# **Coding Standards & Lint Rules Documentation**

## **Overview**

This document outlines the coding standards, best practices, and linting rules for the Chat Application project. Following these guidelines ensures code consistency, maintainability, and quality across the development team.

## **General Coding Standards**

### **Code Quality Principles**

1. **Consistency**: Follow established patterns throughout the codebase
2. **Readability**: Write self-documenting code with clear intentions
3. **Maintainability**: Structure code for easy modification and extension
4. **Performance**: Consider performance implications of code decisions
5. **Security**: Follow security best practices for authentication and data handling

### **General Rules**

* Use TypeScript for all new code
* Prefer explicit typing over any
* Write meaningful comments for complex logic
* Keep functions small and focused (max 50 lines recommended)
* Use descriptive variable and function names
* Remove unused imports and variables
* Use strict mode TypeScript configuration

## **TypeScript Standards**

### **TypeScript Configuration**

Current tsconfig.json settings should be maintained:

{  
 "compilerOptions": {  
 "target": "ES2017",  
 "strict": true,  
 "noEmit": true,  
 "esModuleInterop": true,  
 "module": "esnext",  
 "moduleResolution": "bundler",  
 "resolveJsonModule": true,  
 "isolatedModules": true,  
 "jsx": "preserve"  
 }  
}

### **Type Definitions**

#### **Interface Naming**

* Use PascalCase for interface names
* Prefix interfaces with I only when necessary for disambiguation
* Use descriptive names that indicate the interface purpose

interface ChatMessage {  
 role: 'user' | 'assistant';  
 content: string;  
 timestamp?: Date;  
}  
  
interface ApiResponse<T> {  
 data: T;  
 error?: string;  
}  
  
// Avoid  
interface IMessage {} // Unnecessary I prefix  
interface Data {} // Too generic

#### **Type Definitions**

* Use union types for known string literals
* Prefer interface over type for object shapes
* Use type for unions, primitives, and computed types

type MessageRole = 'user' | 'assistant';  
type ApiStatus = 'loading' | 'success' | 'error';  
  
interface UserAccount {  
 username: string;  
 email: string;  
 roles: string[];  
}  
  
// Avoid  
type UserAccount = {  
 username: string;  
 email: string;  
} // Use interface instead

### **Function Typing**

// Good - Explicit return types for exported functions  
export async function fetchCompletion(  
 prompt: string,  
 token: string  
): Promise<ApiResponse<CompletionData>> {  
 // Implementation  
}  
  
// Good - Arrow functions with explicit types  
const handleSubmit = async (e: React.FormEvent): Promise<void> => {  
 // Implementation  
};  
  
// Good - Generic functions  
function createApiResponse<T>(data: T, error?: string): ApiResponse<T> {  
 return { data, error };  
}

## **React/Next.js Standards**

### **Component Structure**

#### **Functional Components**

* Use functional components with hooks
* Use TypeScript for all component props
* Export components as default exports

// Good - Component structure  
"use client"; // Only when needed  
  
import { useState, useEffect } from "react";  
import { useMsal } from "@azure/msal-react";  
  
interface ChatInterfaceProps {  
 initialMessage?: string;  
 onMessageSent?: (message: string) => void;  
}  
  
export default function ChatInterface({   
 initialMessage,   
 onMessageSent   
}: ChatInterfaceProps) {  
 const [messages, setMessages] = useState<ChatMessage[]>([]);  
 const [isLoading, setIsLoading] = useState(false);  
  
 // Component logic here  
  
 return (  
 <div className="chat-interface">  
 {/\* JSX content \*/}  
 </div>  
 );  
}

#### **Props Interface**

* Define props interface above the component
* Use optional properties with ? when appropriate
* Document complex props with JSDoc comments

interface ComponentProps {  
 /\*\* The primary message to display \*/  
 message: string;  
 /\*\* Optional callback when user interacts \*/  
 onInteraction?: (data: InteractionData) => void;  
 /\*\* Loading state indicator \*/  
 isLoading?: boolean;  
}

