# **Hooks Documentation**

## **Overview**

This document provides comprehensive documentation for all hooks used in the Chat Application project, including React hooks (both built-in and custom) and Git hooks for development workflow automation. It covers implementation patterns, best practices, testing strategies, and troubleshooting guidelines.

## **React Hooks**

### **Built-in React Hooks Usage**

#### **useState Hook**

Used for managing local component state throughout the application.

// Basic state management  
const [input, setInput] = useState<string>("");  
const [isLoading, setIsLoading] = useState<boolean>(false);  
  
// Complex state with proper typing  
interface ChatMessage {  
 role: 'user' | 'assistant';  
 content: string;  
 timestamp?: Date;  
}  
  
const [messages, setMessages] = useState<ChatMessage[]>([]);  
  
// State with initial value from props  
const [selectedTheme, setSelectedTheme] = useState(  
 () => props.initialTheme || 'light'  
);

**Best Practices:**

* Always provide TypeScript types for state
* Use functional updates when state depends on previous state
* Initialize with appropriate default values
* Group related state when it makes sense

#### **useEffect Hook**

Used for side effects, API calls, and lifecycle management.

// Effect with dependency array  
useEffect(() => {  
 const getToken = async () => {  
 if (accounts.length > 0) {  
 try {  
 const response = await instance.acquireTokenSilent({  
 ...loginRequest,  
 account: accounts[0]  
 });  
 setAccessToken(response.accessToken);  
 } catch (error) {  
 console.error("Failed to get access token:", error);  
 }  
 }  
 };  
  
 getToken();  
}, [instance, accounts]); // Dependencies  
  
// Effect with cleanup  
useEffect(() => {  
 const interval = setInterval(() => {  
 // Periodic task  
 }, 5000);  
  
 return () => clearInterval(interval); // Cleanup  
}, []);  
  
// Effect for window resize handling  
useEffect(() => {  
 const handleResize = () => {  
 setWindowWidth(window.innerWidth);  
 };  
  
 window.addEventListener('resize', handleResize);  
 return () => window.removeEventListener('resize', handleResize);  
}, []);

**Best Practices:**

* Always include necessary dependencies in dependency array
* Use cleanup functions to prevent memory leaks
* Separate concerns into multiple useEffect hooks
* Use ESLint exhaustive-deps rule to catch missing dependencies

#### **useCallback Hook**

Used for memoizing callback functions to prevent unnecessary re-renders.

// Memoized event handler  
const handleSubmit = useCallback(async (message: string) => {  
 if (!accessToken) return;  
   
 setIsLoading(true);  
 try {  
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${accessToken}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: message })  
 });  
   
 const data = await response.json();  
 setMessages(prev => [...prev, { role: 'assistant', content: data.content }]);  
 } catch (error) {  
 console.error('API Error:', error);  
 } finally {  
 setIsLoading(false);  
 }  
}, [accessToken]); // Dependencies  
  
// Memoized function passed to child components  
const handleMessageSelect = useCallback((messageId: string) => {  
 setSelectedMessage(messageId);  
 onMessageSelect?.(messageId);  
}, [onMessageSelect]);

#### **useMemo Hook**

Used for memoizing expensive calculations.

// Memoized computed value  
const filteredMessages = useMemo(() => {  
 return messages.filter(message =>   
 message.content.toLowerCase().includes(searchTerm.toLowerCase())  
 );  
}, [messages, searchTerm]);  
  
// Memoized configuration object  
const msalConfig = useMemo(() => ({  
 auth: {  
 clientId: process.env.AZURE\_CLIENT\_ID!,  
 authority: `https://login.microsoftonline.com/${process.env.AZURE\_TENANT\_ID}`,  
 },  
 cache: {  
 cacheLocation: "sessionStorage" as const,  
 storeAuthStateInCookie: false,  
 }  
}), []);  
  
// Memoized component props  
const chatProps = useMemo(() => ({  
 messages: filteredMessages,  
 onSend: handleSubmit,  
 isLoading,  
 placeholder: 'Type your message...'  
}), [filteredMessages, handleSubmit, isLoading]);

#### **useContext Hook**

Used for accessing React context values.

// Theme context usage  
const theme = useContext(ThemeContext);  
  
// Authentication context  
const { user, signIn, signOut } = useContext(AuthContext);  
  
// Custom context with proper typing  
interface ChatContextType {  
 messages: ChatMessage[];  
 sendMessage: (content: string) => Promise<void>;  
 isLoading: boolean;  
 error: string | null;  
}  
  
const ChatContext = createContext<ChatContextType | undefined>(undefined);  
  
// Hook to use chat context with error handling  
function useChatContext() {  
 const context = useContext(ChatContext);  
 if (!context) {  
 throw new Error('useChatContext must be used within a ChatProvider');  
 }  
 return context;  
}

#### **useReducer Hook**

Used for complex state logic with multiple related state values.

// Chat state reducer  
interface ChatState {  
 messages: ChatMessage[];  
 isLoading: boolean;  
 error: string | null;  
 currentUser: string | null;  
}  
  
type ChatAction =   
 | { type: 'SET\_LOADING'; payload: boolean }  
 | { type: 'ADD\_MESSAGE'; payload: ChatMessage }  
 | { type: 'SET\_ERROR'; payload: string | null }  
 | { type: 'SET\_USER'; payload: string | null }  
 | { type: 'CLEAR\_MESSAGES' };  
  
const chatReducer = (state: ChatState, action: ChatAction): ChatState => {  
 switch (action.type) {  
 case 'SET\_LOADING':  
 return { ...state, isLoading: action.payload };  
 case 'ADD\_MESSAGE':  
 return {   
 ...state,   
 messages: [...state.messages, action.payload],  
 error: null  
 };  
 case 'SET\_ERROR':  
 return { ...state, error: action.payload, isLoading: false };  
 case 'SET\_USER':  
 return { ...state, currentUser: action.payload };  
 case 'CLEAR\_MESSAGES':  
 return { ...state, messages: [], error: null };  
 default:  
 return state;  
 }  
};  
  
// Using the reducer  
const [chatState, dispatch] = useReducer(chatReducer, {  
 messages: [],  
 isLoading: false,  
 error: null,  
 currentUser: null  
});

#### **useRef Hook**

Used for accessing DOM elements and storing mutable values.

// DOM element reference  
const inputRef = useRef<HTMLInputElement>(null);  
const chatContainerRef = useRef<HTMLDivElement>(null);  
  
// Focus management  
useEffect(() => {  
 if (inputRef.current) {  
 inputRef.current.focus();  
 }  
}, []);  
  
// Auto-scroll to bottom  
useEffect(() => {  
 if (chatContainerRef.current) {  
 chatContainerRef.current.scrollTop = chatContainerRef.current.scrollHeight;  
 }  
}, [messages]);  
  
// Mutable value that doesn't trigger re-renders  
const requestIdRef = useRef<number>(0);  
  
const makeRequest = useCallback(async () => {  
 const requestId = ++requestIdRef.current;  
   
 const response = await fetch('/api/completions');  
   
 // Only process if this is still the latest request  
 if (requestId === requestIdRef.current) {  
 // Process response  
 }  
}, []);

## **Custom Hooks**

### **Real-World Implementation Examples**

The following examples are based on the actual implementation in this chat application project.

