Context Patterns for Architectural Dependencies

React Context extends far beyond simple data passing—it serves as a powerful architectural tool for implementing dependency injection patterns that enhance application structure, testability, and maintainability. Context-based dependency injection eliminates prop drilling, simplifies component testing, and establishes clear separation between business logic and presentation concerns.

Dependency injection is a design pattern where objects receive their dependencies from external sources rather than creating them internally. In React applications, this pattern prevents prop drilling complications, simplifies testing scenarios, and creates clear architectural boundaries between different application concerns.

Context vs. Prop Drilling Trade-offs

Context excels at resolving the "prop drilling" problem where props must traverse multiple component levels to reach deeply nested children. However, Context requires judicious application—not every shared state warrants Context usage. Consider Context when you have genuinely application-wide concerns or when prop drilling becomes architecturally unwieldy.

Traditional Dependency Injection with Context

Consider how a music practice application might inject various services throughout the component tree:

```
// Traditional prop drilling approach (becomes unwieldy)
function App() {
  const apiService = new PracticeAPIService();
  const analyticsService = new AnalyticsService();
  const storageService = new StorageService();

  return (
    <Dashboard
      apiService={apiService}
      analyticsService={analyticsService}
      storageService={storageService}
    />
  );
}

function Dashboard({ apiService, analyticsService, storageService }) {
  return (
    <div>
      <PracticeHistory
        apiService={apiService}
        analyticsService={analyticsService}
      />
      <SessionPlayer
        apiService={apiService}
        storageService={storageService}
      />
    </div>
  );
}

// Context-based dependency injection (cleaner)
const ServicesContext = createContext();

function App() {
  const services = {
    api: new PracticeAPIService(),
    analytics: new AnalyticsService(),
    storage: new StorageService(),
    notifications: new NotificationService()
  };

  return (
    <ServicesProvider services={services}>
      <Dashboard />
    </ServicesProvider>
  );
}

function useServices() {
  const context = useContext(ServicesContext);
  if (!context) {
    throw new Error('useServices must be used within a ServicesProvider');
  }
  return context;
}

// Components can now access services directly
function PracticeHistory() {
  const { api, analytics } = useServices();
  // Use services without prop drilling
}
```

Service Container Implementation with Context

A service container functions as a centralized registry that manages the creation and lifecycle of application services. This pattern proves particularly valuable for managing API clients, analytics services, storage adapters, and other cross-cutting architectural concerns.

```
import React, { createContext, useContext, useMemo } from 'react';

// Define service interfaces for better type safety
class PracticeAPIService {
  constructor(baseURL, authToken) {
    this.baseURL = baseURL;
    this.authToken = authToken;
  }

  async getSessions(userId) {
    // API implementation
  }

  async createSession(sessionData) {
    // API implementation
  }
}

class AnalyticsService {
  constructor(trackingId) {
    this.trackingId = trackingId;
  }

  track(event, properties) {
    // Analytics implementation
  }
}

class StorageService {
  setItem(key, value) {
    localStorage.setItem(key, JSON.stringify(value));
  }

  getItem(key) {
    const item = localStorage.getItem(key);
    return item ? JSON.parse(item) : null;
  }
}

class NotificationService {
  show(message, type = 'info') {
    // Notification implementation
  }
}
```

```

// Service container context
const ServiceContext = createContext();

export function ServiceProvider({ children, config = {} }) {
  const services = useMemo(() => {
    const api = new PracticeAPIService(
      config.apiBaseURL || '/api',
      config.authToken
    );

    const analytics = new AnalyticsService(
      config.analyticsTrackingId
    );

    const storage = new StorageService();

    const notifications = new NotificationService();

    return {
      api,
      analytics,
      storage,
      notifications
    };
  }, [config]);

  return (
    <ServiceContext.Provider value={services}>
      {children}
    </ServiceContext.Provider>
  );
}

export function useServices() {
  const context = useContext(ServiceContext);
  if (!context) {
    throw new Error('useServices must be used within a ServiceProvider');
  }
  return context;
}

// Individual service hooks for more granular access
export function useAPI() {
  return useServices().api;
}

export function useAnalytics() {
  return useServices().analytics;
}
```

```

export function useStorage() {
  return useServices().storage;
}

export function useNotifications() {
  return useServices().notifications;
}
```

Multi-Context State Management Architectures

Complex applications require multiple Context providers that collaborate to manage different aspects of application state and services effectively.

```
// User authentication context
const AuthContext = createContext();

function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const api = useAPI();

  useEffect(() => {
    api.getCurrentUser()
      .then(setUser)
      .catch(() => setUser(null))
      .finally(() => setLoading(false));
  }, [api]);

  const login = async (credentials) => {
    const user = await api.login(credentials);
    setUser(user);
    return user;
  };

  const logout = async () => {
    await api.logout();
    setUser(null);
  };

  const value = {
    user,
    loading,
    login,
    logout,
    isAuthenticated: !!user
  };

  return (
    <AuthContext.Provider value={value}>
      {children}
    </AuthContext.Provider>
  );
}

function useAuth() {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth must be used within an AuthProvider');
  }
  return context;
}

// Practice data context that depends on auth
const PracticeDataContext = createContext();

function PracticeDataProvider({ children }) {
  const [sessions, setSessions] = useState([]);
  const [loading, setLoading] = useState(false);
  const { user } = useAuth();
  const api = useAPI();

  useEffect(() => {
    if (user) {
      setLoading(true);
      api.getSessions(user.id)
        .then(setSessions)
        .finally(() => setLoading(false));
    } else {
      setSessions([]);
    }
  }, [user, api]);

  const createSession = async (sessionData) => {
    const newSession = await api.createSession({
      ...sessionData,
      userId: user.id
    });
    setSessions(prev => [newSession, ...prev]);
    return newSession;
  };

  const value = {
    sessions,
    loading,
    createSession
  };

  return (
    <PracticeDataContext.Provider value={value}>
      {children}
    </PracticeDataContext.Provider>
  );
}

function usePracticeData() {
  const context = useContext(PracticeDataContext);
  if (!context) {
    throw new Error('usePracticeData must be used within a PracticeDataProvider');
  }
  return context;
}

// App setup with multiple providers
function App() {
  return (
    <ServiceProvider config={{ apiBaseURL: '/api' }}>
      <AuthProvider>
        <PracticeDataProvider>
          <Dashboard />
        </PracticeDataProvider>
      </AuthProvider>
    </ServiceProvider>
  );
}
```

Hierarchical Provider Architecture

Complex applications benefit from hierarchical provider structures that enable granular control over dependencies and state scope. This architectural pattern allows different application sections to access distinct sets of services and state management.

```
// Base provider system with dependency resolution
function createProviderHierarchy() {
  const providers = new Map();

  const registerProvider = (name, Provider, dependencies = []) => {
    providers.set(name, { Provider, dependencies });
  };

  const buildProviderTree = (requestedProviders, children) => {
    // Resolve dependencies and build provider tree
    const sorted = topologicalSort(requestedProviders, providers);

    return sorted.reduceRight((acc, providerName) => {
      const { Provider } = providers.get(providerName);
      return <Provider>{acc}</Provider>;
    }, children);
  };

  return { registerProvider, buildProviderTree };
}

// Application-specific provider configuration
const AppProviderRegistry = createProviderHierarchy();

function ConfigProvider({ children }) {
  const config = {
    apiBaseURL: process.env.REACT_APP_API_URL,
    analyticsTrackingId: process.env.REACT_APP_ANALYTICS_ID
  };

  return (
    <ConfigContext.Provider value={config}>
      {children}
    </ConfigContext.Provider>
  );
}

function ApiProvider({ children }) {
  const config = useConfig();
  const api = useMemo(() => new PracticeAPIService(config.apiBaseURL), [config]);

  return (
    <ApiContext.Provider value={api}>
      {children}
    </ApiContext.Provider>
  );
}

// Register providers with dependencies
AppProviderRegistry.registerProvider('config', ConfigProvider);
AppProviderRegistry.registerProvider('api', ApiProvider, ['config']);
AppProviderRegistry.registerProvider('auth', AuthProvider, ['api']);
AppProviderRegistry.registerProvider('notifications', NotificationProvider);
AppProviderRegistry.registerProvider('practiceSession', PracticeSessionProvider,
  ['api', 'auth', 'notifications']);

// Application root with selective provider loading
function App() {
  return (
    <AppProviders providers={['config', 'api', 'auth', 'practiceSession']}>
      <Dashboard />
    </AppProviders>
  );
}

function AppProviders({ providers, children }) {
  return AppProviderRegistry.buildProviderTree(providers, children);
}
```

Performance Optimization Strategies

Provider architectures require careful performance optimization to prevent unnecessary re-renders and maintain smooth user experiences.

```
// Split context patterns for performance
const UserDataContext = createContext();
const UserActionsContext = createContext();

function OptimizedUserProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  // Memoize actions to prevent unnecessary re-renders
  const actions = useMemo(() => ({
    login: async (credentials) => {
      const user = await api.login(credentials);
      setUser(user);
    },
    logout: async () => {
      await api.logout();
      setUser(null);
    },
    updateUser: (updates) => {
      setUser(prev => ({ ...prev, ...updates }));
    }
  }), []);

  // Memoize data to prevent unnecessary re-renders
  const userData = useMemo(() => ({
    user,
    loading,
    isAuthenticated: !!user
  }), [user, loading]);

  return (
    <UserActionsContext.Provider value={actions}>
      <UserDataContext.Provider value={userData}>
        {children}
      </UserDataContext.Provider>
    </UserActionsContext.Provider>
  );
}

// Components subscribe only to what they need
function UserProfile() {
  const { user, loading } = useContext(UserDataContext);
  // Only re-renders when user data changes
}

function UserActions() {
  const { login, logout } = useContext(UserActionsContext);
  // Never re-renders due to user data changes
}
```

When to Use Context for Dependency Injection

Context-based dependency injection works best for:

- Application-wide services like API clients, analytics, and storage
- Cross-cutting concerns like authentication and theming
- Services that need to be easily mocked for testing
- Avoiding deep prop drilling for frequently used dependencies

Context design principles

- Keep contexts focused on a single concern
- Split frequently changing data from stable configuration
- Use multiple smaller contexts rather than one large context
- Provide clear error messages when contexts are used incorrectly
- Consider performance implications of context value changes

Context overuse

Not every piece of shared state needs Context. Use Context for truly application-wide concerns. For component-specific state sharing, consider lifting state up or using compound components instead.

Advanced Custom Hook Patterns

Custom hooks represent the pinnacle of React's composability philosophy. While basic custom hooks provide foundational reusability, advanced custom hook patterns enable sophisticated architectural solutions that manage state machines, coordinate complex asynchronous operations, and serve as comprehensive abstraction layers for application logic.