### **Hooks Standards**

#### **Custom Hooks**

* Prefix custom hook names with use
* Return objects for multiple values, arrays for related pairs
* Include TypeScript return types

// Good - Custom hook structure  
interface UseAccessTokenReturn {  
 accessToken: string | null;  
 isLoading: boolean;  
 error: string | null;  
}  
  
export function useAccessToken(): UseAccessTokenReturn {  
 const [accessToken, setAccessToken] = useState<string | null>(null);  
 const [isLoading, setIsLoading] = useState(false);  
 const [error, setError] = useState<string | null>(null);  
  
 // Hook logic  
  
 return { accessToken, isLoading, error };  
}

#### **State Management**

* Use descriptive state variable names
* Initialize state with proper types
* Group related state when appropriate

// Good - State management  
const [chatMessages, setChatMessages] = useState<ChatMessage[]>([]);  
const [inputValue, setInputValue] = useState("");  
const [apiStatus, setApiStatus] = useState<ApiStatus>('idle');  
  
// Good - Related state grouping  
interface ChatState {  
 messages: ChatMessage[];  
 isLoading: boolean;  
 error: string | null;  
}  
  
const [chatState, setChatState] = useState<ChatState>({  
 messages: [],  
 isLoading: false,  
 error: null  
});

### **Event Handling**

//Good - Event handler typing  
const handleFormSubmit = (e: React.FormEvent<HTMLFormElement>): void => {  
 e.preventDefault();  
 // Handle submission  
};  
  
const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>): void => {  
 setInputValue(e.target.value);  
};  
  
const handleButtonClick = async (): Promise<void> => {  
 setIsLoading(true);  
 try {  
 await performAction();  
 } catch (error) {  
 console.error('Action failed:', error);  
 } finally {  
 setIsLoading(false);  
 }  
};

## **CSS/Tailwind Standards**

### **Tailwind CSS Guidelines**

#### **Class Organization**

* Order classes logically: layout → spacing → sizing → styling → interactions
* Use responsive prefixes consistently
* Prefer Tailwind utilities over custom CSS

// Good - Organized class structure  
<button  
 className="  
 flex items-center justify-center  
 px-4 py-2   
 w-full sm:w-auto  
 rounded-lg border border-gray-300  
 bg-blue-500 text-white  
 hover:bg-blue-600 focus:ring-2 focus:ring-blue-500  
 disabled:opacity-50 disabled:cursor-not-allowed  
 transition-colors duration-200  
 "  
>  
 Submit  
</button>  
  
// Avoid - Random class order  
<button className="text-white bg-blue-500 px-4 border flex py-2 hover:bg-blue-600">  
 Submit  
</button>

#### **Responsive Design**

* Mobile-first approach (start with base styles, add sm:, md:, etc.)
* Use consistent breakpoints
* Test on multiple screen sizes

// Good - Mobile-first responsive design  
<div className="  
 flex flex-col space-y-4  
 sm:flex-row sm:space-y-0 sm:space-x-4  
 md:max-w-2xl md:mx-auto  
 lg:max-w-4xl  
">  
 {/\* Content \*/}  
</div>

#### **Custom CSS**

* Use CSS modules or Tailwind @apply for complex components
* Avoid inline styles unless dynamic
* Use CSS custom properties for theme values

/\* Good - globals.css additions \*/  
:root {  
 --chat-background: #f8fafc;  
 --chat-border: #e2e8f0;  
 --message-spacing: 1rem;  
}  
  
.chat-container {  
 @apply bg-white/5 rounded-lg p-4 space-y-4;  
 background-color: var(--chat-background);  
}