#### **Current useAccessToken Implementation**

**File**: app/hooks/useAccessToken.ts

"use client";  
  
import { useMsal } from "@azure/msal-react";  
import { useEffect, useState } from "react";  
import { loginRequest } from "../utils/msal-config";  
  
export function useAccessToken() {  
 const { instance, accounts } = useMsal();  
 const [accessToken, setAccessToken] = useState<string | null>(null);  
  
 useEffect(() => {  
 const getToken = async () => {  
 if (accounts.length > 0) {  
 try {  
 const response = await instance.acquireTokenSilent({  
 ...loginRequest,  
 account: accounts[0]  
 });  
 setAccessToken(response.accessToken);  
 } catch (error) {  
 // If silent token acquisition fails, try popup  
 try {  
 const response = await instance.acquireTokenPopup(loginRequest);  
 setAccessToken(response.accessToken);  
 } catch (error) {  
 console.error("Failed to get access token:", error);  
 }  
 }  
 }  
 };  
  
 getToken();  
 }, [instance, accounts]);  
  
 return accessToken;  
}

**Current Usage in ChatInterface**:

// In app/components/ChatInterface.tsx  
const accessToken = useAccessToken();  
  
const handleSubmit = async (e: React.FormEvent) => {  
 e.preventDefault();  
 if (!input.trim() || !accessToken) return;  
  
 // API call with token  
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${accessToken}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: userMessage })  
 });  
};

### **useAccessToken Hook**

**Purpose:** Manages Azure AD access token acquisition and refresh.

**Location:** app/hooks/useAccessToken.ts

"use client";  
  
import { useMsal } from "@azure/msal-react";  
import { useEffect, useState, useCallback } from "react";  
import { loginRequest } from "../utils/msal-config";  
  
interface UseAccessTokenReturn {  
 accessToken: string | null;  
 isLoading: boolean;  
 error: string | null;  
 refreshToken: () => Promise<void>;  
}  
  
export function useAccessToken(): UseAccessTokenReturn {  
 const { instance, accounts } = useMsal();  
 const [accessToken, setAccessToken] = useState<string | null>(null);  
 const [isLoading, setIsLoading] = useState<boolean>(true);  
 const [error, setError] = useState<string | null>(null);  
  
 const acquireToken = useCallback(async () => {  
 if (accounts.length === 0) {  
 setAccessToken(null);  
 setIsLoading(false);  
 setError("No user account found");  
 return;  
 }  
  
 setIsLoading(true);  
 setError(null);  
  
 try {  
 // Try silent token acquisition first  
 const response = await instance.acquireTokenSilent({  
 ...loginRequest,  
 account: accounts[0]  
 });  
   
 setAccessToken(response.accessToken);  
 } catch (silentError) {  
 console.warn("Silent token acquisition failed:", silentError);  
   
 try {  
 // Fallback to popup  
 const response = await instance.acquireTokenPopup(loginRequest);  
 setAccessToken(response.accessToken);  
 } catch (popupError) {  
 console.error("Token acquisition failed:", popupError);  
 setError("Failed to acquire access token");  
 setAccessToken(null);  
 }  
 } finally {  
 setIsLoading(false);  
 }  
 }, [instance, accounts]);  
  
 const refreshToken = useCallback(async () => {  
 await acquireToken();  
 }, [acquireToken]);  
  
 useEffect(() => {  
 acquireToken();  
 }, [acquireToken]);  
  
 return { accessToken, isLoading, error, refreshToken };  
}

**Usage:**

function MyComponent() {  
 const { accessToken, isLoading, error, refreshToken } = useAccessToken();  
  
 if (isLoading) return <div>Loading...</div>;  
 if (error) return <div>Error: {error}</div>;  
 if (!accessToken) return <div>Please sign in</div>;  
  
 return <div>Token: {accessToken.substring(0, 10)}...</div>;  
}

### **useChatMessages Hook**

**Purpose:** Manages chat messages state and operations.

"use client";  
  
import { useState, useCallback, useMemo } from 'react';  
import { ChatMessage, MessageRole } from '@/app/types';  
  
interface UseChatMessagesReturn {  
 messages: ChatMessage[];  
 addMessage: (content: string, role: MessageRole) => void;  
 updateMessage: (id: string, content: string) => void;  
 deleteMessage: (id: string) => void;  
 clearMessages: () => void;  
 getLastUserMessage: () => ChatMessage | undefined;  
 messageCount: number;  
}  
  
export function useChatMessages(): UseChatMessagesReturn {  
 const [messages, setMessages] = useState<ChatMessage[]>([]);  
  
 const addMessage = useCallback((content: string, role: MessageRole) => {  
 const newMessage: ChatMessage = {  
 id: crypto.randomUUID(),  
 content,  
 role,  
 timestamp: new Date()  
 };  
  
 setMessages(prev => [...prev, newMessage]);  
 }, []);  
  
 const updateMessage = useCallback((id: string, content: string) => {  
 setMessages(prev => prev.map(msg =>   
 msg.id === id ? { ...msg, content } : msg  
 ));  
 }, []);  
  
 const deleteMessage = useCallback((id: string) => {  
 setMessages(prev => prev.filter(msg => msg.id !== id));  
 }, []);  
  
 const clearMessages = useCallback(() => {  
 setMessages([]);  
 }, []);  
  
 const getLastUserMessage = useCallback(() => {  
 return messages.filter(msg => msg.role === 'user').pop();  
 }, [messages]);  
  
 const messageCount = useMemo(() => messages.length, [messages]);  
  
 return {  
 messages,  
 addMessage,  
 updateMessage,  
 deleteMessage,  
 clearMessages,  
 getLastUserMessage,  
 messageCount  
 };  
}

### **useApiRequest Hook**

**Purpose:** Generic hook for making API requests with loading states and error handling.

"use client";  
  
import { useState, useCallback } from 'react';  
  
interface UseApiRequestOptions<T> {  
 onSuccess?: (data: T) => void;  
 onError?: (error: Error) => void;  
 transform?: (data: any) => T;  
}  
  
interface UseApiRequestReturn<T> {  
 data: T | null;  
 isLoading: boolean;  
 error: string | null;  
 execute: (...args: any[]) => Promise<T | null>;  
 reset: () => void;  
}  
  
export function useApiRequest<T = any>(  
 apiFunction: (...args: any[]) => Promise<any>,  
 options: UseApiRequestOptions<T> = {}  
): UseApiRequestReturn<T> {  
 const [data, setData] = useState<T | null>(null);  
 const [isLoading, setIsLoading] = useState(false);  
 const [error, setError] = useState<string | null>(null);  
  
 const execute = useCallback(async (...args: any[]): Promise<T | null> => {  
 setIsLoading(true);  
 setError(null);  
  
 try {  
 const response = await apiFunction(...args);  
 const transformedData = options.transform ? options.transform(response) : response;  
   
 setData(transformedData);  
 options.onSuccess?.(transformedData);  
   
 return transformedData;  
 } catch (err) {  
 const errorMessage = err instanceof Error ? err.message : 'An unknown error occurred';  
 setError(errorMessage);  
 options.onError?.(err instanceof Error ? err : new Error(errorMessage));  
   
 return null;  
 } finally {  
 setIsLoading(false);  
 }  
 }, [apiFunction, options]);  
  