The true power of custom hooks emerges through their composability and architectural flexibility. Unlike higher-order components or render props, hooks integrate seamlessly, test in isolation, and provide clear interfaces for the logic they encapsulate. As applications scale in complexity, mastering advanced hook patterns becomes essential for maintaining clean, maintainable codebases.

Hooks as Architectural Boundaries

Advanced custom hooks function as more than state management tools—they serve as architectural boundaries that encapsulate business logic, coordinate side effects, and provide stable interfaces between components and complex application concerns. Well-designed hooks can eliminate the need for external state management libraries in many scenarios.

State Machine Patterns with Custom Hooks

Complex user interactions often benefit from explicit state machine modeling. Custom hooks can encapsulate state machines that manage intricate workflows with clearly defined state transitions and coordinated side effects.

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

// Practice session state machine hook
function usePracticeSessionStateMachine(initialSession = null) {
  const [state, setState] = useState('idle');
  const [session, setSession] = useState(initialSession);
  const [error, setError] = useState(null);
  const [progress, setProgress] = useState(0);

  const timerRef = useRef(null);
  const startTimeRef = useRef(null);

  // State machine transitions
  const transitions = {
    idle: ['preparing', 'error'],
    preparing: ['active', 'error', 'idle'],
    active: ['paused', 'completed', 'error'],
    paused: ['active', 'completed', 'error'],
    completed: ['idle'],
    error: ['idle', 'preparing']
  };

  const canTransition = useCallback((fromState, toState) => {
    return transitions[fromState]?.includes(toState) || false;
  }, []);

  const transition = useCallback((newState, payload = {}) => {
    if (!canTransition(state, newState)) {
      console.warn(`Invalid transition from ${state} to ${newState}`);
      return false;
    }

    setState(newState);

    // Handle side effects based on state transitions
    switch (newState) {
      case 'preparing':
        setError(null);
        setProgress(0);
        break;

      case 'active':
        startTimeRef.current = Date.now();
        timerRef.current = setInterval(() => {
          setProgress(prev => {
            const elapsed = Date.now() - startTimeRef.current;
            const targetDuration = session?.targetDuration || 1800000; // 30 minutes
            return Math.min((elapsed / targetDuration) * 100, 100);
          });
        }, 1000);
        break;

      case 'paused':
      case 'completed':
      case 'error':
        if (timerRef.current) {
          clearInterval(timerRef.current);
          timerRef.current = null;
        }
        break;
    }

    return true;
  }, [state, session, canTransition]);

  // Cleanup on unmount
  useEffect(() => {
    return () => {
      if (timerRef.current) {
        clearInterval(timerRef.current);
      }
    };
  }, []);

  // Public API
  const startSession = useCallback((sessionData) => {
    setSession(sessionData);
    return transition('preparing') && transition('active');
  }, [transition]);

  const pauseSession = useCallback(() => {
    return transition('paused');
  }, [transition]);

  const resumeSession = useCallback(() => {
    return transition('active');
  }, [transition]);

  const completeSession = useCallback(() => {
    return transition('completed');
  }, [transition]);

  const resetSession = useCallback(() => {
    setSession(null);
    setProgress(0);
    setError(null);
    return transition('idle');
  }, [transition]);

  const handleError = useCallback((errorMessage) => {
    setError(errorMessage);
    return transition('error');
  }, [transition]);

  return {
    state,
    session,
    error,
    progress,
    canTransition: (toState) => canTransition(state, toState),
    startSession,
    pauseSession,
    resumeSession,
    completeSession,
    resetSession,
    handleError,
    isIdle: state === 'idle',
    isPreparing: state === 'preparing',
    isActive: state === 'active',
    isPaused: state === 'paused',
    isCompleted: state === 'completed',
    hasError: state === 'error'
  };
}
```

This state machine hook provides a robust foundation for managing complex practice session workflows with clear state transitions and side effect management.

Advanced Data Synchronization and Caching Strategies

Modern applications require sophisticated data coordination from multiple sources while maintaining consistency and optimal performance. Custom hooks can provide advanced caching and synchronization strategies that handle complex data flows seamlessly.

```
// Advanced data synchronization hook with caching
function useDataSync(sources, options = {}) {
  const {
    cacheTimeout = 300000, // 5 minutes
    retryAttempts = 3,
    retryDelay = 1000,
    onError,
    onSuccess
  } = options;

  const [data, setData] = useState(new Map());
  const [loading, setLoading] = useState(new Set());
  const [errors, setErrors] = useState(new Map());
  const cache = useRef(new Map());
  const retryTimeouts = useRef(new Map());

  const isStale = useCallback((sourceId) => {
    const cached = cache.current.get(sourceId);
    if (!cached) return true;
    return Date.now() - cached.timestamp > cacheTimeout;
  }, [cacheTimeout]);

  const fetchSource = useCallback(async (sourceId, source, attempt = 1) => {
    setLoading(prev => new Set([...prev, sourceId]));
    setErrors(prev => {
      const newErrors = new Map(prev);
      newErrors.delete(sourceId);
      return newErrors;
    });

    try {
      const result = await source.fetch();

      // Cache the result
      cache.current.set(sourceId, {
        data: result,
        timestamp: Date.now()
      });

      setData(prev => new Map([...prev, [sourceId, result]]));
      onSuccess?.(sourceId, result);

    } catch (error) {
      if (attempt < retryAttempts) {
        // Schedule retry
        const timeoutId = setTimeout(() => {
          fetchSource(sourceId, source, attempt + 1);
        }, retryDelay * attempt);

        retryTimeouts.current.set(sourceId, timeoutId);
      } else {
        setErrors(prev => new Map([...prev, [sourceId, error]]));
        onError?.(sourceId, error);
      }
    } finally {
      setLoading(prev => {
        const newLoading = new Set(prev);
        newLoading.delete(sourceId);
        return newLoading;
      });
    }
  }, [retryAttempts, retryDelay, onError, onSuccess]);

  const syncData = useCallback(() => {
    Object.entries(sources).forEach(([sourceId, source]) => {
      if (isStale(sourceId)) {
        fetchSource(sourceId, source);
      } else {
        // Use cached data
        const cached = cache.current.get(sourceId);
        setData(prev => new Map([...prev, [sourceId, cached.data]]));
      }
    });
  }, [sources, isStale, fetchSource]);

  // Initial sync and periodic refresh
  useEffect(() => {
    syncData();

    const interval = setInterval(syncData, cacheTimeout);
    return () => clearInterval(interval);
  }, [syncData, cacheTimeout]);

  // Cleanup retry timeouts
  useEffect(() => {
    return () => {
      retryTimeouts.current.forEach(timeoutId => clearTimeout(timeoutId));
    };
  }, []);

  const refetch = useCallback((sourceId) => {
    if (sourceId) {
      cache.current.delete(sourceId);
      const source = sources[sourceId];
      if (source) {
        fetchSource(sourceId, source);
      }
    } else {
      cache.current.clear();
      syncData();
    }
  }, [sources, fetchSource, syncData]);

  return {
    data: Object.fromEntries(data),
    loading: Array.from(loading),
    errors: Object.fromEntries(errors),
    refetch,
    isLoading: loading.size > 0,
    hasErrors: errors.size > 0
  };
}

// Usage example
function PracticeStatsDashboard({ userId }) {
  const dataSources = {
    sessions: {
      fetch: () => PracticeAPI.getSessions(userId)
    },
    progress: {
      fetch: () => PracticeAPI.getProgress(userId)
    },
    goals: {
      fetch: () => PracticeAPI.getGoals(userId)
    }
  };

  const { data, loading, errors, refetch } = useDataSync(dataSources, {
    cacheTimeout: 600000, // 10 minutes
    onError: (sourceId, error) => {
      console.error(`Failed to fetch ${sourceId}:`, error);
    }
  });

  return (
    <div className="practice-stats">
      {loading.includes('sessions') ? (
        <div>Loading sessions...</div>
      ) : (
        <SessionStats sessions={data.sessions} />
      )}

      {data.progress && <ProgressChart data={data.progress} />}
      {data.goals && <GoalTracker goals={data.goals} />}

      <button onClick={() => refetch()}>Refresh All</button>
    </div>
  );
}
```

Async Coordination and Effect Management

Complex applications often need to coordinate multiple asynchronous operations with sophisticated error handling and dependency management.

```
// Advanced async coordination hook
function useAsyncCoordinator() {
  const [operations, setOperations] = useState(new Map());
  const pendingOperations = useRef(new Map());

  const registerOperation = useCallback((id, operation, dependencies = []) => {
    const operationState = {
      id,
      operation,
      dependencies,
      status: 'pending',
      result: null,
      error: null,
      startTime: null,
      endTime: null
    };

    setOperations(prev => new Map([...prev, [id, operationState]]));
    return id;
  }, []);

  const executeOperation = useCallback(async (id) => {
    const operation = operations.get(id);
    if (!operation) return;

    // Check if dependencies are completed
    const uncompletedDeps = operation.dependencies.filter(depId => {
      const dep = operations.get(depId);
      return !dep || dep.status !== 'completed';
    });

    if (uncompletedDeps.length > 0) {
      console.warn(`Operation ${id} has uncompleted dependencies:`, uncompletedDeps);
      return;
    }

    setOperations(prev => {
      const newOps = new Map(prev);
      const updatedOp = { ...operation, status: 'running', startTime: Date.now() };
      newOps.set(id, updatedOp);
      return newOps;
    });

    try {
      const dependencyResults = operation.dependencies.reduce((acc, depId) => {
        const dep = operations.get(depId);
        acc[depId] = dep?.result;
        return acc;
      }, {});

      const result = await operation.operation(dependencyResults);

      setOperations(prev => {
        const newOps = new Map(prev);
        const completedOp = {
          ...newOps.get(id),
          status: 'completed',
          result,
          endTime: Date.now()
        };
        newOps.set(id, completedOp);
        return newOps;
      });

      return result;
    } catch (error) {
      setOperations(prev => {
        const newOps = new Map(prev);
        const errorOp = {
          ...newOps.get(id),
          status: 'error',
          error,
          endTime: Date.now()
        };
        newOps.set(id, errorOp);
        return newOps;
      });

      throw error;
    }
  }, [operations]);

  const executeAll = useCallback(async () => {
    const sortedOps = topologicalSort(Array.from(operations.keys()), operations);
    const results = {};

    for (const opId of sortedOps) {
      try {
        results[opId] = await executeOperation(opId);
      } catch (error) {
        console.error(`Operation ${opId} failed:`, error);
      }
    }

    return results;
  }, [operations, executeOperation]);

  const reset = useCallback(() => {
    setOperations(new Map());
    pendingOperations.current.clear();
  }, []);

  return {
    registerOperation,
    executeOperation,
    executeAll,
    reset,
    operations: Array.from(operations.values()),
    isComplete: Array.from(operations.values()).every(op =>
      op.status === 'completed' || op.status === 'error'
    )
  };
}

```