## **File Organization**

### **Directory Structure**

app/  
├── components/ # Reusable UI components  
│ ├── ui/ # Basic UI components (buttons, inputs)  
│ ├── forms/ # Form-specific components  
│ └── layout/ # Layout components  
├── hooks/ # Custom React hooks  
├── utils/ # Utility functions  
├── types/ # TypeScript type definitions  
├── lib/ # External library configurations  
├── api/ # API route handlers  
│ └── [endpoint]/  
│ └── route.ts  
└── (routes)/ # Page components and layouts

### **File Naming**

* **Components**: PascalCase (e.g., ChatInterface.tsx, AuthButton.tsx)
* **Hooks**: camelCase starting with "use" (e.g., useAccessToken.ts)
* **Utilities**: camelCase (e.g., apiHelpers.ts, dateUtils.ts)
* **Types**: PascalCase (e.g., ApiTypes.ts, ChatTypes.ts)
* **API Routes**: lowercase (e.g., route.ts in appropriate folders)

### **Import Organization**

// Good - Import order  
// 1. React and Next.js imports  
import { useState, useEffect } from "react";  
import { NextRequest, NextResponse } from "next/server";  
  
// 2. Third-party libraries  
import { useMsal } from "@azure/msal-react";  
  
// 3. Internal imports (absolute paths)  
import { useAccessToken } from "@/app/hooks/useAccessToken";  
import { ChatMessage } from "@/app/types/ChatTypes";  
import { apiHelpers } from "@/app/utils/apiHelpers";  
  
// 4. Relative imports  
import "./Component.css";

## **Naming Conventions**

### **Variables and Functions**

* Use camelCase for variables and functions
* Use descriptive names that indicate purpose
* Avoid abbreviations unless widely understood

// Good  
const accessToken = getAccessToken();  
const isUserAuthenticated = checkAuthStatus();  
const handleFormSubmission = async () => {};  
  
// Avoid  
const tkn = getToken();  
const chkAuth = () => {};  
const handleSubmit = () => {}; // Too generic

### **Constants**

* Use SCREAMING\_SNAKE\_CASE for module-level constants
* Group related constants in objects

// Good  
const API\_ENDPOINTS = {  
 COMPLETIONS: '/api/completions',  
 USERS: '/api/users'  
} as const;  
  
const MAX\_RETRY\_ATTEMPTS = 3;  
const DEFAULT\_TIMEOUT = 5000;  
  
// Avoid  
const apiUrl = "<https://api.example.com>"; // Should be constant case  
const maxRetries = 3; // Should be constant case for module-level constants

### **Component and Type Names**

* Use PascalCase for all component names and TypeScript types
* Use descriptive names that indicate the component's purpose

// Good  
interface ChatMessage {  
 id: string;  
 content: string;  
 role: MessageRole;  
}  
  
type ApiResponseStatus = 'loading' | 'success' | 'error';  
  
export default function ChatInterface() {}  
export function MessageBubble() {}  
  
// Avoid  
interface message {} // Should be PascalCase  
type status = string; // Should be PascalCase and more specific

## **ESLint Configuration**

### **Recommended ESLint Setup**

Create .eslintrc.json:

{  
 "extends": [  
 "next/core-web-vitals",  
 "@typescript-eslint/recommended",  
 "eslint:recommended"  
 ],  
 "parser": "@typescript-eslint/parser",  
 "plugins": ["@typescript-eslint", "react-hooks"],  
 "rules": {  
 // TypeScript Rules  
 "@typescript-eslint/no-unused-vars": "error",  
 "@typescript-eslint/no-explicit-any": "warn",  
 "@typescript-eslint/explicit-function-return-type": "warn",  
 "@typescript-eslint/prefer-const": "error",  
 "@typescript-eslint/no-non-null-assertion": "warn",  
   
 // React Rules  
 "react-hooks/rules-of-hooks": "error",  
 "react-hooks/exhaustive-deps": "warn",  
 "react/prop-types": "off", // Not needed with TypeScript  
 "react/react-in-jsx-scope": "off", // Not needed in Next.js  
   