 const reset = useCallback(() => {  
 setData(null);  
 setError(null);  
 setIsLoading(false);  
 }, []);  
  
 return { data, isLoading, error, execute, reset };  
}

**Usage:**

// API function  
async function sendChatMessage(message: string, token: string) {  
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${token}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: message })  
 });  
  
 if (!response.ok) {  
 throw new Error('Failed to send message');  
 }  
  
 return response.json();  
}  
  
// Component usage  
function ChatComponent() {  
 const { accessToken } = useAccessToken();  
 const { data, isLoading, error, execute } = useApiRequest(  
 sendChatMessage,  
 {  
 onSuccess: (data) => console.log('Message sent successfully:', data),  
 onError: (error) => console.error('Failed to send message:', error),  
 transform: (response) => response.data.content  
 }  
 );  
  
 const handleSend = async (message: string) => {  
 if (accessToken) {  
 await execute(message, accessToken);  
 }  
 };  
  
 return (  
 <div>  
 {isLoading && <div>Sending...</div>}  
 {error && <div>Error: {error}</div>}  
 <button onClick={() => handleSend('Hello')}>Send Message</button>  
 </div>  
 );  
}

### **useLocalStorage Hook**

**Purpose:** Sync state with localStorage with proper error handling.

"use client";  
  
import { useState, useEffect, useCallback } from 'react';  
  
export function useLocalStorage<T>(  
 key: string,  
 initialValue: T  
): [T, (value: T | ((prev: T) => T)) => void, () => void] {  
 // Get initial value from localStorage or use provided initial value  
 const [storedValue, setStoredValue] = useState<T>(() => {  
 if (typeof window === 'undefined') {  
 return initialValue;  
 }  
  
 try {  
 const item = window.localStorage.getItem(key);  
 return item ? JSON.parse(item) : initialValue;  
 } catch (error) {  
 console.error(`Error reading localStorage key "${key}":`, error);  
 return initialValue;  
 }  
 });  
  
 // Update localStorage when state changes  
 const setValue = useCallback((value: T | ((prev: T) => T)) => {  
 try {  
 setStoredValue(prev => {  
 const newValue = value instanceof Function ? value(prev) : value;  
   
 if (typeof window !== 'undefined') {  
 window.localStorage.setItem(key, JSON.stringify(newValue));  
 }  
   
 return newValue;  
 });  
 } catch (error) {  
 console.error(`Error setting localStorage key "${key}":`, error);  
 }  
 }, [key]);  
  
 // Remove from localStorage  
 const removeValue = useCallback(() => {  
 try {  
 setStoredValue(initialValue);  
 if (typeof window !== 'undefined') {  
 window.localStorage.removeItem(key);  
 }  
 } catch (error) {  
 console.error(`Error removing localStorage key "${key}":`, error);  
 }  
 }, [key, initialValue]);  
  
 return [storedValue, setValue, removeValue];  
}

### **useDebounce Hook**

**Purpose:** Debounce a value to limit API calls or expensive operations.

"use client";  
  
import { useState, useEffect } from 'react';  
  
export function useDebounce<T>(value: T, delay: number): T {  
 const [debouncedValue, setDebouncedValue] = useState<T>(value);  
  
 useEffect(() => {  
 const handler = setTimeout(() => {  
 setDebouncedValue(value);  
 }, delay);  
  
 return () => {  
 clearTimeout(handler);  
 };  
 }, [value, delay]);  
  
 return debouncedValue;  
}

**Usage:**

function SearchComponent() {  
 const [searchTerm, setSearchTerm] = useState('');  
 const debouncedSearchTerm = useDebounce(searchTerm, 300);  
  
 useEffect(() => {  
 if (debouncedSearchTerm) {  
 // Perform search API call  
 searchMessages(debouncedSearchTerm);  
 }  
 }, [debouncedSearchTerm]);  
  
 return (  
 <input  
 value={searchTerm}  
 onChange={(e) => setSearchTerm(e.target.value)}  
 placeholder="Search messages..."  
 />  
 );  
}

### **Recommended Additional Hooks for Chat Application**

#### **Enhanced useChat Hook**

This improved version extracts all chat logic from the component:

// app/hooks/useChat.ts  
"use client";  
  
import { useState, useCallback } from "react";  
import { useAccessToken } from "./useAccessToken";  
  
interface Message {  
 role: 'user' | 'assistant';  
 content: string;  
 timestamp: number;  
 id: string;  
}  
  
interface UseChatReturn {  
 messages: Message[];  
 input: string;  
 isLoading: boolean;  
 error: string | null;  
 setInput: (value: string) => void;  
 sendMessage: () => Promise<void>;  
 clearMessages: () => void;  
}  
  
export function useChat(): UseChatReturn {  
 const [input, setInput] = useState("");  
 const [messages, setMessages] = useState<Message[]>([]);  
 const [isLoading, setIsLoading] = useState(false);  
 const [error, setError] = useState<string | null>(null);  
 const accessToken = useAccessToken();  
  
 const sendMessage = useCallback(async () => {  
 if (!input.trim() || !accessToken || isLoading) return;  
  
 const userMessage: Message = {  
 role: 'user',  
 content: input.trim(),  
 timestamp: Date.now(),  
 id: crypto.randomUUID()  
 };  
  
 setInput("");  
 setIsLoading(true);  
 setError(null);  
 setMessages(prev => [...prev, userMessage]);  
  
 try {  
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${accessToken}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: userMessage.content })  
 });  
  
 const data = await response.json();  
  
 if (!response.ok) {  
 throw new Error(data.error || 'Failed to get response');  
 }  
  
 const assistantMessage: Message = {  
 role: 'assistant',  
 content: data.data?.content || data.data || 'No response content found',  
 timestamp: Date.now(),  
 id: crypto.randomUUID()  
 };  
  
 setMessages(prev => [...prev, assistantMessage]);  
 } catch (err: any) {  
 const errorMessage = `Error: ${err.message || 'An unexpected error occurred'}`;  
 setError(errorMessage);  
 } finally {  
 setIsLoading(false);  
 }  
 }, [input, accessToken, isLoading]);  
  
 const clearMessages = useCallback(() => {  
 setMessages([]);  
 setError(null);  
 }, []);  
  
 return {  
 messages,  
 input,  
 isLoading,  
 error,  
 setInput,  
 sendMessage,  
 clearMessages  
 };  
}