```
// Usage example for complex practice session initialization
function usePracticeSessionInitialization(sessionConfig) {
  const coordinator = useAsyncCoordinator();
  const [initializationState, setInitializationState] = useState('idle');

  const initializeSession = useCallback(async () => {
    setInitializationState('initializing');

    try {
      // Register dependent operations
      const validateConfigId = coordinator.registerOperation(
        'validateConfig',
        async () => validateSessionConfig(sessionConfig)
      );

      const loadResourcesId = coordinator.registerOperation(
        'loadResources',
        async ({ validateConfig }) => loadSessionResources(validateConfig),
        ['validateConfig']
      );

      const setupAudioId = coordinator.registerOperation(
        'setupAudio',
        async ({ loadResources }) => setupAudioContext(loadResources.audioFiles),
        ['loadResources']
      );

      const initializeTimerId = coordinator.registerOperation(
        'initializeTimer',
        async ({ validateConfig }) => initializeSessionTimer(validateConfig.duration),
        ['validateConfig']
      );

      // Execute all operations
      const results = await coordinator.executeAll();

      setInitializationState('completed');
      return results;
    } catch (error) {
      setInitializationState('error');
      throw error;
    }
  }, [sessionConfig, coordinator]);

  return {
    initializeSession,
    initializationState,
    operations: coordinator.operations,
    reset: coordinator.reset
  };
}
```

Resource Management and Cleanup Patterns

Advanced hooks often need to manage complex resources with sophisticated cleanup strategies to prevent memory leaks and resource contention.

```
// Advanced resource management hook
function useResourceManager() {
  const resources = useRef(new Map());
  const cleanupFunctions = useRef(new Map());

  const registerResource = useCallback((id, resource, cleanup) => {
    // Clean up existing resource if it exists
    if (resources.current.has(id)) {
      releaseResource(id);
    }

    resources.current.set(id, resource);
    if (cleanup) {
      cleanupFunctions.current.set(id, cleanup);
    }

    return resource;
  }, []);

  const releaseResource = useCallback((id) => {
    const cleanup = cleanupFunctions.current.get(id);
    if (cleanup) {
      try {
        cleanup();
      } catch (error) {
        console.error(`Error cleaning up resource ${id}:`, error);
      }
    }

    resources.current.delete(id);
    cleanupFunctions.current.delete(id);
  }, []);

  const getResource = useCallback((id) => {
    return resources.current.get(id);
  }, []);

  const releaseAll = useCallback(() => {
    resources.current.forEach((_, id) => releaseResource(id));
  }, [releaseResource]);

  // Cleanup on unmount
  useEffect(() => {
    return () => releaseAll();
  }, [releaseAll]);

  return {
    registerResource,
    releaseResource,
    getResource,
    releaseAll,
    resourceCount: resources.current.size
  };
}

// Specialized hook for practice session resources
function usePracticeSessionResources() {
  const resourceManager = useResourceManager();
  const [resourceState, setResourceState] = useState({});

  const loadAudioResource = useCallback(async (audioUrl) => {
    try {
      const audio = new Audio(audioUrl);

      // Wait for audio to be ready
      await new Promise((resolve, reject) => {
        audio.addEventListener('canplaythrough', resolve);
        audio.addEventListener('error', reject);
        audio.load();
      });

      resourceManager.registerResource('audio', audio, () => {
        audio.pause();
        audio.src = '';
      });

      setResourceState(prev => ({ ...prev, audioLoaded: true }));
      return audio;
    } catch (error) {
      setResourceState(prev => ({ ...prev, audioError: error.message }));
      throw error;
    }
  }, [resourceManager]);

  const loadMetronomeResource = useCallback(async () => {
    try {
      const metronome = new MetronomeEngine();
      await metronome.initialize();

      resourceManager.registerResource('metronome', metronome, () => {
        metronome.stop();
        metronome.destroy();
      });

      setResourceState(prev => ({ ...prev, metronomeLoaded: true }));
      return metronome;
    } catch (error) {
      setResourceState(prev => ({ ...prev, metronomeError: error.message }));
      throw error;
    }
  }, [resourceManager]);

  const getAudio = useCallback(() => {
    return resourceManager.getResource('audio');
  }, [resourceManager]);

  const getMetronome = useCallback(() => {
    return resourceManager.getResource('metronome');
  }, [resourceManager]);

  return {
    loadAudioResource,
    loadMetronomeResource,
    getAudio,
    getMetronome,
    releaseAll: resourceManager.releaseAll,
    resourceState
  };
}
```

Composable Hook Factories

Advanced patterns often involve creating hooks that generate other hooks, providing flexible abstractions for common patterns.

```
// Factory for creating data management hooks
function createDataHook(config) {
  const {
    endpoint,
    transform = data => data,
    cacheKey,
    dependencies = [],
    onError,
    onSuccess
  } = config;

  return function useData(...params) {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    const fetchData = useCallback(async () => {
      try {
        setLoading(true);
        setError(null);

        const response = await fetch(endpoint(...params));
        const rawData = await response.json();
        const transformedData = transform(rawData);

        setData(transformedData);
        onSuccess?.(transformedData);
      } catch (err) {
        setError(err);
        onError?.(err);
      } finally {
        setLoading(false);
      }
    }, params);

    useEffect(() => {
      fetchData();
    }, [fetchData, ...dependencies]);

    return {
      data,
      loading,
      error,
      refetch: fetchData
    };
  };
}

// Factory usage
const usePracticeSessions = createDataHook({
  endpoint: (userId) => `/api/users/${userId}/sessions`,
  transform: (sessions) => sessions.map(session => ({
    ...session,
    date: new Date(session.date),
    duration: session.duration * 60 // Convert to seconds
  })),
  cacheKey: 'practice-sessions'
});

const useSessionAnalytics = createDataHook({
  endpoint: (userId, dateRange) => `/api/users/${userId}/analytics?${dateRange}`,
  transform: (analytics) => ({
    ...analytics,
    averageSession: analytics.totalTime / analytics.sessionCount
  })
});

// Hook composition factory
function createCompositeHook(...hookFactories) {
  return function useComposite(...params) {
    const results = hookFactories.map(factory => factory(...params));

    return results.reduce((acc, result, index) => {
      acc[`hook${index}`] = result;
      return acc;
    }, {
      loading: results.some(r => r.loading),
      error: results.find(r => r.error)?.error,
      refetchAll: () => results.forEach(r => r.refetch?.())
    });
  };
}
```

These advanced hook patterns provide powerful abstractions that can significantly improve code organization, reusability, and maintainability in complex React applications. They represent the evolution of React's compositional model and demonstrate how hooks can serve as architectural foundations for sophisticated applications.

Provider Patterns and Architectural Composition

Provider patterns extend far beyond simple prop drilling solutions. When implemented with architectural sophistication, providers become the foundational infrastructure of scalable applications—they replace complex state management libraries, coordinate service dependencies, and establish clean architectural boundaries that enhance code maintainability and development experience.

The provider pattern's architectural strength emerges through its ability to create clear boundaries while preserving flexibility and testability. Advanced provider patterns manage complex application state, coordinate multiple service dependencies, and provide elegant solutions for cross-cutting concerns including authentication, theming, and API management.

Providers as Architectural Infrastructure

Well-designed provider patterns form the foundational infrastructure of scalable React applications. They provide dependency injection, state management, and service coordination while maintaining clear separation of concerns. Advanced provider architectures can eliminate the need for external state management libraries in many application scenarios.

Hierarchical Provider Composition Strategies

Complex applications benefit from hierarchical provider structures that enable granular control over dependencies and state scope. This architectural pattern allows different application sections to access distinct sets of services and state management capabilities.

```
// Base provider system with dependency resolution
function createProviderHierarchy() {
  const providers = new Map();

  const registerProvider = (name, Provider, dependencies = []) => {
    providers.set(name, { Provider, dependencies });
  };

  const buildProviderTree = (requestedProviders, children) => {
    // Resolve dependencies and build provider tree
    const sorted = topologicalSort(requestedProviders, providers);

    return sorted.reduceRight((acc, providerName) => {
      const { Provider } = providers.get(providerName);
      return <Provider key={providerName}>{acc}</Provider>;
    }, children);
  };

  return { registerProvider, buildProviderTree };
}

// Application-specific provider configuration
const AppProviderRegistry = createProviderHierarchy();

function ConfigProvider({ children }) {
  const config = {
    apiBaseURL: process.env.REACT_APP_API_URL,
    analyticsTrackingId: process.env.REACT_APP_ANALYTICS_ID,
    features: {
      advancedAnalytics: process.env.REACT_APP_ADVANCED_ANALYTICS === 'true',
      socialSharing: process.env.REACT_APP_SOCIAL_SHARING === 'true'
    }
  };

  return (
    <ConfigContext.Provider value={config}>
      {children}
    </ConfigContext.Provider>
  );
}

function ApiProvider({ children }) {
  const config = useConfig();
  const api = useMemo(() => new PracticeAPIService(config.apiBaseURL), [config]);

  return (
    <ApiContext.Provider value={api}>
      {children}
    </ApiContext.Provider>
  );
}

// Register providers with dependencies
AppProviderRegistry.registerProvider('config', ConfigProvider);
AppProviderRegistry.registerProvider('api', ApiProvider, ['config']);
AppProviderRegistry.registerProvider('auth', AuthProvider, ['api']);
AppProviderRegistry.registerProvider('notifications', NotificationProvider);
AppProviderRegistry.registerProvider('practiceSession', PracticeSessionProvider,
  ['api', 'auth', 'notifications']);
```

```

// Application root with selective provider loading
function App() {
  return (
    <AppProviders providers={['config', 'api', 'auth', 'practiceSession']}>
      <Dashboard />
    </AppProviders>
  );
}

function AppProviders({ providers, children }) {
  return AppProviderRegistry.buildProviderTree(providers, children);
}
```

Service Container Patterns

Service containers provide sophisticated dependency injection with lazy loading, service decoration, and complex service resolution patterns.