 // General Rules  
 "no-console": "warn",  
 "no-debugger": "error",  
 "no-duplicate-imports": "error",  
 "no-unused-expressions": "error",  
 "prefer-const": "error",  
 "no-var": "error",  
   
 // Code Style  
 "indent": ["error", 2],  
 "quotes": ["error", "double"],  
 "semi": ["error", "always"],  
 "comma-dangle": ["error", "never"],  
 "object-curly-spacing": ["error", "always"],  
 "array-bracket-spacing": ["error", "never"]  
 },  
 "env": {  
 "browser": true,  
 "node": true,  
 "es6": true  
 }  
}

### **ESLint Scripts in package.json**

{  
 "scripts": {  
 "lint": "eslint . --ext .ts,.tsx,.js,.jsx",  
 "lint:fix": "eslint . --ext .ts,.tsx,.js,.jsx --fix",  
 "lint:check": "eslint . --ext .ts,.tsx,.js,.jsx --max-warnings 0"  
 }  
}

## **Prettier Configuration**

### **Prettier Setup**

Create .prettierrc:

{  
 "semi": true,  
 "trailingComma": "none",  
 "singleQuote": false,  
 "printWidth": 80,  
 "tabWidth": 2,  
 "useTabs": false,  
 "bracketSpacing": true,  
 "bracketSameLine": false,  
 "arrowParens": "avoid",  
 "endOfLine": "lf",  
 "quoteProps": "as-needed"  
}

Create .prettierignore:

node\_modules  
.next  
out  
dist  
build  
\*.min.js  
\*.min.css

### **Prettier Scripts**

{  
 "scripts": {  
 "format": "prettier --write .",  
 "format:check": "prettier --check ."  
 }  
}

## **Import/Export Standards**

### **Import Grouping and Ordering**

// Good - Proper import organization  
// 1. Node modules  
import React, { useState, useEffect } from "react";  
import { NextRequest, NextResponse } from "next/server";  
  
// 2. Third-party libraries  
import { useMsal } from "@azure/msal-react";  
  
// 3. Internal modules (absolute imports)  
import { useAccessToken } from "@/app/hooks/useAccessToken";  
import { ChatMessage, ApiResponse } from "@/app/types";  
  
// 4. Relative imports  
import "./styles.css";  
  
// 5. Type-only imports (separate from value imports)  
import type { ComponentProps } from "react";  
import type { CustomType } from "@/app/types";

### **Export Standards**

//Good - Named exports for utilities  
export const apiHelpers = {  
 formatResponse,  
 handleError  
};  
  
export { validateToken } from "./validators";  
  
// Good - Default export for main component  
export default function ChatInterface() {  
 // Component implementation  
}  
  
// Good - Type exports  
export type { ChatMessage, ApiResponse } from "./types";

### **Re-exports (Index files)**

// Good - app/components/index.ts  
export { default as ChatInterface } from "./ChatInterface";  
export { default as AuthButton } from "./AuthButton";  
export { default as MessageBubble } from "./MessageBubble";  
  
// Good - app/types/index.ts  
export type { ChatMessage } from "./ChatTypes";  
export type { ApiResponse } from "./ApiTypes";  
export type { UserAccount } from "./AuthTypes";

## **Error Handling Standards**

### **API Error Handling**

// Good - Consistent API error handling  
interface ApiError {  
 message: string;  
 code?: string;  
 status?: number;  
}  
  
async function handleApiRequest<T>(  
 request: () => Promise<Response>  
): Promise<ApiResponse<T>> {  
 try {  
 const response = await request();  
   
 if (!response.ok) {  
 const errorData = await response.json().catch(() => ({}));  
 throw new ApiError({  
 message: errorData.error || 'Request failed',  
 status: response.status  
 });  
 }  
   
 return await response.json();  
 } catch (error) {  
 console.error('API request failed:', error);  
 throw error;  
 }  
}