#### **useLocalStorage Hook for Message Persistence**

// app/hooks/useLocalStorage.ts  
"use client";  
  
import { useState, useEffect } from "react";  
  
export function useLocalStorage<T>(key: string, initialValue: T) {  
 const [storedValue, setStoredValue] = useState<T>(() => {  
 if (typeof window === "undefined") {  
 return initialValue;  
 }  
  
 try {  
 const item = window.localStorage.getItem(key);  
 return item ? JSON.parse(item) : initialValue;  
 } catch (error) {  
 console.warn(`Error reading localStorage key "${key}":`, error);  
 return initialValue;  
 }  
 });  
  
 const setValue = (value: T | ((val: T) => T)) => {  
 try {  
 const valueToStore = value instanceof Function ? value(storedValue) : value;  
 setStoredValue(valueToStore);  
   
 if (typeof window !== "undefined") {  
 window.localStorage.setItem(key, JSON.stringify(valueToStore));  
 }  
 } catch (error) {  
 console.warn(`Error setting localStorage key "${key}":`, error);  
 }  
 };  
  
 return [storedValue, setValue] as const;  
}

#### **useDebounce Hook for Input Optimization**

// app/hooks/useDebounce.ts  
"use client";  
  
import { useState, useEffect } from "react";  
  
export function useDebounce<T>(value: T, delay: number): T {  
 const [debouncedValue, setDebouncedValue] = useState<T>(value);  
  
 useEffect(() => {  
 const handler = setTimeout(() => {  
 setDebouncedValue(value);  
 }, delay);  
  
 return () => {  
 clearTimeout(handler);  
 };  
 }, [value, delay]);  
  
 return debouncedValue;  
}

### **Performance Best Practices**

#### **Memoization in Custom Hooks**

// app/hooks/useOptimizedChat.ts  
"use client";  
  
import { useState, useCallback, useMemo } from "react";  
import { useAccessToken } from "./useAccessToken";  
  
export function useOptimizedChat() {  
 const [messages, setMessages] = useState<Message[]>([]);  
 const [input, setInput] = useState("");  
 const [isLoading, setIsLoading] = useState(false);  
 const accessToken = useAccessToken();  
  
 // Memoize message operations to prevent unnecessary re-renders  
 const addMessage = useCallback((message: Message) => {  
 setMessages(prev => [...prev, message]);  
 }, []);  
  
 const clearMessages = useCallback(() => {  
 setMessages([]);  
 }, []);  
  
 // Memoize derived state  
 const messageCount = useMemo(() => messages.length, [messages]);  
 const hasMessages = useMemo(() => messages.length > 0, [messages]);  
 const lastMessage = useMemo(() => messages[messages.length - 1], [messages]);  
  
 // Memoize API call function  
 const sendMessage = useCallback(async () => {  
 if (!input.trim() || !accessToken || isLoading) return;  
  
 const userMessage: Message = {  
 role: 'user',  
 content: input.trim(),  
 timestamp: Date.now(),  
 id: crypto.randomUUID()  
 };  
  
 setInput("");  
 setIsLoading(true);  
 addMessage(userMessage);  
  
 try {  
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${accessToken}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: userMessage.content })  
 });  
  
 const data = await response.json();  
  
 if (!response.ok) {  
 throw new Error(data.error || 'Failed to get response');  
 }  
  
 const assistantMessage: Message = {  
 role: 'assistant',  
 content: data.data?.content || data.data || 'No response',  
 timestamp: Date.now(),  
 id: crypto.randomUUID()  
 };  
  
 addMessage(assistantMessage);  
 } catch (error: any) {  
 const errorMessage: Message = {  
 role: 'assistant',  
 content: `Error: ${error.message}`,  
 timestamp: Date.now(),  
 id: crypto.randomUUID()  
 };  
 addMessage(errorMessage);  
 } finally {  
 setIsLoading(false);  
 }  
 }, [input, accessToken, isLoading, addMessage]);  
  
 return {  
 messages,  
 input,  
 isLoading,  
 messageCount,  
 hasMessages,  
 lastMessage,  
 setInput,  
 sendMessage,  
 clearMessages,  
 addMessage  
 };  
}

#### **Optimizing Hook Dependencies**

// Good: Optimize dependencies by extracting values  
const user = { id: 1, name: 'John', email: '[john@example.com](mailto:john@example.com)' };  
  
useEffect(() => {  
 if (user.id) {  
 fetchUserData(user.id);  
 }  
}, [user.id]); // Only depend on user.id, not entire user object  
  
// Good: Memoize objects that are dependencies  
const apiConfig = useMemo(() => ({  
 baseURL: process.env.NEXT\_PUBLIC\_API\_URL,  
 timeout: 5000,  
 headers: { 'Content-Type': 'application/json' }  
}), []);  
  
useEffect(() => {  
 initializeAPI(apiConfig);  
}, [apiConfig]);

#### **Preventing Unnecessary Re-renders**

// app/hooks/useStableCallback.ts  
"use client";  
  
import { useRef, useCallback } from "react";  
  
export function useStableCallback<T extends (...args: any[]) => any>(callback: T): T {  
 const callbackRef = useRef(callback);  
  
 // Update the ref when callback changes  
 callbackRef.current = callback;  
  
 // Return a stable callback that always calls the latest version  
 return useCallback((...args) => {  
 return callbackRef.current(...args);  
 }, []) as T;  
}  
  
// Usage in components  
function ChatComponent() {  
 const [messages, setMessages] = useState([]);  
  
 // This callback is stable and won't cause re-renders of child components  
 const onMessageReceived = useStableCallback((message: Message) => {  
 setMessages(prev => [...prev, message]);  
 // Some other logic that might change...  
 });  
  
 return (  
 <MessageList   
 messages={messages}   
 onMessageReceived={onMessageReceived} // Stable reference  
 />  
 );  
}

### **Advanced Hook Patterns**

#### **Compound Hook Pattern**

// app/hooks/useChatWithNotifications.ts  
"use client";  
  
import { useChat } from "./useChat";  
import { useNotifications } from "./useNotifications";  
import { useEffect } from "react";  
  
export function useChatWithNotifications() {  
 const chat = useChat();  
 const notifications = useNotifications();  
  
 // Show notification for new messages  
 useEffect(() => {  
 const lastMessage = chat.messages[chat.messages.length - 1];  
 if (lastMessage && lastMessage.role === 'assistant') {  
 notifications.show({  
 title: 'New message',  
 message: lastMessage.content.substring(0, 50) + '...',  
 type: 'info'  
 });  
 }  
 }, [chat.messages, notifications]);  
  
 return {  
 ...chat,  
 notifications: notifications.notifications,  
 clearNotifications: notifications.clear  
 };  
}

#### **Context + Hook Pattern**

// app/contexts/ChatContext.tsx  
"use client";  
  
import { createContext, useContext, ReactNode } from "react";  
import { useChat } from "@/app/hooks/useChat";  
  
interface ChatContextType {  
 messages: Message[];  
 input: string;  
 isLoading: boolean;  
 error: string | null;  
 setInput: (value: string) => void;  
 sendMessage: () => Promise<void>;  
 clearMessages: () => void;  
}  
  
const ChatContext = createContext<ChatContextType | undefined>(undefined);  
  
export function ChatProvider({ children }: { children: ReactNode }) {  
 const chat = useChat();  
  
 return (  
 <ChatContext.Provider value={chat}>  
 {children}  
 </ChatContext.Provider>  
 );  
}  
  
export function useChatContext() {  
 const context = useContext(ChatContext);  
 if (!context) {  
 throw new Error('useChatContext must be used within a ChatProvider');  
 }  
 return context;  
}