```
// Advanced service container implementation
class ServiceContainer {
  constructor() {
    this.services = new Map();
    this.singletons = new Map();
    this.factories = new Map();
    this.decorators = new Map();
  }

  register(name, factory, options = {}) {
    const { singleton = false, dependencies = [] } = options;

    this.factories.set(name, {
      factory,
      dependencies,
      singleton
    });
  }

  resolve(name) {
    // Check if singleton instance exists
    if (this.singletons.has(name)) {
      return this.singletons.get(name);
    }

    const serviceConfig = this.factories.get(name);
    if (!serviceConfig) {
      throw new Error(`Service '${name}' not registered`);
    }

    // Resolve dependencies
    const dependencies = serviceConfig.dependencies.reduce((deps, depName) => {
      deps[depName] = this.resolve(depName);
      return deps;
    }, {});

    // Create service instance
    let instance = serviceConfig.factory(dependencies);

    // Apply decorators
    const decorators = this.decorators.get(name) || [];
    instance = decorators.reduce((service, decorator) => decorator(service), instance);

    // Store singleton if needed
    if (serviceConfig.singleton) {
      this.singletons.set(name, instance);
    }

    return instance;
  }

  decorate(serviceName, decorator) {
    if (!this.decorators.has(serviceName)) {
      this.decorators.set(serviceName, []);
    }
    this.decorators.get(serviceName).push(decorator);
  }

  clear() {
    this.services.clear();
    this.singletons.clear();
  }
}

// Service container provider
function ServiceContainerProvider({ children }) {
  const container = useMemo(() => {
    const serviceContainer = new ServiceContainer();

    // Register core services
    serviceContainer.register('config', () => ({
      apiBaseURL: process.env.REACT_APP_API_URL,
      enableAnalytics: process.env.REACT_APP_ANALYTICS === 'true'
    }), { singleton: true });

    serviceContainer.register('httpClient', ({ config }) => {
      return new HttpClient(config.apiBaseURL);
    }, { dependencies: ['config'], singleton: true });

    serviceContainer.register('practiceAPI', ({ httpClient }) => {
      return new PracticeAPIService(httpClient);
    }, { dependencies: ['httpClient'], singleton: true });

    serviceContainer.register('analytics', ({ config }) => {
      return config.enableAnalytics ? new AnalyticsService() : new NoOpAnalyticsService();
    }, { dependencies: ['config'], singleton: true });

    // Add logging decorator to all services
    serviceContainer.decorate('practiceAPI', (service) => {
      return new Proxy(service, {
        get(target, prop) {
          if (typeof target[prop] === 'function') {
            return function(...args) {
              console.log(`Calling ${prop} with args:`, args);
              return target[prop].apply(target, args);
            };
          }
          return target[prop];
        }
      });
    });

    return serviceContainer;
  }, []);

  return (
    <ServiceContainerContext.Provider value={container}>
      {children}
    </ServiceContainerContext.Provider>
  );
}

function useService(serviceName) {
  const container = useContext(ServiceContainerContext);
  return useMemo(() => container.resolve(serviceName), [container, serviceName]);
}
```

Performance Optimization Strategies

Provider architectures require careful performance optimization to prevent unnecessary re-renders and maintain smooth user experiences.

```
// Split context patterns for performance
```

```
const UserDataContext = createContext();
const UserActionsContext = createContext();

function OptimizedUserProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const [preferences, setPreferences] = useState({});

  // Memoize actions to prevent unnecessary re-renders
  const actions = useMemo(() => ({
    login: async (credentials) => {
      const user = await api.login(credentials);
      setUser(user);
      return user;
    },
    logout: async () => {
      await api.logout();
      setUser(null);
      setPreferences({});
    },
    updateUser: (updates) => {
      setUser(prev => ({ ...prev, ...updates }));
    },
    updatePreferences: (newPreferences) => {
      setPreferences(prev => ({ ...prev, ...newPreferences }));
    }
  }), []);

  // Memoize stable data to prevent unnecessary re-renders
  const userData = useMemo(() => ({
    user,
    loading,
    preferences,
    isAuthenticated: !!user,
    isAdmin: user?.role === 'admin'
  }), [user, loading, preferences]);

  return (
    <UserActionsContext.Provider value={actions}>
      <UserDataContext.Provider value={userData}>
        {children}
      </UserDataContext.Provider>
    </UserActionsContext.Provider>
  );
}

// Components subscribe only to what they need
function UserProfile() {
  const { user, loading } = useContext(UserDataContext);
  // Only re-renders when user data changes, not when actions change

  if (loading) return <div>Loading...</div>;

  return (
    <div className="user-profile">
      <h2>{user?.name}</h2>
      <p>{user?.email}</p>
    </div>
  );
}

function UserActions() {
  const { logout, updateUser } = useContext(UserActionsContext);
  // Never re-renders due to user data changes

  return (
    <div className="user-actions">
      <button onClick={logout}>Logout</button>
      <button onClick={() => updateUser({ lastActive: new Date() })}>
        Update Activity
      </button>
    </div>
  );
}
```

Multi-Tenant Provider Architecture

For applications that need to support multiple contexts or tenants, advanced provider patterns can manage isolated state while sharing common services.

```
// Multi-tenant provider system
function createTenantProvider(tenantId) {
  return function TenantProvider({ children }) {
    const [tenantData, setTenantData] = useState(null);
    const [loading, setLoading] = useState(true);
    const globalServices = useServices();

    useEffect(() => {
      globalServices.api.getTenant(tenantId)
        .then(setTenantData)
        .finally(() => setLoading(false));
    }, [tenantId, globalServices.api]);

    const tenantServices = useMemo(() => ({
      ...globalServices,
      tenantAPI: new TenantSpecificAPI(tenantId, globalServices.httpClient),
      tenantConfig: tenantData?.config || {},
      tenantId
    }), [globalServices, tenantData, tenantId]);

    if (loading) return <div>Loading tenant...</div>;

    return (
      <TenantContext.Provider value={tenantServices}>
        {children}
      </TenantContext.Provider>
    );
  };
}

// Workspace isolation provider
function WorkspaceProvider({ workspaceId, children }) {
  const tenant = useTenant();
  const [workspace, setWorkspace] = useState(null);
  const [permissions, setPermissions] = useState({});

  useEffect(() => {
    Promise.all([
      tenant.tenantAPI.getWorkspace(workspaceId),
      tenant.tenantAPI.getWorkspacePermissions(workspaceId)
    ]).then(([workspaceData, permissionsData]) => {
      setWorkspace(workspaceData);
      setPermissions(permissionsData);
    });
  }, [workspaceId, tenant.tenantAPI]);

  const workspaceServices = useMemo(() => ({
    ...tenant,
    workspace,
    permissions,
    workspaceAPI: new WorkspaceAPI(workspaceId, tenant.tenantAPI)
  }), [tenant, workspace, permissions, workspaceId]);

  return (
    <WorkspaceContext.Provider value={workspaceServices}>
      {children}
    </WorkspaceContext.Provider>
  );
}

// Usage with nested providers
function App() {
  const tenantId = useCurrentTenant();
  const workspaceId = useCurrentWorkspace();

  const TenantProvider = createTenantProvider(tenantId);

  return (
    <GlobalServicesProvider>
```

```
      <TenantProvider>
        <WorkspaceProvider workspaceId={workspaceId}>
          <Dashboard />
        </WorkspaceProvider>
      </TenantProvider>
    </GlobalServicesProvider>
  );
}
```

Event-Driven Provider Patterns

Advanced provider architectures can incorporate event-driven patterns for loose coupling and reactive updates.

```
// Event bus provider for loose coupling
function EventBusProvider({ children }) {
  const eventBus = useMemo(() => {
    const listeners = new Map();

    const on = (event, callback) => {
      if (!listeners.has(event)) {
        listeners.set(event, new Set());
      }
      listeners.get(event).add(callback);

      // Return unsubscribe function
      return () => {
        listeners.get(event)?.delete(callback);
      };
    };

    const emit = (event, data) => {
      const eventListeners = listeners.get(event);
      if (eventListeners) {
        eventListeners.forEach(callback => {
          try {
            callback(data);
          } catch (error) {
            console.error(`Error in event listener for ${event}:`, error);
          }
        });
      }
    };

    const once = (event, callback) => {
      const unsubscribe = on(event, (data) => {
        callback(data);
        unsubscribe();
      });
      return unsubscribe;
    };

    return { on, emit, once };
  }, []);

  return (
    <EventBusContext.Provider value={eventBus}>
      {children}
    </EventBusContext.Provider>
  );
}

// Practice session provider with event integration
function PracticeSessionProvider({ children }) {
  const [currentSession, setCurrentSession] = useState(null);
  const [sessionHistory, setSessionHistory] = useState([]);
  const eventBus = useEventBus();
  const api = useAPI();

  // Listen for session events
  useEffect(() => {
    const unsubscribeStart = eventBus.on('session:start', async (sessionData) => {
      const session = await api.createSession(sessionData);
      setCurrentSession(session);
      eventBus.emit('session:created', session);
    });

    const unsubscribeComplete = eventBus.on('session:complete', async (sessionId) => {
      const completedSession = await api.completeSession(sessionId);
      setCurrentSession(null);
      setSessionHistory(prev => [completedSession, ...prev]);
      eventBus.emit('session:completed', completedSession);
    });

    return () => {
      unsubscribeStart();
      unsubscribeComplete();
    };
  }, [eventBus, api]);

  const contextValue = {
    currentSession,
    sessionHistory,
    startSession: (sessionData) => eventBus.emit('session:start', sessionData),
    completeSession: (sessionId) => eventBus.emit('session:complete', sessionId)
  };

  return (
    <PracticeSessionContext.Provider value={contextValue}>
      {children}
    </PracticeSessionContext.Provider>
  );
}