### **Component Error Handling**

// Good - Component error handling  
export default function ChatInterface() {  
 const [error, setError] = useState<string | null>(null);  
  
 const handleSubmit = async (message: string): Promise<void> => {  
 setError(null);  
   
 try {  
 await sendMessage(message);  
 } catch (err) {  
 const errorMessage = err instanceof Error   
 ? err.message   
 : 'An unexpected error occurred';  
 setError(errorMessage);  
 console.error('Failed to send message:', err);  
 }  
 };  
  
 return (  
 <div>  
 {error && (  
 <div className="error-message" role="alert">  
 {error}  
 </div>  
 )}  
 {/\* Component content \*/}  
 </div>  
 );  
}

## **Security Standards**

### **Authentication**

// Good - Secure token handling  
export function useAccessToken(): string | null {  
 const [token, setToken] = useState<string | null>(null);  
  
 useEffect(() => {  
 // Never log tokens in production  
 if (process.env.NODE\_ENV === 'development') {  
 console.log('Token acquired');  
 }  
   
 // Store tokens securely  
 // Don't expose tokens in component props or context  
 }, []);  
  
 return token;  
}

### **API Security**

// Good - API route security  
export async function POST(request: NextRequest) {  
 // Always validate authorization  
 const authHeader = request.headers.get('authorization');  
   
 if (!authHeader || !authHeader.startsWith('Bearer ')) {  
 return NextResponse.json(  
 { error: 'Invalid authorization header' },  
 { status: 401 }  
 );  
 }  
  
 // Validate and sanitize input  
 const body = await request.json();  
 const { prompt } = body;  
   
 if (!prompt || typeof prompt !== 'string') {  
 return NextResponse.json(  
 { error: 'Invalid prompt' },  
 { status: 400 }  
 );  
 }  
  
 // Continue with secure request handling  
}

**Data Sanitization**

// Good - Input sanitization  
function sanitizeInput(input: string): string {  
 return input  
 .trim()  
 .replace(/[<>\"']/g, '') // Remove potentially harmful characters  
 .substring(0, 1000); // Limit length  
}  
  
// Good - Validate data types  
function validateChatMessage(data: unknown): ChatMessage {  
 if (!data || typeof data !== 'object') {  
 throw new Error('Invalid message data');  
 }  
  
 const message = data as Record<string, unknown>;  
   
 if (typeof message.content !== 'string' || !message.content.trim()) {  
 throw new Error('Message content is required');  
 }  
  
 return {  
 content: sanitizeInput(message.content),  
 role: message.role === 'assistant' ? 'assistant' : 'user',  
 timestamp: new Date()  
 };  
}

## **Testing Standards**

### **Unit Test Structure**

// Good - Test file structure (useAccessToken.test.ts)  
import { renderHook, waitFor } from '@testing-library/react';  
import { useAccessToken } from '@/app/hooks/useAccessToken';  
  
describe('useAccessToken', () => {  
 beforeEach(() => {  
 // Setup test environment  
 });  
  
 afterEach(() => {  
 // Cleanup  
 });  
  
 it('should return null initially', () => {  
 const { result } = renderHook(() => useAccessToken());  
 expect(result.current).toBeNull();  
 });  
  
 it('should acquire token when user is authenticated', async () => {  
 // Test implementation  
 });  
  
 it('should handle token acquisition errors gracefully', async () => {  
 // Test error scenarios  
 });  
});