#### **Higher-Order Hook Pattern**

// app/hooks/withErrorHandling.ts  
"use client";  
  
import { useState, useCallback } from "react";  
  
interface WithErrorHandlingReturn<T> {  
 data: T | null;  
 loading: boolean;  
 error: string | null;  
 execute: (...args: any[]) => Promise<void>;  
 reset: () => void;  
}  
  
export function withErrorHandling<T>(  
 asyncFunction: (...args: any[]) => Promise<T>  
): () => WithErrorHandlingReturn<T> {  
 return function useAsyncWithErrorHandling() {  
 const [data, setData] = useState<T | null>(null);  
 const [loading, setLoading] = useState(false);  
 const [error, setError] = useState<string | null>(null);  
  
 const execute = useCallback(async (...args: any[]) => {  
 setLoading(true);  
 setError(null);  
  
 try {  
 const result = await asyncFunction(...args);  
 setData(result);  
 } catch (err: any) {  
 setError(err.message || 'An error occurred');  
 console.error('Async operation failed:', err);  
 } finally {  
 setLoading(false);  
 }  
 }, []);  
  
 const reset = useCallback(() => {  
 setData(null);  
 setError(null);  
 setLoading(false);  
 }, []);  
  
 return { data, loading, error, execute, reset };  
 };  
}  
  
// Usage  
const useApiCall = withErrorHandling(  
 async (endpoint: string) => {  
 const response = await fetch(endpoint);  
 if (!response.ok) throw new Error('API call failed');  
 return response.json();  
 }  
);

## **Hook Patterns and Best Practices**

### **1. Hook Composition Pattern**

Combine multiple hooks to create more powerful functionality:

function useChatInterface() {  
 const { accessToken, isLoading: tokenLoading } = useAccessToken();  
 const { messages, addMessage, clearMessages } = useChatMessages();  
 const [input, setInput] = useState('');  
 const [isOnline] = useOnlineStatus();  
   
 const { execute: sendMessage, isLoading: sendingMessage } = useApiRequest(  
 async (message: string) => {  
 if (!accessToken) throw new Error('No access token');  
   
 const response = await fetch('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': `Bearer ${accessToken}`,  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: message })  
 });  
   
 return response.json();  
 },  
 {  
 onSuccess: (data) => {  
 addMessage(data.content, 'assistant');  
 }  
 }  
 );  
  
 const handleSubmit = useCallback(async () => {  
 if (!input.trim() || !isOnline) return;  
   
 addMessage(input, 'user');  
 setInput('');  
 await sendMessage(input);  
 }, [input, isOnline, addMessage, sendMessage]);  
  
 return {  
 messages,  
 input,  
 setInput,  
 handleSubmit,  
 isLoading: tokenLoading || sendingMessage,  
 isOnline,  
 clearMessages  
 };  
}

### **2. Error Boundary Hook**

Create a hook for error handling:

interface UseErrorBoundaryReturn {  
 hasError: boolean;  
 error: Error | null;  
 resetError: () => void;  
 captureError: (error: Error) => void;  
}  
  
export function useErrorBoundary(): UseErrorBoundaryReturn {  
 const [error, setError] = useState<Error | null>(null);  
  
 const resetError = useCallback(() => {  
 setError(null);  
 }, []);  
  
 const captureError = useCallback((error: Error) => {  
 setError(error);  
 // Log to external service  
 if (process.env.NODE\_ENV === 'production') {  
 console.error('Captured error:', error);  
 // Send to error tracking service  
 }  
 }, []);  
  
 return {  
 hasError: !!error,  
 error,  
 resetError,  
 captureError  
 };  
}

### **3. Custom Hook Guidelines**

**Hook Naming:**

* Always start with "use"
* Use descriptive names that indicate the hook's purpose
* Use camelCase convention

**Return Values:**

* For single values, return the value directly
* For multiple related values, return an object
* For pairs of related values (like state getters/setters), return an array

// Good - Single value  
export function useAccessToken(): string | null  
  
// Good - Multiple related values  
export function useApiRequest(): { data: any, isLoading: boolean, error: string | null }  
  
// Good - Pair of related values  
export function useState<T>(initial: T): [T, (value: T) => void]

**Error Handling:**

* Always handle errors gracefully
* Provide meaningful error messages
* Use TypeScript for better error handling

export function useApiCall<T>(url: string) {  
 const [data, setData] = useState<T | null>(null);  
 const [error, setError] = useState<string | null>(null);  
  
 const execute = useCallback(async () => {  
 try {  
 setError(null);  
 const response = await fetch(url);  
   
 if (!response.ok) {  
 throw new Error(`HTTP ${response.status}: ${response.statusText}`);  
 }  
   
 const result = await response.json();  
 setData(result);  
 } catch (err) {  
 const message = err instanceof Error ? err.message : 'Unknown error';  
 setError(message);  
 setData(null);  
 }  
 }, [url]);  
  
 return { data, error, execute };  
}

## **Hook Testing**

### **Testing Custom Hooks**

Use @testing-library/react-hooks to test custom hooks:

// \_\_tests\_\_/hooks/useAccessToken.test.ts  
import { renderHook, waitFor } from '@testing-library/react';  
import { useAccessToken } from '@/app/hooks/useAccessToken';  
  
// Mock MSAL  
jest.mock('@azure/msal-react', () => ({  
 useMsal: jest.fn()  
}));  
  
describe('useAccessToken', () => {  
 const mockInstance = {  
 acquireTokenSilent: jest.fn(),  
 acquireTokenPopup: jest.fn()  
 };  
  
 const mockAccounts = [{ username: '[test@example.com](mailto:test@example.com)' }];  
  
 beforeEach(() => {  
 ([require('@azure/msal-react').useMsal](mailto:require('@azure/msal-react').useMsal) as jest.Mock).mockReturnValue({  
 instance: mockInstance,  
 accounts: mockAccounts  
 });  
 });  
  
 afterEach(() => {  
 jest.clearAllMocks();  
 });  
  
 it('should return null initially and then acquire token', async () => {  
 mockInstance.acquireTokenSilent.mockResolvedValue({  
 accessToken: 'mock-token'  
 });  
  
 const { result } = renderHook(() => useAccessToken());  
  
 expect(result.current.accessToken).toBeNull();  
 expect(result.current.isLoading).toBe(true);  
  
 await waitFor(() => {  
 expect(result.current.accessToken).toBe('mock-token');  
 expect(result.current.isLoading).toBe(false);  
 });  
 });  
  
 it('should handle silent token acquisition failure and fallback to popup', async () => {  
 mockInstance.acquireTokenSilent.mockRejectedValue(new Error('Silent failed'));  
 mockInstance.acquireTokenPopup.mockResolvedValue({  
 accessToken: 'popup-token'  
 });  
  
 const { result } = renderHook(() => useAccessToken());  
  
 await waitFor(() => {  
 expect(result.current.accessToken).toBe('popup-token');  
 expect(result.current.error).toBeNull();  
 });  
  
 expect(mockInstance.acquireTokenSilent).toHaveBeenCalled();  
 expect(mockInstance.acquireTokenPopup).toHaveBeenCalled();  
 });  
  
 it('should handle complete token acquisition failure', async () => {  
 mockInstance.acquireTokenSilent.mockRejectedValue(new Error('Silent failed'));  
 mockInstance.acquireTokenPopup.mockRejectedValue(new Error('Popup failed'));  
  
 const { result } = renderHook(() => useAccessToken());  
  
 await waitFor(() => {  
 expect(result.current.accessToken).toBeNull();  
 expect(result.current.error).toBe('Failed to acquire access token');  
 });  
 });  
});

### **Testing the useChat Hook**

// \_\_tests\_\_/hooks/useChat.test.ts  
import { renderHook, act, waitFor } from '@testing-library/react';  
import { useChat } from '@/app/hooks/useChat';  
  