// Analytics provider that reacts to session events
function AnalyticsProvider({ children }) {
  const eventBus = useEventBus();
  const analytics = useService('analytics');

  useEffect(() => {
    const unsubscribeCreated = eventBus.on('session:created', (session) => {
      analytics.track('practice_session_started', {
        sessionId: session.id,
        piece: session.piece,
        duration: session.targetDuration
      });
    });

    const unsubscribeCompleted = eventBus.on('session:completed', (session) => {
      analytics.track('practice_session_completed', {
        sessionId: session.id,
        actualDuration: session.actualDuration,
        targetDuration: session.targetDuration,
        completion: session.actualDuration / session.targetDuration
      });
    });

    return () => {
      unsubscribeCreated();
      unsubscribeCompleted();
    };
  }, [eventBus, analytics]);

  return <>{children}</>;
}
```

When to Use Advanced Provider Patterns

Advanced provider patterns work best for:

- Large applications with complex state management needs
- Multi-tenant or multi-workspace applications
- Applications requiring sophisticated dependency injection
- Systems with many cross-cutting concerns
- Applications that need to coordinate between multiple isolated contexts

Provider architecture principles

- Keep providers focused on a single concern or domain
- Use hierarchical composition for complex dependency relationships
- Split frequently changing data from stable configuration
- Implement proper error boundaries around provider trees
- Consider performance implications of context value changes
- Use event-driven patterns for loose coupling between providers

Complexity management

Advanced provider patterns add significant complexity to your application architecture. Use them when the benefits clearly outweigh the costs, and ensure your team understands the patterns before implementing them in production code.

Error Boundaries and Resilient Error Handling

Error boundaries represent one of React's most critical architectural patterns for building resilient applications. While error handling may not be the most exciting development topic, it distinguishes professional applications from experimental projects and ensures positive user experiences when inevitable failures occur.

Effective error handling transforms potentially catastrophic failures into manageable user experiences. Real-world applications face countless failure scenarios: network timeouts, browser inconsistencies, unexpected user interactions, and external service disruptions. Sophisticated error handling patterns prepare applications to handle these scenarios gracefully while maintaining functionality and user trust.

Error Boundaries as Application Resilience

Error boundaries provide React's mechanism for graceful failure handling—when components fail, error boundaries prevent application crashes by displaying fallback interfaces instead of blank screens. Advanced error handling patterns combine error boundaries with monitoring systems, retry logic, and fallback strategies to create robust error management architectures.

Modern React applications require comprehensive error handling strategies that gracefully degrade functionality, provide meaningful user feedback, and maintain application stability even when individual features fail. Advanced error handling patterns integrate error boundaries with context providers, custom hooks, and monitoring systems to establish resilient error management architectures.

Error Boundary Architecture Fundamentals

Before exploring advanced patterns, understanding error boundary capabilities and limitations proves essential. Error boundaries catch JavaScript errors throughout child component trees, log error details, and display fallback interfaces instead of crashed component hierarchies.

Error Boundary Limitations

Error boundaries do not catch errors inside event handlers, asynchronous code (e.g., `setTimeout` or `requestAnimationFrame` callbacks), or errors thrown during server-side rendering. For these scenarios, additional error handling strategies are required.

Advanced Error Boundary Implementation Patterns

Modern error boundaries extend beyond simple try-catch wrappers to provide comprehensive error management with retry logic, fallback strategies, and integrated error reporting capabilities.

```
// Advanced error boundary with retry and fallback strategies
class AdvancedErrorBoundary extends Component {
  constructor(props) {
    super(props);

    this.state = {
      hasError: false,
      error: null,
      errorInfo: null,
      retryCount: 0,
      errorId: null
    };

    this.retryTimeouts = new Set();
  }

  static getDerivedStateFromError(error) {
    // Basic error state update
    return {
      hasError: true,
      error,
      errorId: `error_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`
    };
  }

  componentDidCatch(error, errorInfo) {
    const { onError, maxRetries = 3, retryDelay = 1000 } = this.props;

    // Enhanced error state with detailed information
    this.setState({
      error,
      errorInfo,
      retryCount: this.state.retryCount + 1
    });

    // Report error to monitoring service
    this.reportError(error, errorInfo);
    // Call custom error handler
    if (onError) {
      onError(error, errorInfo, {
        retryCount: this.state.retryCount,
        canRetry: this.state.retryCount < maxRetries
      });
    }

    // Auto-retry logic for recoverable errors
    if (this.isRecoverableError(error) && this.state.retryCount < maxRetries) {
      const timeout = setTimeout(() => {
        this.retry();
      }, retryDelay * Math.pow(2, this.state.retryCount)); // Exponential backoff

      this.retryTimeouts.add(timeout);
    }
  }

  componentWillUnmount() {
    // Clean up retry timeouts
    this.retryTimeouts.forEach(timeout => clearTimeout(timeout));
  }

  isRecoverableError = (error) => {
    // Define which errors are recoverable
    const recoverableErrors = [
      'ChunkLoadError', // Code splitting errors
      'NetworkError',   // Network-related errors
      'TimeoutError'    // Request timeout errors
    ];

    return recoverableErrors.some(errorType =>
      error.name === errorType || error.message.includes(errorType)
    );
  };

  reportError = async (error, errorInfo) => {
    const { errorReporting } = this.props;

    if (!errorReporting) return;

    try {
      const errorReport = {
        id: this.state.errorId,
        message: error.message,
        stack: error.stack,
        componentStack: errorInfo.componentStack,
        timestamp: new Date().toISOString(),
        userAgent: navigator.userAgent,
        url: window.location.href,
        userId: this.props.userId,
        buildVersion: process.env.REACT_APP_VERSION,
        retryCount: this.state.retryCount,
        additionalContext: {
          props: this.props.errorContext,
          state: this.state
        }
      };

      await errorReporting.report(errorReport);
    } catch (reportingError) {
      console.error('Failed to report error:', reportingError);
    }
  };

  retry = () => {
    this.setState({
      hasError: false,
      error: null,
      errorInfo: null,
      errorId: null
    });
  };

  render() {
    if (this.state.hasError) {
      const { fallback: Fallback, children } = this.props;
      const { error, retryCount, maxRetries = 3 } = this.props;

      // Custom fallback component
      if (Fallback) {
        return (
          <Fallback
            error={this.state.error}
            errorInfo={this.state.errorInfo}
            retry={this.retry}
            canRetry={retryCount < maxRetries}
            retryCount={retryCount}
          />
        );
      }

      // Default fallback UI
      return (
        <ErrorFallback
          error={this.state.error}
          retry={this.retry}
          canRetry={retryCount < maxRetries}
          retryCount={retryCount}
        />
      );
    }

    return this.props.children;
  }
}

// Enhanced fallback component
function ErrorFallback({
  error,
  retry,
  canRetry,
  retryCount,
  title = "Something went wrong",
  showDetails = false
}) {
  const [showErrorDetails, setShowErrorDetails] = useState(showDetails);

  return (
    <div className="error-boundary-fallback">
      <div className="error-content">
        <div className="error-icon">[!]</div>
        <h2>{title}</h2>
        <p>We're sorry, but something unexpected happened.</p>

        {retryCount > 0 && (
          <p className="retry-info">
            Retry attempts: {retryCount}
          </p>
        )}

        <div className="error-actions">
          {canRetry && (
            <button
              onClick={retry}
              className="retry-button"
            >
              Try Again
            </button>
          )}

          <button
            onClick={() => window.location.reload()}
            className="reload-button"
          >
            Reload Page
          </button>

          <button
```

```
            onClick={() => setShowErrorDetails(!showErrorDetails)}
            className="details-button"
          >
            {showErrorDetails ? 'Hide' : 'Show'} Details
          </button>
        </div>

        {showErrorDetails && (
          <details className="error-details">
            <summary>Technical Details</summary>
            <pre className="error-stack">
              {error.stack}
            </pre>
          </details>
        )}
      </div>
    </div>
  );
}

// Hook for programmatic error boundary usage
function useErrorBoundary() {
  const [error, setError] = useState(null);

  const resetError = useCallback(() => {
    setError(null);
  }, []);

  const captureError = useCallback((error) => {
    setError(error);
  }, []);

  useEffect(() => {
    if (error) {
      throw error;
    }
  }, [error]);

  return { captureError, resetError };
}

// Practice app error boundary configuration
function PracticeErrorBoundary({ children, feature }) {
  const errorReporting = useService('errorReporting');
  const auth = useAuth();

  return (
    <AdvancedErrorBoundary
      onError={(error, errorInfo, context) => {
        console.error(`Error in ${feature}:`, error, context);
      }}
      errorReporting={errorReporting}
      userId={auth.getCurrentUser()?.id}
      errorContext={{ feature }}
      maxRetries={3}
      retryDelay={1000}
      fallback={({ error, retry, canRetry, retryCount }) => (
        <div className="practice-error-fallback">
          <h3>Practice Feature Unavailable</h3>
          <p>
            The {feature} feature is temporarily unavailable.
            {canRetry ? ' We\'ll try to restore it automatically.' : ''}
          </p>
          {canRetry && (
            <button onClick={retry}>
              Retry Now ({retryCount}/3)
            </button>
          )}
        </div>
      )}
    >
      {children}
    </AdvancedErrorBoundary>
  );
}
```

Implementing Context-Based Error Management

Context patterns can create application-wide error management systems that coordinate error handling across different features and provide centralized error reporting and recovery.

```
// Global error management context
const ErrorManagementContext = createContext();

function ErrorManagementProvider({ children }) {
  const [errors, setErrors] = useState(new Map());
  const [globalErrorState, setGlobalErrorState] = useState('healthy');

  // Error categorization and priority
  const errorCategories = {
    CRITICAL: { priority: 1, color: 'red', autoRetry: false },
    HIGH: { priority: 2, color: 'orange', autoRetry: true },
    MEDIUM: { priority: 3, color: 'yellow', autoRetry: true },
    LOW: { priority: 4, color: 'blue', autoRetry: true }
  };

  const errorManager = useMemo(() => ({
    // Register an error with context
    reportError: (error, context = {}) => {
      const errorId = `${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
      const severity = classifyError(error);

      const errorEntry = {
        id: errorId,
        error,
        context,
        severity,
        timestamp: new Date(),
        resolved: false,
        retryCount: 0,
        category: errorCategories[severity]
      };

      setErrors(prev => new Map(prev).set(errorId, errorEntry));

      // Update global error state based on severity
      if (severity === 'CRITICAL') {
        setGlobalErrorState('critical');
      } else if (severity === 'HIGH' && globalErrorState === 'healthy') {
        setGlobalErrorState('degraded');
      }

      return errorId;
    },

    // Resolve an error
    resolveError: (errorId) => {
      setErrors(prev => {
        const newErrors = new Map(prev);
        const error = newErrors.get(errorId);
        if (error) {
          newErrors.set(errorId, { ...error, resolved: true });
        }
        return newErrors;
      });

      // Update global state if no critical errors remain
      const unresolvedCritical = Array.from(errors.values())
        .some(e => e.severity === 'CRITICAL' && !e.resolved && e.id !== errorId);

      if (!unresolvedCritical) {
        const unresolvedHigh = Array.from(errors.values())
          .some(e => e.severity === 'HIGH' && !e.resolved && e.id !== errorId);

        setGlobalErrorState(unresolvedHigh ? 'degraded' : 'healthy');
      }
    },

    // Retry error resolution
    retryError: async (errorId, retryFunction) => {
      const error = errors.get(errorId);
      if (!error) return;

      try {
        await retryFunction();
        errorManager.resolveError(errorId);
      } catch (retryError) {
        setErrors(prev => {
          const newErrors = new Map(prev);
          const errorEntry = newErrors.get(errorId);
          if (errorEntry) {
            newErrors.set(errorId, {
              ...errorEntry,
              retryCount: errorEntry.retryCount + 1,
              lastRetryError: retryError
            });
          }
          return newErrors;
        });
      }
    },

    // Get errors by category
    getErrorsByCategory: (category) => {
      return Array.from(errors.values())
        .filter(error => error.severity === category && !error.resolved);
    },

    // Get all active errors
    getActiveErrors: () => {
      return Array.from(errors.values())
        .filter(error => !error.resolved);
    },