### **Component Testing**

// Good - Component test structure  
import { render, screen, fireEvent, waitFor } from '@testing-library/react';  
import ChatInterface from '@/app/components/ChatInterface';  
  
describe('ChatInterface', () => {  
 const mockProps = {  
 onMessageSent: jest.fn()  
 };  
  
 beforeEach(() => {  
 jest.clearAllMocks();  
 });  
  
 it('should render input field and send button', () => {  
 render(<ChatInterface {...mockProps} />);  
   
 expect(screen.getByPlaceholderText('Type your message...')).toBeInTheDocument();  
 expect(screen.getByRole('button', { name: 'Send' })).toBeInTheDocument();  
 });  
  
 it('should handle form submission', async () => {  
 render(<ChatInterface {...mockProps} />);  
   
 const input = screen.getByPlaceholderText('Type your message...');  
 const sendButton = screen.getByRole('button', { name: 'Send' });  
   
 fireEvent.change(input, { target: { value: 'Test message' } });  
 fireEvent.click(sendButton);  
   
 await waitFor(() => {  
 expect(mockProps.onMessageSent).toHaveBeenCalledWith('Test message');  
 });  
 });  
});

## **Documentation Standards**

### **JSDoc Comments**

/\*\*  
 \* Custom hook for managing Azure AD access tokens  
 \*   
 \* @returns {string | null} The current access token or null if not authenticated  
 \*   
 \* @example  
 \* ```tsx  
 \* function MyComponent() {  
 \* const token = useAccessToken();  
 \*   
 \* if (!token) {  
 \* return <div>Please sign in</div>;  
 \* }  
 \*   
 \* return <div>Welcome!</div>;  
 \* }  
 \* ```  
 \*/  
export function useAccessToken(): string | null {  
 // Implementation  
}  
  
/\*\*  
 \* Sends a chat message to the AI completion API  
 \*   
 \* @param prompt - The user's message to send  
 \* @param token - Valid Azure AD access token  
 \* @returns Promise resolving to the AI's response  
 \*   
 \* @throws {Error} When authentication fails or API request fails  
 \*/  
export async function sendChatMessage(  
 prompt: string,  
 token: string  
): Promise<CompletionResponse> {  
 // Implementation  
}

### **README Updates** Code Quality - Run `npm run lint` before committing - Run `npm run format` to auto-format code - Run `npm run type-check` to verify TypeScript

Adding New Features  
1. Create feature branch from `main`  
2. Follow established patterns in similar components  
3. Add proper TypeScript types  
4. Include unit tests for new functionality  
5. Update documentation as needed  
  
Code Review Checklist  
- Code follows established patterns  
- TypeScript types are properly defined  
- Error handling is implemented  
- Security considerations are addressed  
- Tests are included for new functionality

## **Enforcement and Tools**

### **Pre-commit Hooks (Husky + lint-staged)**

// package.json  
{  
 "husky": {  
 "hooks": {  
 "pre-commit": "lint-staged"  
 }  
 },  
 "lint-staged": {  
 "\*.{ts,tsx,js,jsx}": [  
 "eslint --fix",  
 "prettier --write",  
 "git add"  
 ],  
 "\*.{css,scss,md}": [  
 "prettier --write",  
 "git add"  
 ]  
 }  
}

### **VS Code Settings**

Create .vscode/settings.json:

{  
 "editor.formatOnSave": true,  
 "editor.defaultFormatter": "esbenp.prettier-vscode",  
 "editor.codeActionsOnSave": {  
 "source.fixAll.eslint": true,  
 "source.organizeImports": true  
 },  
 "typescript.preferences.quoteStyle": "double",  
 "javascript.preferences.quoteStyle": "double",  
 "eslint.validate": [  
 "javascript",  
 "javascriptreact",  
 "typescript",  
 "typescriptreact"  
 ]  
}

## **Package Dependencies**

### **Required Development Dependencies**

{  
 "devDependencies": {  
 "@typescript-eslint/eslint-plugin": "^6.0.0",  
 "@typescript-eslint/parser": "^6.0.0",  
 "eslint": "^8.0.0",  
 "eslint-config-next": "^14.0.0",  
 "prettier": "^3.0.0",  
 "husky": "^8.0.0",  
 "lint-staged": "^13.0.0",  
 "@testing-library/react": "^13.0.0",  
 "@testing-library/jest-dom": "^5.0.0",  
 "jest": "^29.0.0"  
 }  
}