// Mock the useAccessToken hook  
jest.mock('@/app/hooks/useAccessToken', () => ({  
 useAccessToken: () => 'mock-access-token'  
}));  
  
// Mock fetch  
global.fetch = jest.fn();  
  
describe('useChat', () => {  
 beforeEach(() => {  
 jest.clearAllMocks();  
 });  
  
 it('should initialize with empty state', () => {  
 const { result } = renderHook(() => useChat());  
  
 expect(result.current.messages).toEqual([]);  
 expect(result.current.input).toBe('');  
 expect(result.current.isLoading).toBe(false);  
 expect(result.current.error).toBe(null);  
 });  
  
 it('should update input value', () => {  
 const { result } = renderHook(() => useChat());  
  
 act(() => {  
 result.current.setInput('Hello, world!');  
 });  
  
 expect(result.current.input).toBe('Hello, world!');  
 });  
  
 it('should send message successfully', async () => {  
 const mockResponse = {  
 ok: true,  
 json: () => Promise.resolve({  
 data: { content: 'Hello from AI!' }  
 })  
 };  
  
 (global.fetch as jest.Mock).mockResolvedValueOnce(mockResponse);  
  
 const { result } = renderHook(() => useChat());  
  
 act(() => {  
 result.current.setInput('Hello');  
 });  
  
 await act(async () => {  
 await result.current.sendMessage();  
 });  
  
 expect(result.current.messages).toHaveLength(2); // User + Assistant  
 expect(result.current.messages[0].content).toBe('Hello');  
 expect(result.current.messages[0].role).toBe('user');  
 expect(result.current.messages[1].content).toBe('Hello from AI!');  
 expect(result.current.messages[1].role).toBe('assistant');  
 expect(result.current.input).toBe(''); // Input should be cleared  
 });  
  
 it('should handle API errors', async () => {  
 const mockResponse = {  
 ok: false,  
 json: () => Promise.resolve({  
 error: 'API Error'  
 })  
 };  
  
 (global.fetch as jest.Mock).mockResolvedValueOnce(mockResponse);  
  
 const { result } = renderHook(() => useChat());  
  
 act(() => {  
 result.current.setInput('Hello');  
 });  
  
 await act(async () => {  
 await result.current.sendMessage();  
 });  
  
 expect(result.current.error).toBe('Error: API Error');  
 expect(result.current.messages).toHaveLength(2); // User message + Error message  
 expect(result.current.messages[1].content).toContain('Error: API Error');  
 });  
  
 it('should clear messages', () => {  
 const { result } = renderHook(() => useChat());  
  
 // Add some messages first  
 act(() => {  
 result.current.setInput('Test');  
 });  
  
 act(() => {  
 result.current.clearMessages();  
 });  
  
 expect(result.current.messages).toEqual([]);  
 expect(result.current.error).toBe(null);  
 });  
});

#### **Testing useLocalStorage Hook**

// \_\_tests\_\_/hooks/useLocalStorage.test.ts  
import { renderHook, act } from '@testing-library/react';  
import { useLocalStorage } from '@/app/hooks/useLocalStorage';  
  
// Mock localStorage  
const localStorageMock = {  
 getItem: jest.fn(),  
 setItem: jest.fn(),  
 removeItem: jest.fn(),  
 clear: jest.fn(),  
};  
  
Object.defineProperty(window, 'localStorage', {  
 value: localStorageMock  
});  
  
describe('useLocalStorage', () => {  
 beforeEach(() => {  
 jest.clearAllMocks();  
 });  
  
 it('should initialize with initial value when localStorage is empty', () => {  
 localStorageMock.getItem.mockReturnValue(null);  
  
 const { result } = renderHook(() =>   
 useLocalStorage('test-key', 'initial-value')  
 );  
  
 expect(result.current[0]).toBe('initial-value');  
 expect(localStorageMock.getItem).toHaveBeenCalledWith('test-key');  
 });  
  
 it('should initialize with value from localStorage', () => {  
 localStorageMock.getItem.mockReturnValue('"stored-value"');  
  
 const { result } = renderHook(() =>   
 useLocalStorage('test-key', 'initial-value')  
 );  
  
 expect(result.current[0]).toBe('stored-value');  
 });  
  
 it('should update localStorage when value changes', () => {  
 localStorageMock.getItem.mockReturnValue(null);  
  
 const { result } = renderHook(() =>   
 useLocalStorage('test-key', 'initial-value')  
 );  
  
 act(() => {  
 result.current[1]('new-value');  
 });  
  
 expect(result.current[0]).toBe('new-value');  
 expect(localStorageMock.setItem).toHaveBeenCalledWith(  
 'test-key',   
 '"new-value"'  
 );  
 });  
  
 it('should handle localStorage errors gracefully', () => {  
 localStorageMock.getItem.mockImplementation(() => {  
 throw new Error('localStorage error');  
 });  
  
 const consoleSpy = jest.spyOn(console, 'warn').mockImplementation();  
  
 const { result } = renderHook(() =>   
 useLocalStorage('test-key', 'fallback-value')  
 );  
  
 expect(result.current[0]).toBe('fallback-value');  
 expect(consoleSpy).toHaveBeenCalled();  
  
 consoleSpy.mockRestore();  
 });  
});

#### **Testing Hook Integration**

// \_\_tests\_\_/components/ChatInterface.integration.test.tsx  
import { render, screen, fireEvent, waitFor } from '@testing-library/react';  
import { MsalProvider } from '@azure/msal-react';  
import ChatInterface from '@/app/components/ChatInterface';  
  
// Mock MSAL  
const mockMsalInstance = {  
 acquireTokenSilent: jest.fn(),  
 acquireTokenPopup: jest.fn(),  
};  
  
const mockMsalConfig = {  
 auth: { clientId: 'test-client-id' }  
};  
  
describe('ChatInterface Integration', () => {  
 beforeEach(() => {  
 global.fetch = jest.fn();  
 jest.clearAllMocks();  
 });  
  
 const renderWithMsal = (component: React.ReactElement) => {  
 return render(  
 <MsalProvider instance={mockMsalInstance} configuration={mockMsalConfig}>  
 {component}  
 </MsalProvider>  
 );  
 };  
  
 it('should handle complete chat flow', async () => {  
 // Mock successful token acquisition  
 mockMsalInstance.acquireTokenSilent.mockResolvedValue({  
 accessToken: 'mock-token'  
 });  
  
 // Mock successful API response  
 (global.fetch as jest.Mock).mockResolvedValue({  
 ok: true,  
 json: () => Promise.resolve({  
 data: { content: 'AI response' }  
 })  
 });  
  
 renderWithMsal(<ChatInterface />);  
  
 const input = screen.getByPlaceholderText('Type your message...');  
 const sendButton = screen.getByText('Send');  
  
 // Type a message  
 fireEvent.change(input, { target: { value: 'Hello AI' } });  
   
 // Send the message  
 fireEvent.click(sendButton);  
  
 // Wait for user message to appear  
 await waitFor(() => {  
 expect(screen.getByText('Hello AI')).toBeInTheDocument();  
 });  
  
 // Wait for AI response  
 await waitFor(() => {  
 expect(screen.getByText('AI response')).toBeInTheDocument();  
 });  
  