    // Clear resolved errors
    clearResolvedErrors: () => {
      setErrors(prev => {
        const newErrors = new Map();
        Array.from(prev.values())
          .filter(error => !error.resolved)
          .forEach(error => newErrors.set(error.id, error));
        return newErrors;
      });
    }
```

```
  }), [errors, globalErrorState]);

  // Auto-retry mechanism for retryable errors
  useEffect(() => {
    const retryableErrors = Array.from(errors.values())
      .filter(error =>
        !error.resolved &&
        error.category.autoRetry &&
        error.retryCount < 3
      );

    retryableErrors.forEach(error => {
      const delay = Math.pow(2, error.retryCount) * 1000; // Exponential backoff

      setTimeout(() => {
        if (error.context.retryFunction) {
          errorManager.retryError(error.id, error.context.retryFunction);
        }
      }, delay);
    });
  }, [errors, errorManager]);

  const contextValue = useMemo(() => ({
    ...errorManager,
    errors,
    globalErrorState,
    errorCategories
  }), [errorManager, errors, globalErrorState]);

  return (
    <ErrorManagementContext.Provider value={contextValue}>
      {children}
      <GlobalErrorDisplay />
    </ErrorManagementContext.Provider>
  );
}

// Helper function to classify errors
function classifyError(error) {
  // Network errors
  if (error.name === 'NetworkError' || error.message.includes('fetch')) {
    return 'HIGH';
  }

  // Authentication errors
  if (error.status === 401 || error.status === 403) {
    return 'CRITICAL';
  }

  // Code splitting errors
  if (error.name === 'ChunkLoadError') {
    return 'MEDIUM';
  }

  // Validation errors
  if (error.name === 'ValidationError') {
    return 'LOW';
  }

  // Unknown errors default to HIGH
  return 'HIGH';
}

// Hook for using error management
function useErrorManagement() {
  const context = useContext(ErrorManagementContext);
  if (!context) {
    throw new Error('useErrorManagement must be used within ErrorManagementProvider');
  }
  return context;
}

// Global error display component
function GlobalErrorDisplay() {
  const { getActiveErrors, resolveError, globalErrorState } = useErrorManagement();
  const [isVisible, setIsVisible] = useState(false);

  const activeErrors = getActiveErrors();
  const criticalErrors = activeErrors.filter(e => e.severity === 'CRITICAL');

  useEffect(() => {
    setIsVisible(criticalErrors.length > 0);
  }, [criticalErrors.length]);

  if (!isVisible) return null;

  return (
    <div className={`global-error-banner ${globalErrorState}`}>
      <div className="error-content">
        <span className="error-icon">[!]</span>
        <div className="error-message">
          {criticalErrors.length === 1 ? (
            <span>A critical error has occurred: {criticalErrors[0].error.message}</span>
          ) : (
            <span>{criticalErrors.length} critical errors require attention</span>
          )}
        </div>
        <div className="error-actions">
          <button
            onClick={() => criticalErrors.forEach(e => resolveError(e.id))}
            className="dismiss-button"
          >
            Dismiss
          </button>
          <button
            onClick={() => window.location.reload()}
            className="reload-button"
          >
            Reload Page
          </button>
        </div>
      </div>
    </div>
  );
}

// Practice-specific error handling hooks
function usePracticeSessionErrors() {
  const { reportError, resolveError } = useErrorManagement();

  const handleSessionError = useCallback((error, sessionId) => {
    const errorId = reportError(error, {
      feature: 'practice-session',
      sessionId,
      retryFunction: () => {
        // Retry logic specific to practice sessions
        return new Promise((resolve, reject) => {
          // Attempt to recover session state
          setTimeout(() => {
            if (Math.random() > 0.3) {
              resolve();
            } else {
              reject(new Error('Retry failed'));
            }
          }, 1000);
        });
      }
    });

    return errorId;
  }, [reportError]);

  return { handleSessionError, resolveError };
```

```
}

// Usage in practice components
function PracticeSessionPlayer({ sessionId }) {
  const { handleSessionError } = usePracticeSessionErrors();
  const [sessionData, setSessionData] = useState(null);
  const [error, setError] = useState(null);

  const loadSession = useCallback(async () => {
    try {
      const data = await api.getSession(sessionId);
      setSessionData(data);
      setError(null);
    } catch (loadError) {
      setError(loadError);
      handleSessionError(loadError, sessionId);
    }
  }, [sessionId, handleSessionError]);

  useEffect(() => {
    loadSession();
  }, [loadSession]);

  if (error) {
    return (
      <div className="session-error">
        <p>Failed to load practice session</p>
        <button onClick={loadSession}>Retry</button>
      </div>
    );
  }

  // Component implementation...
}
```

Mastering Asynchronous Error Handling

Modern React applications heavily rely on asynchronous operations, requiring sophisticated patterns for handling async errors, implementing retry logic, and managing loading states with proper error boundaries.

Async error handling challenges

Error boundaries don't catch errors in async operations, event handlers, or effects. You need additional patterns to handle these scenarios effectively.

```
// Advanced async error handling hook
function useAsyncOperation(operation, options = {}) {
  const {
    retries = 3,
    retryDelay = 1000,
    timeout = 30000,
    onError,
    onSuccess,
    dependencies = []
  } = options;

  const [state, setState] = useState({
    data: null,
    loading: false,
    error: null,
    retryCount: 0
  });

  const { reportError } = useErrorManagement();

  const executeOperation = useCallback(async (...args) => {
    let currentRetry = 0;

    setState(prev => ({
      ...prev,
      loading: true,
      error: null,
      retryCount: 0
    }));

    while (currentRetry <= retries) {
      try {
        // Create timeout promise
        const timeoutPromise = new Promise((_, reject) =>
          setTimeout(() => reject(new Error('Operation timeout')), timeout)
        );

        // Race operation against timeout
        const data = await Promise.race([
          operation(...args),
          timeoutPromise
        ]);

        setState(prev => ({
          ...prev,
          data,
          loading: false,
          error: null,
          retryCount: currentRetry
        }));

        if (onSuccess) {
          onSuccess(data);
        }

        return data;

      } catch (error) {
        currentRetry++;

        setState(prev => ({
          ...prev,
          retryCount: currentRetry,
          error: currentRetry > retries ? error : prev.error
        }));

        if (currentRetry <= retries) {
          // Exponential backoff for retries
          const delay = retryDelay * Math.pow(2, currentRetry - 1);
          await new Promise(resolve => setTimeout(resolve, delay));
        } else {
          // Final failure - report error and update state
          setState(prev => ({
            ...prev,
            loading: false,
            error
          }));

          const errorId = reportError(error, {
            operation: operation.name || 'async-operation',
            args,
            retries,
            finalRetryCount: currentRetry - 1
          });

          if (onError) {
            onError(error, errorId);
          }

          throw error;
        }
      }
    }
  }, [operation, retries, retryDelay, timeout, onError, onSuccess, reportError, ...dependencies]);

  const reset = useCallback(() => {
    setState({
      data: null,
      loading: false,
      error: null,
      retryCount: 0
    });
  }, []);

  return {
    ...state,
    execute: executeOperation,
    reset
  };
}

// Async error boundary for handling promise rejections
function AsyncErrorBoundary({ children, fallback }) {
  const [asyncError, setAsyncError] = useState(null);
  const { reportError } = useErrorManagement();

  useEffect(() => {
    const handleUnhandledRejection = (event) => {
      setAsyncError(event.reason);
      reportError(event.reason, {
        type: 'unhandled-promise-rejection',
        source: 'async-error-boundary'
      });
      event.preventDefault();
    };

    window.addEventListener('unhandledrejection', handleUnhandledRejection);

    return () => {
      window.removeEventListener('unhandledrejection', handleUnhandledRejection);
    };
  }, [reportError]);

  const resetAsyncError = useCallback(() => {
    setAsyncError(null);
  }, []);

  if (asyncError) {
    if (fallback) {
      return fallback({ error: asyncError, reset: resetAsyncError });
    }

    return (
      <div className="async-error-fallback">
        <h3>Async Operation Failed</h3>
        <p>An asynchronous operation encountered an error.</p>
        <button onClick={resetAsyncError}>Continue</button>
        <details>
          <summary>Error Details</summary>
          <pre>{asyncError.message}</pre>
        </details>
      </div>
    );
  }

  return children;
}

// Practice session async operations
function usePracticeSessionOperations(sessionId) {
  const api = useService('apiClient');
  const { reportError } = useErrorManagement();

  // Load session data with error handling
  const loadSession = useAsyncOperation(
    async (id) => {
      const session = await api.getSession(id);
      return session;
    },
    {
      retries: 2,
      timeout: 10000,
      onError: (error, errorId) => {
        console.error('Failed to load session:', error);
      }
    }
  );

  // Save session progress with retry logic
  const saveProgress = useAsyncOperation(
    async (progressData) => {
      const result = await api.saveSessionProgress(sessionId, progressData);
      return result;
    },
    {
      retries: 5, // More retries for save operations
      retryDelay: 500,
      onError: (error, errorId) => {
        // Show user notification for save failures
        showNotification('Failed to save progress', 'error');
      },
      onSuccess: (data) => {
        showNotification('Progress saved', 'success');
      }
    }
  );

  // Upload audio recording with progress tracking
  const uploadRecording = useAsyncOperation(
    async (audioBlob, onProgress) => {
      const formData = new FormData();
      formData.append('audio', audioBlob);
      formData.append('sessionId', sessionId);

      const result = await api.uploadRecording(formData, {
        onUploadProgress: onProgress
      });

      return result;
    },
    {
      retries: 3,
      timeout: 60000, // Longer timeout for uploads
      onError: (error, errorId) => {
        if (error.name === 'NetworkError') {
          showNotification('Check your internet connection and try again', 'warning');
        } else {
          showNotification('Failed to upload recording', 'error');
        }
      }
    }
  );

  return {
    loadSession,
    saveProgress,
    uploadRecording
  };
}

// Component using async error patterns
function PracticeSessionDashboard({ sessionId }) {
  const { loadSession, saveProgress } = usePracticeSessionOperations(sessionId);
  const [autoSaveEnabled, setAutoSaveEnabled] = useState(true);

  // Load session on mount
  useEffect(() => {
    loadSession.execute(sessionId);
  }, [sessionId, loadSession.execute]);

