a RESTful API with Express includes:

1. Setup and structure:

- Initialize project with npm and install dependencies
- Create folder structure (routes, controllers, models, middleware)
- Configure Express with necessary middleware (json parsing, cors, etc.)

2. Define data models:

- Create MongoDB schemas using Mongoose
- Define validation rules and relationships

3. Create routes and controllers:

- Separate route definitions from business logic
- Follow RESTful conventions for endpoints:
  - GET /resources (list)
  - POST /resources (create)
  - GET /resources/:id (read)
  - PUT/PATCH /resources/:id (update)
  - DELETE /resources/:id (delete)

4. Implement middleware:

- Authentication/authorization
- Input validation
- Error handling
- Logging

5. Error handling and validation:

- Create consistent error response format
- Validate request data using libraries like Joi or Express-validator
- Handle edge cases and exceptions

6. Testing and documentation:

- Write tests for API endpoints
- Document API using tools like Swagger/OpenAPI

7. Example code structure:

```
// models/user.js
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true }
});

// controllers/users.js
const getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};

// routes/users.js
router.get('/', userController.getUsers);
router.post('/', userController.createUser);
```

```
// app.js
app.use('/api/users', userRoutes);
```

TypeScript Questions

"What benefits does TypeScript add to a React project?"

Answer:

TypeScript provides several benefits to React projects:

1. **Type safety**: Catches type-related errors during development instead of runtime
2. **Better IDE support**: Enhanced autocomplete, inline documentation, and refactoring tools
3. **Self-documenting code**: Props and state are clearly defined with their types
4. **Improved team collaboration**: Types serve as contracts between components
5. **Safer refactoring**: The compiler catches breaking changes
6. **Better integration with libraries**: Type definitions for most popular libraries
7. **Clearer component interfaces**: Props and their requirements are explicitly defined
8. **Reduced runtime errors**: Many bugs are caught during compilation

"Explain how you'd type props for a React component"

Answer:

In TypeScript, I type React component props using interfaces or type aliases:

1. Using interface (most common):

```
interface UserCardProps {
  name: string;
  age: number;
  email?: string;  // Optional prop
  onSelect: (id: number) => void;  // Function prop
  status: 'active' | 'inactive';  // Union type
}

const UserCard: React.FC<UserCardProps> = ({ name, age, email, onSelect, status }) => {
  return (
    <div onClick={() => onSelect(1)}>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      {email && <p>Email: {email}</p>}
      <span>Status: {status}</span>
    </div>
  );
};
```

2. Using type alias:

```
type ButtonProps = {
  text: string;
  variant?: 'primary' | 'secondary';
  onClick: () => void;
  disabled?: boolean;
};

function Button({ text, variant = 'primary', onClick, disabled }: ButtonProps) {
  return (
    <button
      className={variant}
      onClick={onClick}
      disabled={disabled}
    >
      {text}
    </button>
  );
}
```

3. For children props:

```
interface LayoutProps {
  children: React.ReactNode;
  sidebar?: React.ReactNode;
}
```

4. For generic components:

```
interface ListProps<T> {
  items: T[];
  renderItem: (item: T) => React.ReactNode;
}

function List<T>({ items, renderItem }: ListProps<T>) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{renderItem(item)}</li>
      ))}
    </ul>
  );
}
```

"What's the difference between interface and type?"

Answer:

While `interface` and `type` are similar in TypeScript, they have key differences:

1. Declaration merging:

- Interfaces can be extended by declaring them multiple times

- Types cannot be re-opened to add new properties

```
// Interface merging
interface User { name: string; }
interface User { age: number; }
// Becomes: interface User { name: string; age: number; }

// Type cannot be merged
type User = { name: string; };
// Error: type User = { age: number; };
```

2. Extends and implements:

- Interfaces can extend other interfaces with `extends`
- Classes can implement interfaces with `implements`
- Types can use intersection (&) for similar functionality

```
// Interface extending
interface Animal { name: string; }
interface Dog extends Animal { breed: string; }

// Type intersections
type Animal = { name: string; };
type Dog = Animal & { breed: string; };
```

3. Complex types:

- Types can use unions, mapped types, conditional types more easily
- Interfaces are limited to object-like structures

```
// Only possible with type
type Status = 'loading' | 'success' | 'error';
type Nullable<T> = T | null;
type ReadOnly<T> = { readonly [P in keyof T]: T[P] };
```

4. When to use which:

- Use `interface` for public API definitions, object shapes, and when you want to allow extension
- Use `type` for unions, primitives, tuples, complex mapped types, or when you need exact type constraints

System Design Questions

"How would you structure a MERN stack project?"

