Micro-frontends 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 shared dependency loading strategy for micro-frontends.

Solution:

```
const sharedDeps = {
  react: { singleton: true, requiredVersion: '^17.0.0' },
  'react-dom': { singleton: true, requiredVersion: '^17.0.0' }
};

module.exports = {
  shared: sharedDeps
}
```

2. Design a micro-frontend asset preloading strategy for performance optimization.

Implementation:

```
const preloadMfe = (name) => {
  const link = document.createElement('link');
  link.rel = 'modulepreload';
  link.href = `/mfe/${name}/remoteEntry.js`;
  document.head.appendChild(link);
  return import(`/mfe/${name}/remoteEntry.js`);
}
```

3. Implement a module federation configuration for a host application that loads a remote micro-frontend component.

Solution:

Here's a webpack configuration for module federation:

```
const ModuleFederationPlugin = require('webpack').container.ModuleFederationPlugin;

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        mfe1: 'mfe1@http://localhost:3001/remoteEntry.js'
      },
      shared: ['react', 'react-dom']
    })
  ]
}
```

4. Create a micro-frontend router that handles route-based component loading across different applications.

Implementation:

```
class MicroRouter {
  constructor() {
    this.routes = new Map();
    window.addEventListener('popstate', this.handleRoute.bind(this));
  }
  register(path, loadMfe) {
    this.routes.set(path, loadMfe);
  }
  async handleRoute() {
    const mfe = this.routes.get(window.location.pathname);
    await mfe?.();
  }
}
```

5. Implement a shared state management solution for micro-frontends using custom events.

Solution:

```
class SharedState {
  constructor() {
    this.state = {};
    this.channel = new BroadcastChannel('mfe-state');
    this.channel.onmessage = ({data}) => this.handleStateChange(data);
  }
  setState(update) {
    this.state = {...this.state, ...update};
    this.channel.postMessage(this.state);
  }
}
```

6. Design a communication bridge between micro-frontends using MessageChannel API.

Implementation:

```
class MfeBridge {
  static connect(mfeA, mfeB) {
    const channel = new MessageChannel();
    mfeA.postMessage('connect', '*', [channel.port1]);
    mfeB.postMessage('connect', '*', [channel.port2]);
    return channel;
  }
}
```

Key features:

- Secure communication channel between micro-frontends
- Isolated message passing
- Cross-origin support

7. Create a loading boundary component for micro-frontend error handling and fallbacks.

Solution:

```
const MfeErrorBoundary = ({ fallback, children }) => {
  const [hasError, setError] = useState(false);
  useEffect(() => {
    window.addEventListener('mfe-error', () => setError(true));
    return () => window.removeEventListener('mfe-error', () => setError(true));
  }, []);
  return hasError ? fallback : children;
}
```

8. Implement a versioning strategy for micro-frontend deployments using semantic versioning.

Implementation:

```
const loadMfeVersion = async (name, version) => {
  const manifest = await fetch(`/mfe-manifest.json`);
  const { url } = manifest[name][version];
  return import(/* webpackIgnore: true */ url);
}
```

Best practices:

- Version tracking in manifest
- Backward compatibility checks
- Graceful fallbacks

9. Design a shared authentication solution for micro-frontends using JWT tokens.

Solution:

```
class AuthService {
  static setToken(token) {
    localStorage.setItem('mfe-token', token);
    window.dispatchEvent(new CustomEvent('auth-change'));
  }
  static getToken() {
    return localStorage.getItem('mfe-token');
  }
}
```

10. Create a style isolation mechanism for micro-frontends using Shadow DOM.

Implementation:

```
class IsolatedStyles extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({mode: 'closed'});
    const style = document.createElement('style');
    style.textContent = ':host { display: block; }';
    this.shadowRoot.appendChild(style);
  }
}
```

Data Structures and Algorithms

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

1. How would you implement a shared state management system across multiple micro-frontends?

Solution Approaches:

- **Custom Event Bus**: Implement a pub/sub pattern using browser's CustomEvents
- **Redux with Federation**: Share Redux store across MFEs
- **Observable Pattern**: Use RxJS for state synchronization

```
// Custom Event Bus Implementation
const eventBus = {
  emit: (event, data) => window.dispatchEvent(
    new CustomEvent(event, { detail: data })),
  on: (event, callback) => window.addEventListener(
    event, (e) => callback(e.detail))
};
```

2. Design a caching strategy for micro-frontend assets to optimize loading performance

Implementation Strategy:

- **Service Worker Cache**: For static assets
- **Memory Cache**: For runtime dependencies
- **Version-based invalidation**: For updates

```
const cacheStrategy = {
  version: 'v1',
  async cacheAssets(assets) {
    const cache = await caches.open(this.version);
    return cache.addAll(assets);
  },
  async fetch(request) {
    return await caches.match(request) || fetch(request);
  }
};
```

3. Implement a module federation plugin configuration for sharing common dependencies

Webpack Configuration:

```
const ModuleFederationPlugin = require('webpack').container.ModuleFederationPlugin;

module.exports = {
  plugins: [new ModuleFederationPlugin({
    name: 'host',
    shared: { react: { singleton: true, eager: true } }
  })]
};
```

- **Singleton**: Ensures single instance
- **Eager**: Loads dependency at startup

4. Design a routing system that works across multiple micro-frontends

Implementation Approach:

- **Event-based communication**
- **History API integration**
- **Route synchronization**

```
class MicroFrontendRouter {
  constructor(baseRoute) {
    this.baseRoute = baseRoute;
    window.addEventListener('popstate', this.handleRoute);
  }
  navigate(path) {
    history.pushState(null, '', `${this.baseRoute}${path}`);
  }
};
```

5. Implement a load balancing algorithm for micro-frontend deployment

Round-Robin Implementation:

```
class LoadBalancer {
  constructor(instances) {
    this.instances = instances;
    this.currentIndex = 0;
  }
  getNext() {
    const instance = this.instances[this.currentIndex];
    this.currentIndex = (this.currentIndex + 1) % this.instances.length;
    return instance;
  }
};
```

6. Design a communication bridge between micro-frontends using PostMessage API

Secure Communication Implementation:

```
class MicroFrontendBridge {
  static sendMessage(target, message) {
    target.postMessage({ data: message, source: 'mfe' }, '*');
  }
  static listen(callback) {
    window.addEventListener('message',
      (event) => event.data.source === 'mfe' && callback(event.data));
  }
};
```

7. Implement a dependency injection container for micro-frontend services

DI Container Implementation:

```
class DIContainer {
  constructor() {
    this.services = new Map();
  }
  register(key, implementation) {
    this.services.set(key, implementation);
  }
  resolve(key) {
    return this.services.get(key);
  }
};
```

8. Design a versioning system for micro-frontend deployments

Semantic Versioning Implementation:

```
class MFEVersion {
  static compare(v1, v2) {
    const [major1, minor1] = v1.split('.');
    const [major2, minor2] = v2.split('.');
    return major1 === major2 ? minor1 - minor2 : major1 - major2;
  }
};
```

9. Implement a circuit breaker pattern for micro-frontend API calls

Circuit Breaker Implementation:

```
class CircuitBreaker {
  constructor(failureThreshold = 5) {
    this.failures = 0;
    this.threshold = failureThreshold;
    this.state = 'CLOSED';
  }
  async call(fn) {
    if (this.state === 'OPEN') throw new Error('Circuit breaker open');
    try { return await fn(); }
    catch (e) { this.failures++; throw e; }
  }
};
```

10. Design a shared authentication system for micro-frontends

Token-based Auth Implementation:

```
class AuthService {
  static setToken(token) {
    localStorage.setItem('auth_token', token);
    window.dispatchEvent(new CustomEvent('auth_changed'));
  }
  static getToken() {
    return localStorage.getItem('auth_token');
  }
};
```

Coding and Debugging

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

1. How would you implement a basic module federation setup for a micro-frontend architecture?

Solution:

Here's a basic webpack configuration for module federation:

```
const ModuleFederationPlugin = require('webpack').container.ModuleFederationPlugin;

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      filename: 'remoteEntry.js',
      remotes: { mfe1: 'mfe1@http://localhost:3001/remoteEntry.js' },
      shared: ['react', 'react-dom']
    })
  ]
}
```

2. Create a function to handle cross-micro-frontend communication using a pub/sub pattern.

Implementation:

```
class EventBus {
  constructor() {
    this.events = new Map();
  }
  subscribe(event, callback) {
    if (!this.events.has(event)) this.events.set(event, []);
    this.events.get(event).push(callback);
  }
  publish(event, data) {
    if (this.events.has(event)) {
      this.events.get(event).forEach(cb => cb(data));
    }
  }
}
```

3. How would you implement a shared authentication state across multiple micro-frontends?

Solution:

```
// In a shared auth-store.js
class AuthStore {
  constructor() {
    this.token = localStorage.getItem('auth_token');
    this.subscribers = new Set();
  }
  setToken(token) {
    this.token = token;
    localStorage.setItem('auth_token', token);
    this.notify();
  }
  subscribe(callback) { this.subscribers.add(callback); }
}
```

4. Write a function to handle versioning conflicts between shared dependencies in micro-frontends.

Implementation:

```
function checkDependencyConflicts(apps) {
  const deps = new Map();
  return apps.reduce((conflicts, app) => {
    Object.entries(app.dependencies).forEach(([pkg, version]) => {
      if (deps.has(pkg) && deps.get(pkg) !== version) {
        conflicts.push(`${pkg}: ${deps.get(pkg)} vs ${version}`);
      }
      deps.set(pkg, version);
    });
    return conflicts;
  }, []);
}
```

5. Implement a loading state handler for asynchronously loaded micro-frontends.

Solution:

```
function MicroFrontendLoader({ name, url }) {
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    loadComponent(url)
      .then(() => setLoading(false))
      .catch(err => setError(err));
  }, [url]);

  return loading ?  : error ?  :
;
}
```

6. Create a function to handle style isolation between micro-frontends using Shadow DOM.

Implementation:

```
function createIsolatedStyles(elementId, styles) {
  const host = document.getElementById(elementId);
  const shadow = host.attachShadow({ mode: 'closed' });
  const styleSheet = document.createElement('style');
  styleSheet.textContent = styles;
  shadow.appendChild(styleSheet);
  return shadow;
}
```

7. Implement a shared state management solution for micro-frontends without using external libraries.

Solution:

```
class SharedState {
  constructor(initialState = {}) {
    this.state = initialState;
    this.listeners = new Set();
  }
  setState(newState) {
    this.state = { ...this.state, ...newState };
    this.listeners.forEach(listener => listener(this.state));
  }
  subscribe(listener) { this.listeners.add(listener); }
}
```

8. Write a function to handle routing across multiple micro-frontends.

Implementation:

```
class MicroFrontendRouter {
  constructor(routes) {
    this.routes = routes;
    window.addEventListener('popstate', this.handleRoute.bind(this));
  }
  handleRoute() {
    const path = window.location.pathname;
    const app = this.routes.find(r => path.startsWith(r.path));
    return app ? this.loadMicroFrontend(app) : this.show404();
  }
}
```

9. Implement a error boundary component for micro-frontends.

Solution:

```
class MicroFrontendErrorBoundary extends React.Component {
  state = { hasError: false, error: null };
  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }
  componentDidCatch(error, info) {
    console.error('Micro-frontend error:', error, info);
  }
  render() {
    return this.state.hasError ?  : this.props.children;
  }
}
```

10. Create a function to handle asset preloading for micro-frontends.

Implementation:

```
function preloadMicroFrontend(config) {
  const { name, entry, assets } = config;
  return Promise.all([
    import(entry),
    ...assets.map(asset => {
      const link = document.createElement('link');
      link.rel = 'prefetch';
      link.href = asset;
      document.head.appendChild(link);
    })
  ]);
}
```