  // Auto-save with error handling
  useEffect(() => {
    if (!autoSaveEnabled || !loadSession.data) return;

    const autoSaveInterval = setInterval(async () => {
      try {
        await saveProgress.execute({
          sessionId,
          timestamp: Date.now(),
          progressData: getCurrentProgressData()
        });
      } catch (error) {
        // Auto-save errors are handled by the async operation
        // We might want to disable auto-save after multiple failures
        if (saveProgress.retryCount >= 3) {
          setAutoSaveEnabled(false);
          showNotification('Auto-save disabled due to errors', 'warning');
        }
      }
    }, 30000); // Auto-save every 30 seconds

    return () => clearInterval(autoSaveInterval);
  }, [autoSaveEnabled, loadSession.data, saveProgress, sessionId]);

  if (loadSession.loading) {
    return <div>Loading session...</div>;
  }

  if (loadSession.error) {
    return (
      <div className="session-load-error">
        <h3>Failed to load session</h3>
        <p>Retry attempt {loadSession.retryCount}</p>
        <button onClick={() => loadSession.execute(sessionId)}>
          Try Again
        </button>
      </div>
    );
  }

  return (
    <AsyncErrorBoundary
      fallback={({ error, reset }) => (
        <div className="session-async-error">
          <h3>Session Error</h3>
          <p>An error occurred during session operation.</p>
          <button onClick={reset}>Continue</button>
        </div>
      )}
    >
      <div className="practice-session-dashboard">
        {/* Session content */}
        <div className="auto-save-status">
          {saveProgress.loading && <span>Saving...</span>}
          {saveProgress.error && (
            <span className="save-error">
              Save failed (retry {saveProgress.retryCount})
            </span>
          )}
          {!autoSaveEnabled && (
            <button onClick={() => setAutoSaveEnabled(true)}>
              Enable Auto-save
            </button>
          )}
        </div>
      </div>
    </AsyncErrorBoundary>
  );
}
```

Building resilient applications

Advanced error handling patterns create resilient applications that gracefully handle failures while maintaining user experience. By combining error boundaries with context-based error management and sophisticated async error handling, you can build applications that not only survive errors but actively learn from them to improve reliability over time.

Advanced Component Composition Techniques

Advanced composition techniques represent the sophisticated edge of React component architecture. These patterns transform component composition from basic JSX assembly into a refined architectural discipline that enables incredibly flexible systems while maintaining code clarity and maintainability.

These sophisticated patterns may initially appear excessive for straightforward applications. However, when building design systems, component libraries, or applications requiring extensive customization, these techniques become indispensable tools that enable component APIs to scale gracefully with evolving requirements rather than constraining development.

The fundamental principle underlying advanced composition focuses on building systems that grow with your needs rather than against them. When implemented thoughtfully, these patterns make complex customization scenarios feel intuitive and manageable while preserving code quality and developer experience.

Modern React applications benefit from composition patterns that cleanly separate concerns, enable sophisticated customization, and maintain performance while providing excellent developer experience. These patterns often eliminate complex prop drilling requirements, reduce component coupling, and create more testable, maintainable codebases.

Composition Over Configuration Philosophy

Advanced composition patterns favor flexible component assembly over rigid configuration approaches. By creating composable building blocks, you can construct complex interfaces from simple, well-tested components while maintaining the ability to customize behavior at any level of the component hierarchy.

Slot-Based Composition Architecture

Slot-based composition provides a powerful alternative to traditional prop-based customization, enabling components to accept complex, nested content while maintaining clean interfaces and predictable behavior patterns.

```
// Advanced slot system for flexible component composition
function createSlotSystem() {
  // Slot provider for distributing named content
  const SlotProvider = ({ slots, children }) => {
    const slotMap = useMemo(() => {
      const map = new Map();

      // Process slot definitions
      Object.entries(slots || {}).forEach(([name, content]) => {
        map.set(name, content);
      });

      // Extract slots from children
      React.Children.forEach(children, (child) => {
        if (React.isValidElement(child) && child.props.slot) {
          map.set(child.props.slot, child);
        }
      });

      return map;
    }, [slots, children]);

    return (
      <SlotContext.Provider value={slotMap}>
        {children}
      </SlotContext.Provider>
    );
  };

  // Slot consumer for rendering named content
  const Slot = ({ name, fallback, multiple = false, ...props }) => {
    const slots = useContext(SlotContext);
    const content = slots.get(name);

    if (!content && fallback) {
      return typeof fallback === 'function' ? fallback(props) : fallback;
    }

    if (!content) return null;

    // Handle multiple content items
    if (multiple && Array.isArray(content)) {
      return content.map((item, index) => (
        <Fragment key={index}>
          {React.isValidElement(item) ? React.cloneElement(item, props) : item}
        </Fragment>
      ));
    }

    // Single content item
    return React.isValidElement(content)
      ? React.cloneElement(content, props)
      : content;
  };

  return { SlotProvider, Slot };
}

const { SlotProvider, Slot } = createSlotSystem();
const SlotContext = createContext(new Map());

// Practice session card with slot-based composition
function PracticeSessionCard({ session, children, ...slots }) {
  return (
    <SlotProvider slots={slots}>
      <div className="practice-session-card">
        <header className="card-header">
          <div className="session-info">
            <h3 className="session-title">{session.title}</h3>
            <Slot
              name="subtitle"
              fallback={<p className="session-date">{session.date}</p>}
            />
          </div>

          <Slot
            name="headerActions"
            fallback={<DefaultHeaderActions sessionId={session.id} />}
            session={session}
          />
        </header>

        <div className="card-body">
          <Slot
            name="content"
            fallback={<DefaultSessionContent session={session} />}
            session={session}
          />

          <div className="session-metrics">
            <Slot
              name="metrics"
              multiple
              session={session}
            />
          </div>
        </div>

        <footer className="card-footer">
          <Slot
            name="footerActions"
            fallback={<DefaultFooterActions session={session} />}
            session={session}
          />

          <Slot name="extraContent" />
        </footer>

        {children}
      </div>
    </SlotProvider>
  );
}

// Usage with different slot configurations
function PracticeSessionList() {
  return (
    <div className="session-list">
      {sessions.map(session => (
        <PracticeSessionCard
          key={session.id}
          session={session}
          headerActions={<CustomSessionActions session={session} />}
          metrics={[
            <MetricBadge key="duration" value={session.duration} label="Duration" />,
            <MetricBadge key="score" value={session.score} label="Score" />,
            <MetricBadge key="accuracy" value={session.accuracy} label="Accuracy" />
          ]}
        >
          <SessionProgress sessionId={session.id} slot="content" />
          <ShareButton sessionId={session.id} slot="footerActions" />
        </PracticeSessionCard>
      ))}
    </div>
  );
}

// Slot-based modal system
function Modal({ isOpen, onClose, children, ...slots }) {
  if (!isOpen) return null;

  return (
    <div className="modal-overlay" onClick={onClose}>
      <div className="modal-content" onClick={e => e.stopPropagation()}>
```

```
        <SlotProvider slots={slots}>
          <div className="modal-header">
            <Slot name="title" fallback={<h2>Modal</h2>} />
            <Slot
              name="closeButton"
              fallback={<button onClick={onClose}>X</button>}
              onClose={onClose}
            />
          </div>

          <div className="modal-body">
            <Slot name="content" fallback={children} />
          </div>

          <div className="modal-footer">
            <Slot
              name="actions"
              fallback={<button onClick={onClose}>Close</button>}
              onClose={onClose}
            />
          </div>
        </SlotProvider>
      </div>
    </div>
  );
}

// Usage with complex customization
function SessionEditModal({ session, isOpen, onClose, onSave }) {
  return (
    <Modal
      isOpen={isOpen}
      onClose={onClose}
      title={<h2>Edit Practice Session</h2>}
      content={<SessionEditForm session={session} onSave={onSave} />}
      actions={
        <div className="modal-actions">
          <button onClick={onClose}>Cancel</button>
          <button onClick={onSave} className="primary">Save Changes</button>
        </div>
      }
    />
  );
}
```

Builder pattern for complex components

The builder pattern enables the construction of complex components through a fluent, chainable API that provides excellent developer experience and type safety.

```
// Advanced component builder system
class ComponentBuilder {
  constructor(Component) {
    this.Component = Component;
    this.props = {};
    this.children = [];
    this.slots = {};
    this.middlewares = [];
  }

  // Add props with validation
  withProps(props) {
    this.props = { ...this.props, ...props };
    return this;
  }

  // Add children
  withChildren(...children) {
    this.children.push(...children);
    return this;
  }

  // Add named slots
  withSlot(name, content) {
    this.slots[name] = content;
    return this;
  }

  // Add middleware for prop transformation
  withMiddleware(middleware) {
    this.middlewares.push(middleware);
    return this;
  }

  // Conditional prop setting
  when(condition, callback) {
    if (condition) {
      callback(this);
    }
    return this;
  }

  // Build the final component
  build() {
    // Apply middlewares to transform props
    const finalProps = this.middlewares.reduce(
      (props, middleware) => middleware(props),
      { ...this.props, ...this.slots }
    );

    return React.createElement(
      this.Component,
      finalProps,
      ...this.children
    );
  }

  // Create a reusable preset
  preset(name, configuration) {
    const builder = new ComponentBuilder(this.Component);
    configuration(builder);

    // Store preset for reuse
    ComponentBuilder.presets = ComponentBuilder.presets || {};
    ComponentBuilder.presets[name] = configuration;

    return builder;
  }

  // Apply a preset
  applyPreset(name) {
    const preset = ComponentBuilder.presets?.[name];
    if (preset) {
      preset(this);
    }
    return this;
  }
}

// Practice session builder
function createPracticeSessionBuilder() {
  return new ComponentBuilder(PracticeSessionCard);
}

// Middleware for automatic prop enhancement
const withAnalytics = (props) => ({
  ...props,
  onClick: (originalOnClick) => (...args) => {
    // Track click events
    analytics.track('session_card_clicked', { sessionId: props.session?.id });
    if (originalOnClick) originalOnClick(...args);
  }
});

const withAccessibility = (props) => ({
  ...props,
  role: props.role || 'article',
  tabIndex: props.tabIndex || 0,
  'aria-label': props['aria-label'] || `Practice session: ${props.session?.title}`
});

// Usage with builder pattern
function SessionGallery({ sessions, viewMode, userRole }) {
  return (
    <div className="session-gallery">
      {sessions.map(session => {
        const builder = createPracticeSessionBuilder()
          .withProps({ session })
          .withMiddleware(withAnalytics)
          .withMiddleware(withAccessibility)
          .when(viewMode === 'detailed', builder =>
            builder
              .withSlot('metrics', <DetailedMetrics session={session} />)
              .withSlot('content', <SessionAnalysis session={session} />)
          )
          .when(viewMode === 'compact', builder =>
            builder
              .withSlot('content', <CompactSessionInfo session={session} />)
          )
          .when(userRole === 'admin', builder =>
            builder
              .withSlot('headerActions', <AdminActions session={session} />)
          );

        return builder.build();
      })}
    </div>
  );
}

// Form builder for complex forms
class FormBuilder extends ComponentBuilder {
  constructor() {
    super('form');
    this.fields = [];
    this.validation = {};
    this.sections = new Map();
  }

  addField(name, type, options = {}) {
    this.fields.push({ name, type, options });
    return this;
  }

  addSection(name, fields) {
    this.sections.set(name, fields);
    return this;
  }

  withValidation(fieldName, validator) {
    this.validation[fieldName] = validator;
    return this;
  }

  withConditionalField(fieldName, condition, field) {
    const existingField = this.fields.find(f => f.name === fieldName);
    if (existingField) {
      existingField.conditional = { condition, field };
    }
    return this;
  }

  build() {
    return (
      <DynamicForm
        fields={this.fields}
        sections={this.sections}
        validation={this.validation}
        {...this.props}
      />
    );
  }
}