 // Verify API was called correctly  
 expect(global.fetch).toHaveBeenCalledWith('/api/completions', {  
 method: 'POST',  
 headers: {  
 'Authorization': 'Bearer mock-token',  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ prompt: 'Hello AI' })  
 });  
 });  
});

### **Jest Configuration for Hook Testing**

// jest.config.js  
module.exports = {  
 testEnvironment: 'jsdom',  
 setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],  
 moduleNameMapping: {  
 '^@/(.\*)$': '<rootDir>/$1',  
 },  
 testPathIgnorePatterns: ['<rootDir>/.next/', '<rootDir>/node\_modules/'],  
 transform: {  
 '^.+\\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],  
 },  
 collectCoverageFrom: [  
 'app/\*\*/\*.{js,jsx,ts,tsx}',  
 '!app/\*\*/\*.d.ts',  
 '!app/\*\*/index.{js,ts}',  
 ],  
};  
 // jest.setup.js  
import '@testing-library/jest-dom';  
  
// Mock crypto.randomUUID for Node.js environment  
Object.defineProperty(global.crypto, 'randomUUID', {  
 value: () => Math.random().toString(36).substring(2, 15),  
});  
  
// Mock window.localStorage  
const localStorageMock = {  
 getItem: jest.fn(),  
 setItem: jest.fn(),  
 removeItem: jest.fn(),  
 clear: jest.fn(),  
};  
  
Object.defineProperty(window, 'localStorage', {  
 value: localStorageMock  
});  
  
// Reset all mocks before each test  
beforeEach(() => {  
 jest.clearAllMocks();  
});

## **Git Hooks**

### **Pre-commit Hook**

Automatically run code quality checks before commits:

#!/bin/sh  
# .husky/pre-commit  
  
echo "Running pre-commit checks..."  
  
# Run type checking  
echo "Type checking..."  
npm run type-check  
if [ $? -ne 0 ]; then  
 echo "TypeScript errors found. Please fix them before committing."  
 exit 1  
fi  
  
# Run linting  
echo "Linting..."  
npm run lint:strict  
if [ $? -ne 0 ]; then  
 echo "Linting errors found. Please fix them before committing."  
 exit 1  
fi  
  
# Run formatting check  
echo " Checking code formatting..."  
npm run format:check  
if [ $? -ne 0 ]; then  
 echo "Code formatting issues found. Run 'npm run format' to fix them."  
 exit 1  
fi  
  
# Run tests  
echo " Running tests..."  
npm run test  
if [ $? -ne 0 ]; then  
 echo " Tests failed. Please fix them before committing."  
 exit 1  
fi  
  
echo "\All pre-commit checks passed!"

### **Commit Message Hook**

Enforce commit message conventions:

#!/bin/sh  
# .husky/commit-msg  
  
commit\_regex='^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?: .{1,50}'  
  
if ! grep -qE "$commit\_regex" "$1"; then  
 echo "Invalid commit message format!"  
 echo ""  
 echo "Commit message should follow the pattern:"  
 echo "type(scope): description"  
 echo ""  
 echo "Types: feat, fix, docs, style, refactor, test, chore"  
 echo "Example: feat(chat): add message history functionality"  
 echo ""  
 exit 1  
fi  
  
echo "Commit message format is valid!"

### **Pre-push Hook**

Run additional checks before pushing:

#!/bin/sh  
# .husky/pre-push  
  
echo "Running pre-push checks..."  
  
# Check if we're on a protected branch  
protected\_branch='main'  
current\_branch=$(git rev-parse --abbrev-ref HEAD);  
  
if [ $protected\_branch = $current\_branch ]; then  
 echo "Direct push to $protected\_branch branch is not allowed!"  
 echo "Please create a pull request instead."  
 exit 1  
fi  
  
# Run full test suite  
echo "Running full test suite..."  
npm run test:coverage  
if [ $? -ne 0 ]; then  
 echo "Tests failed. Please fix them before pushing."  
 exit 1  
fi  
  
# Check test coverage  
echo "Checking test coverage..."  
npm run test:coverage -- --passWithNoTests --watchAll=false --coverageThreshold='{"global":{"branches":80,"functions":80,"lines":80,"statements":80}}'  
if [ $? -ne 0 ]; then  
 echo "Test coverage below threshold. Please add more tests."  
 exit 1  
fi  
  
# Build check  
echo " Building application..."  
npm run build  
if [ $? -ne 0 ]; then  
 echo "Build failed. Please fix build errors before pushing."  
 exit 1  
fi  
  
echo "All pre-push checks passed!"

### **Installing Git Hooks**

Configure Husky for automatic git hook management:

// package.json  
{  
 "scripts": {  
 "prepare": "husky install",  
 "postinstall": "husky install"  
 },  
 "husky": {  
 "hooks": {  
 "pre-commit": "lint-staged",  
 "commit-msg": "commitlint -E HUSKY\_GIT\_PARAMS",  
 "pre-push": "npm run test && npm run build"  
 }  
 },  
 "lint-staged": {  
 "\*.{ts,tsx,js,jsx}": [  
 "eslint --fix",  
 "prettier --write"  
 ],  
 "\*.{css,scss,md,json}": [  
 "prettier --write"  
 ]  
 }  
}

Installation commands:

# Install Husky  
npm install --save-dev husky  
  
# Initialize Husky  
npx husky install  
  
# Add pre-commit hook  
npx husky add .husky/pre-commit "npx lint-staged"  
  
# Add commit message hook  
npx husky add .husky/commit-msg "npx commitlint --edit $1"  
  
# Add pre-push hook  
npx husky add .husky/pre-push "npm run test && npm run build"

## **Performance Optimization**

### **Hook Performance Best Practices**

#### **1. Memoization Strategy**

// Bad - Creates new object on every render  
function BadComponent({ userId }: { userId: string }) {  
 const userConfig = {  
 id: userId,  
 settings: { theme: 'dark' }  
 };  
  
 return <UserProfile config={userConfig} />;  
}  
  
// Good - Memoized object  
function GoodComponent({ userId }: { userId: string }) {  
 const userConfig = useMemo(() => ({  
 id: userId,  
 settings: { theme: 'dark' }  
 }), [userId]);  
  
 return <UserProfile config={userConfig} />;  
}

#### **2. Callback Optimization**

// Bad - Creates new function on every render  
function BadComponent() {  
 const [count, setCount] = useState(0);  
  
 return (  
 <ExpensiveChild   
 onClick={() => setCount(count + 1)}   
 />  
 );  
}  
  
// Good - Memoized callback  
function GoodComponent() {  
 const [count, setCount] = useState(0);  
  
 const handleClick = useCallback(() => {  
 setCount(prev => prev + 1);  
 }, []);  
  
 return (  
 <ExpensiveChild onClick={handleClick} />  
 );  
}

#### **3. Effect Optimization**

// Bad - Effect runs on every render  
function BadComponent({ user, settings }) {  
 useEffect(() => {  
 updateUserSettings(user.id, settings);  
 }); // No dependency array  
  
 return <div>...</div>;  
}  
  
// Good - Effect runs only when dependencies change  
function GoodComponent({ user, settings }) {  
 useEffect(() => {  
 updateUserSettings(user.id, settings);  
 }, [user.id, settings]); // Proper dependencies  
  
 return <div>...</div>;  
}

### **Hook Profiling**

Use React DevTools Profiler to identify performance bottlenecks:

// Add performance marks for debugging  
function useAccessTokenWithProfiling() {  
 const { instance, accounts } = useMsal();  
 const [accessToken, setAccessToken] = useState<string | null>(null);  
  
 useEffect(() => {  
 performance.mark('token-acquisition-start');  
   
 const getToken = async () => {  
 try {  
 const response = await instance.acquireTokenSilent({  
 ...loginRequest,  
 account: accounts[0]  
 });  
   
 performance.mark('token-acquisition-end');  
 performance.measure(  
 'token-acquisition',  
 'token-acquisition-start',  
 'token-acquisition-end'  
 );  
   
 setAccessToken(response.accessToken);  
 } catch (error) {  
 performance.mark('token-acquisition-error');  
 console.error("Failed to get access token:", error);  
 }  
 };  
  
 if (accounts.length > 0) {  
 getToken();  
 }  
 }, [instance, accounts]);  
  
 return accessToken;  
}

## **Troubleshooting**

### **Common Hook Issues**

#### **1. Stale Closure Problem**

**Problem:** Hook captures outdated values in closures.

// Problem - Stale closure  
function BadCounter() {  
 const [count, setCount] = useState(0);  
  
 useEffect(() => {  
 const interval = setInterval(() => {  
 setCount(count + 1); // Always uses initial count (0)  
 }, 1000);  
  
 return () => clearInterval(interval);  
 }, []); // Empty dependency array  
  
 return <div>{count}</div>;  
}  
  
// Solution 1 - Functional update  
function GoodCounter1() {  
 const [count, setCount] = useState(0);  
  
 useEffect(() => {  
 const interval = setInterval(() => {  
 setCount(prev => prev + 1); // Uses current count  
 }, 1000);  
  
 return () => clearInterval(interval);  
 }, []);  
  
 return <div>{count}</div>;  
}  
  
// Solution 2 - Include dependency  
function GoodCounter2() {  
 const [count, setCount] = useState(0);  
  
 useEffect(() => {  
 const interval = setInterval(() => {  
 setCount(count + 1);  
 }, 1000);  
  
 return () => clearInterval(interval);  
 }, [count]); // Include count in dependencies  
  
 return <div>{count}</div>;  
}

#### **2. Infinite Re-render Loops**

**Problem:** Effect triggers re-render which triggers effect again.

// Problem - Infinite loop  
function BadComponent({ userId }: { userId: string }) {  
 const [user, setUser] = useState(null);  
  
 useEffect(() => {  
 fetchUser(userId).then(setUser);  
 }, [user]); // user changes trigger new fetch  
  
 return <div>{user?.name}</div>;  
}  
  
// Solution - Correct dependencies  
function GoodComponent({ userId }: { userId: string }) {  
 const [user, setUser] = useState(null);  
  
 useEffect(() => {  
 fetchUser(userId).then(setUser);  
 }, [userId]); // Only userId changes trigger fetch  
  
 return <div>{user?.name}</div>;  
}

#### **3. Memory Leaks**

**Problem:** Effect cleanup not properly implemented.

// Problem - Memory leak  
function BadComponent() {  
 const [data, setData] = useState(null);  
  
 useEffect(() => {  
 const interval = setInterval(() => {  
 fetchData().then(setData);  
 }, 5000);  
 // No cleanup - interval continues after unmount  
 }, []);  
  
 return <div>{data}</div>;  
}  
  
// Solution - Proper cleanup  
function GoodComponent() {  
 const [data, setData] = useState(null);  
  
 useEffect(() => {  
 let isMounted = true;  
   
 const interval = setInterval(() => {  
 fetchData().then(result => {  
 if (isMounted) {  
 setData(result);  
 }  
 });  
 }, 5000);  
  
 return () => {  
 isMounted = false;  
 clearInterval(interval);  
 };  
 }, []);  
  
 return <div>{data}</div>;  
}

### **Debugging Hooks**

#### **1. Custom Hook Debugging**

function useDebugValue<T>(value: T, formatter?: (value: T) => any) {  
 useDebugValue(value, formatter);  
 return value;  
}  
  
// Usage in custom hook  
function useAccessToken() {  
 const [token, setToken] = useState<string | null>(null);  
   
 // This will show in React DevTools  
 useDebugValue(token, token => token ? 'Token acquired' : 'No token');  
   
 return token;  
}

#### **2. Effect Debugging**

function useEffectDebugger(effectHook: any, dependencies: any[], dependencyNames: string[] = []) {  
 const previousDeps = usePrevious(dependencies);  
 const changedDeps = dependencies.reduce((accum, dependency, index) => {  
 if (previousDeps && previousDeps[index] !== dependency) {  
 const keyName = dependencyNames[index] || index;  
 return {  
 ...accum,  
 [keyName]: {  
 before: previousDeps[index],  
 after: dependency  
 }  
 };  
 }  
  
 return accum;  
 }, {});  
  
 if (Object.keys(changedDeps).length) {  
 console.log('[useEffect] Changed dependencies:', changedDeps);  
 }  
  
 useEffect(effectHook, dependencies);  
}  
  
// Usage  
function MyComponent({ userId, settings }) {  
 useEffectDebugger(() => {  
 // Effect logic  
 }, [userId, settings], ['userId', 'settings']);  
}

## **Migration Guide**

### **Migrating from Class Components**

#### **Before (Class Component)**

class ChatComponent extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 messages: [],  
 input: '',  
 isLoading: false  
 };  
 }  
  
 async componentDidMount() {  
 await this.loadMessages();  
 }  
  
 componentDidUpdate(prevProps) {  
 if (prevProps.userId !== this.props.userId) {  
 this.loadMessages();  
 }  
 }  
  
 componentWillUnmount() {  
 this.cleanup();  
 }  
  
 loadMessages = async () => {  
 this.setState({ isLoading: true });  
 try {  
 const messages = await fetchMessages(this.props.userId);  
 this.setState({ messages });  
 } catch (error) {  
 console.error('Failed to load messages:', error);  
 } finally {  
 this.setState({ isLoading: false });  
 }  
 };  
  
 render() {  
 return (  
 <div>  
 {this.state.messages.map(message => (  
 <div key={message.id}>{message.content}</div>  
 ))}  
 </div>  
 );  
 }  
}

#### **After (Function Component with Hooks)**

function ChatComponent({ userId }: { userId: string }) {  
 const [messages, setMessages] = useState<ChatMessage[]>([]);  
 const [input, setInput] = useState('');  
 const [isLoading, setIsLoading] = useState(false);  
  
 const loadMessages = useCallback(async () => {  
 setIsLoading(true);  
 try {  
 const fetchedMessages = await fetchMessages(userId);  
 setMessages(fetchedMessages);  
 } catch (error) {  
 console.error('Failed to load messages:', error);  
 } finally {  
 setIsLoading(false);  
 }  
 }, [userId]);  
  
 useEffect(() => {  
 loadMessages();  
 }, [loadMessages]);  
  
 useEffect(() => {  
 // Cleanup logic  
 return () => {  
 // Cleanup code here  
 };  
 }, []);  
  
 return (  
 <div>  
 {messages.map(message => (  
 <div key={message.id}>{message.content}</div>  
 ))}  
 </div>  
 );  
}