Answer:

For a well-organized MERN stack project, I would structure it as:

1. Project root:

- package.json (workspace configuration if using monorepo)
- README.md
- .gitignore, .env.example
- Docker files (if using containerization)

2. Client (Frontend):

```
/client
  /public           # Static files
  /src
    /components      # Reusable UI components
      /common       # Shared components like buttons, inputs
      /layout       # Layout components
      /features      # Feature-specific components
    /hooks          # Custom React hooks
    /pages          # Page components (for routing)
    /services       # API communication
    /utils          # Helper functions
    /context        # React context providers
    /types          # TypeScript type definitions
    /assets         # Images, fonts, etc.
    /styles         # Global styles
    App.tsx         # Main component
    index.tsx       # Entry point
```

3. Server (Backend):

```
/server
  /src
    /controllers     # Request handlers
    /models          # MongoDB schemas
    /routes          # API route definitions
    /middleware      # Custom middleware
    /utils          # Helper functions
    /config          # Configuration files
    /services        # Business logic
    /types          # TypeScript type definitions
    /validation      # Input validation schemas
    index.ts         # Entry point
  /tests            # Unit and integration tests
```

4. Additional considerations:

- Use environment variables for configuration
- Add proper logging
- Include documentation
- Set up CI/CD pipeline configuration
- Add testing framework setup

"How would you handle state management in a large application?"

Answer:

For state management in large applications, I would implement a multi-layered approach:

1. Local component state:

- Use useState or useReducer for component-specific state
- Keep state as close as possible to where it's used

2. Shared state with React Context:

- Create context providers for sharing state between related components
- Organize contexts by domain/feature
- Use context selectors to prevent unnecessary re-renders

```
const UserContext = createContext();

function UserProvider({ children }) {
  const [user, setUser] = useState(null);
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

3. Global state with Redux Toolkit:

- Implement Redux for truly global state
- Organize using Redux Toolkit's slice pattern
- Use selectors for efficient access
- Example:
- const cartSlice = createSlice({ name: 'cart', initialState: { items: [] }, reducers: {   addItem: (state, action) => {   state.items.push(action.payload);   } }});

4. Server state management:

- Use React Query or SWR for server data
- Handles caching, refetching, and synchronization

```
const { data, isLoading } = useQuery(
  ['products'],
  fetchProducts
);
```

5. State persistence:

- Local storage for persistent state across sessions
- Use middleware for syncing with storage

6. Optimize for performance:

- Memoize selectors with reselect
- Use React.memo for expensive components
- Implement virtualization for long lists

7. Organization strategies:

- Split state by domain/feature
- Keep related state together
- Document state shape and usage

"How would you ensure your application is secure?"

Answer:

To ensure application security in a MERN stack project, I would implement:

1. Authentication & Authorization:

- Use JWT with proper expiration and refresh tokens
- Implement role-based access control
- Store passwords using bcrypt with appropriate salt rounds
- Use HTTP-only cookies for tokens when possible

2. API Security:

- Validate all input on server-side (never trust client data)
- Implement rate limiting to prevent brute force attacks
- Use HTTPS for all communications
- Add CORS configuration to restrict origins

```
app.use(cors({ origin: 'https://myapp.com' }));
app.use(rateLimit({ windowMs: 15 * 60 * 1000, max: 100 }));
```

3. Database Security:

- Use parameterized queries to prevent injection attacks
- Implement least privilege access for database users
- Sanitize data before storing
- Regular database backups

4. Frontend Security:

- Prevent XSS by sanitizing user input
- Use Content Security Policy headers
- Implement protection against CSRF attacks
- Avoid exposing sensitive information in client code

5. Environment & Deployment Security:

- Use environment variables for secrets (never commit them)
- Implement proper error handling (no sensitive info in errors)
- Regular dependency updates to patch vulnerabilities
- Use security headers (Helmet.js in Express)

```
app.use(helmet());
```

6. Monitoring & Maintenance:

- Implement logging for security events
- Regular security audits
- Use tools like Snyk or npm audit to check dependencies
- Keep frameworks and libraries updated

7. Coding Practices:

- Follow OWASP guidelines
- Code reviews with security focus
- Never store sensitive data in localStorage (use HTTP-only cookies)
- Implement proper session management

By combining these approaches, I create multiple layers of security that protect both the application and its users.