// Practice session form with builder
function createSessionForm(sessionType) {
  return new FormBuilder()
    .withProps({ className: 'practice-session-form' })
    .addField('title', 'text', { required: true, label: 'Session Title' })
    .addField('duration', 'number', { required: true, min: 1, max: 240 })
    .when(sessionType === 'performance', builder =>
      builder
        .addField('piece', 'select', {
          options: availablePieces,
          label: 'Musical Piece'
        })
        .addField('tempo', 'slider', { min: 60, max: 200, default: 120 })
    )
    .when(sessionType === 'technique', builder =>
      builder
        .addField('technique', 'select', {
          options: techniques,
          label: 'Technique Focus'
        })
        .addField('difficulty', 'radio', {
          options: ['Beginner', 'Intermediate', 'Advanced']
        })
    )
    .addField('notes', 'textarea', { optional: true })
    .withValidation('title', (value) =>
      value.length >= 3 ? null : 'Title must be at least 3 characters'
    );
}

// Layout builder for complex layouts
class LayoutBuilder {
  constructor() {
    this.structure = { type: 'container', children: [] };
    this.current = this.structure;
    this.stack = [];
  }

  row(callback) {
    const row = { type: 'row', children: [] };
    this.current.children.push(row);
    this.stack.push(this.current);
    this.current = row;

    if (callback) callback(this);

    this.current = this.stack.pop();
    return this;
  }

  col(size, callback) {
    const col = { type: 'col', size, children: [] };
    this.current.children.push(col);
    this.stack.push(this.current);
    this.current = col;

    if (callback) callback(this);

    this.current = this.stack.pop();
    return this;
  }

  component(Component, props = {}) {
    this.current.children.push({
      type: 'component',
      Component,
      props
    });
    return this;
  }

  build() {
    return <LayoutRenderer structure={this.structure} />;
  }
}

// Layout renderer component
function LayoutRenderer({ structure }) {
  const renderNode = (node, index) => {
    switch (node.type) {
      case 'container':
        return (
          <div key={index} className="layout-container">
            {node.children.map(renderNode)}
          </div>
        );

      case 'row':
        return (
          <div key={index} className="layout-row">
            {node.children.map(renderNode)}
          </div>
        );

      case 'col':
        return (
          <div key={index} className={`layout-col col-${node.size}`}>
            {node.children.map(renderNode)}
          </div>
        );

      case 'component':
        return <node.Component key={index} {...node.props} />;

      default:
        return null;
    }
  };

  return renderNode(structure, 0);
}

// Usage: Complex dashboard layout
function PracticeDashboard({ user, sessions, analytics }) {
  const layout = new LayoutBuilder()
```

```
    .row(row => row
      .col(8, col => col
        .component(WelcomeHeader, { user })
        .row(innerRow => innerRow
          .col(6, col => col
            .component(ActiveSessionCard, { session: sessions.active })
          )
          .col(6, col => col
            .component(QuickStats, { stats: analytics.today })
          )
        )
        .component(RecentSessions, { sessions: sessions.recent })
      )
      .col(4, col => col
        .component(PracticeCalendar, { sessions: sessions.all })
        .component(GoalsWidget, { goals: user.goals })
        .component(AchievementsWidget, { achievements: user.achievements })
      )
    );

  return layout.build();
}
```

Polymorphic component patterns

Polymorphic components provide ultimate flexibility by allowing the underlying element or component type to be changed while maintaining consistent behavior and styling.

```
// Advanced polymorphic component implementation
function createPolymorphicComponent(defaultComponent = 'div') {
  const PolymorphicComponent = React.forwardRef(
    ({ as: Component = defaultComponent, children, ...props }, ref) => {
      return (
        <Component ref={ref} {...props}>
          {children}
        </Component>
      );
    }
  );

  // Add display name for debugging
  PolymorphicComponent.displayName = 'PolymorphicComponent';

  return PolymorphicComponent;
}
```

```

// Base polymorphic text component
const Text = React.forwardRef(({
  as = 'span',
  variant = 'body',
  size = 'medium',
  weight = 'normal',
  color = 'inherit',
  children,
  className,
  ...props
}, ref) => {
  const Component = as;

  const textClasses = classNames(
    'text',
    `text--${variant}`,
    `text--${size}`,
    `text--${weight}`,
    `text--${color}`,
    className
  );

  return (
    <Component ref={ref} className={textClasses} {...props}>
      {children}
    </Component>
  );
});

// Polymorphic button component with advanced features
const Button = React.forwardRef(({
  as = 'button',
  variant = 'primary',
  size = 'medium',
  loading = false,
  disabled = false,
  leftIcon,
  rightIcon,
  children,
  onClick,
  className,
  ...props
}, ref) => {
  const Component = as;
  const isDisabled = disabled || loading;

  const buttonClasses = classNames(
    'button',
    `button--${variant}`,
    `button--${size}`,
    {
      'button--loading': loading,
      'button--disabled': isDisabled
    },
    className
  );

  const handleClick = useCallback((event) => {
    if (isDisabled) {
      event.preventDefault();
      return;
    }

    if (onClick) {
      onClick(event);
    }
  }, [onClick, isDisabled]);

  return (
    <Component
      ref={ref}
      className={buttonClasses}
      onClick={handleClick}
      disabled={Component === 'button' ? isDisabled : undefined}
      aria-disabled={isDisabled}
      {...props}
    >
      {leftIcon && (
        <span className="button__icon button__icon--left">
          {leftIcon}
        </span>
      )}

      <span className="button__content">
        {loading ? <Spinner size="small" /> : children}
      </span>

      {rightIcon && (
        <span className="button__icon button__icon--right">
          {rightIcon}
        </span>
      )}
    </Component>
  );
});

// Polymorphic card component
const Card = React.forwardRef(({
  as = 'div',
  variant = 'default',
  padding = 'medium',
  shadow = true,
  bordered = false,
  clickable = false,
  children,
  className,
  onClick,
  ...props
}, ref) => {
  const Component = as;

  const cardClasses = classNames(
    'card',
    `card--${variant}`,
    `card--padding-${padding}`,
    {
      'card--shadow': shadow,
      'card--bordered': bordered,
      'card--clickable': clickable
    },
    className
  );

  return (
    <Component
      ref={ref}
      className={cardClasses}
      onClick={onClick}
      role={clickable ? 'button' : undefined}
      tabIndex={clickable ? 0 : undefined}
      {...props}
    >
      {children}
    </Component>
  );
});

// Practice session components using polymorphic patterns
function SessionActionButton({ session, action, ...props }) {
  // Dynamically choose component based on action type
  const getButtonProps = () => {
    switch (action.type) {
      case 'external':
        return {
          as: 'a',
          href: action.url,
          target: '_blank',
          rel: 'noopener noreferrer'
        };

      case 'route':
        return {
          as: Link,
          to: action.path
        };

      case 'download':
        return {
          as: 'a',
          href: action.downloadUrl,
          download: action.filename
        };

      default:
        return {
          as: 'button',
          onClick: action.handler
        };
    }
  };

  return (
    <Button
      {...getButtonProps()}
      variant={action.variant || 'secondary'}
      leftIcon={action.icon}
      {...props}
    >
      {action.label}
    </Button>
  );
}

// Polymorphic metric display
function MetricDisplay({
  metric,
  as = 'div',
  interactive = false,
  size = 'medium',
  ...props
}) {
  const baseProps = {
    className: `metric metric--${size}`,
    role: interactive ? 'button' : undefined,
    tabIndex: interactive ? 0 : undefined
  };

  if (interactive) {
    return (
      <Card
        as={as}
        clickable
        padding="small"
        {...baseProps}
        {...props}
      >
        <MetricContent metric={metric} />
      </Card>
    );
  }

  return (
    <Text
      as={as}
      variant="metric"
      {...baseProps}
      {...props}
    >
      <MetricContent metric={metric} />
    </Text>
  );
}

// Adaptive session list item
function SessionListItem({ session, viewMode, actions = [] }) {
  const getItemComponent = () => {
    switch (viewMode) {
      case 'card':
        return {
          as: Card,
          variant: 'elevated',
          clickable: true
        };

      case 'row':
        return {
          as: 'tr',
          className: 'session-row'
        };

      case 'list':
        return {
          as: 'li',
          className: 'session-list-item'
        };

      default:
        return {
          as: 'div',
          className: 'session-item'
        };
    }
  };

  const itemProps = getItemComponent();

  return (
    <Card {...itemProps}>
      <div className="session-header">
        <Text as="h3" variant="heading" size="small">
          {session.title}
        </Text>
        <Text variant="caption" color="muted">
          {session.date}
        </Text>
      </div>

      <div className="session-content">
        <SessionMetrics session={session} viewMode={viewMode} />
      </div>

      {actions.length > 0 && (
        <div className="session-actions">
          {actions.map((action, index) => (
            <SessionActionButton
              key={index}
              session={session}
              action={action}
              size="small"
            />
          ))}
        </div>
      )}
    </Card>
  );
}

// Usage with different contexts
function PracticeSessionsView({ sessions, viewMode }) {
  const containerProps = {
    card: { as: 'div', className: 'sessions-grid' },
    row: { as: 'table', className: 'sessions-table' },
    list: { as: 'ul', className: 'sessions-list' }
  }[viewMode] || { as: 'div' };

  return (
    <div {...containerProps}>
      {sessions.map(session => (
        <SessionListItem
          key={session.id}
          session={session}
          viewMode={viewMode}
          actions={[
            { type: 'route', path: `/sessions/${session.id}`, label: 'View' },
            { type: 'button', handler: () => editSession(session.id), label: 'Edit' }
          ]}
        />
      ))}
    </div>
  );
}
```

Advanced composition techniques provide the foundation for building truly flexible and maintainable component systems. By leveraging slots, builders, and polymorphic patterns, you can create components that adapt to diverse requirements while maintaining consistency and performance. These patterns enable component libraries that feel native to React while providing the flexibility typically associated with more complex frameworks.