Colors of Life - Frontend Development Guidelines

This document outlines the guidelines and best practices for frontend development on the Colors of Life platform. Following these guidelines will ensure consistency, maintainability, and a high-quality user experience.

Design System Implementation

Component Hierarchy

1. **Base UI Components**:
   - Pure, unstyled functional components
   - Highly reusable across the entire application
   - Examples: Button, Input, Card, Typography
2. **Compound Components**:
   - Combinations of base components that form a more complex UI element
   - Examples: SearchBar, ProductCard, NavigationMenu
3. **Feature Components**:
   - Components specific to a feature or domain
   - Examples: KlingTryOnViewer, StyleQuiz, AvatarEditor

Styling Approach

1. **TailwindCSS**:
   - Use Tailwind utility classes for component styling
   - Create a consistent design system by extending the Tailwind theme
   - Utilize `clsx` or `tailwind-merge` for conditional styling
2. **Custom Design System**:
   - Extend Tailwind with our custom design tokens
   - Define colors, typography, spacing, and other design variables in a central location
   - Use CSS variables for theme values to support dark mode and customization
3. **Component Styling Structure**:

tsx

Copy

```tsx
// Preferred approach
function Button({ variant = 'primary', size = 'md', className, ...props }) {
  const baseStyles = 'rounded font-medium transition-colors';
  const variantStyles = {
    primary: 'bg-primary-600 text-white hover:bg-primary-700',
    secondary: 'bg-gray-100 text-gray-900 hover:bg-gray-200',
    // other variants...
  };
  const sizeStyles = {
    sm: 'text-sm px-3 py-1.5',
    md: 'text-base px-4 py-2',
    lg: 'text-lg px-5 py-2.5',
  };

  return (
    <button
      className={clsx(
        baseStyles,
        variantStyles[variant],
        sizeStyles[size],
        className
      )}
      {...props}
    />
  );
}
```

Apple-Like UI Guidelines

1. **Typography**:
   - Use SF Pro font family (system font on Apple devices)
   - Clear typographic hierarchy with limited font sizes
   - Consistent line heights and letter spacing
2. **Visual Design**:
   - Clean, minimalist interfaces with ample whitespace
   - Subtle shadows and depth cues
   - Limited color palette with purposeful accent colors
   - Rounded corners for UI elements (consistent border-radius)

3. **Animation and Transitions**:
    - Subtle, purpose-driven animations
    - Natural, physics-based transitions
    - Avoid flashy or distracting effects

State Management

Principles

1. **State Locality**:
    - Keep state as close as possible to where it's used
    - Use React's built-in state management for component-level state
    - Lift state up only when necessary
2. **Global State Management**:
    - Use React Context for theme, auth, and other app-wide concerns
    - Implement Zustand for more complex state management needs
    - Create separate stores for discrete domains (cart, user preferences, etc.)
3. **State Structure**:
    - Keep state normalized to avoid duplication
    - Use TypeScript interfaces to define state shape
    - Document state structure with comments

Implementation Example

tsx

Copy

```tsx
// Auth store with Zustand
import create from 'zustand';
import { persist } from 'zustand/middleware';

interface User {
  id: string;
  name: string;
  email: string;
  // other user properties...
}

interface AuthState {
  user: User | null;
  token: string | null;
  isLoading: boolean;
  error: string | null;
  login: (email: string, password: string) => Promise<void>;
  logout: () => void;
  // other actions...
}

export const useAuthStore = create<AuthState>()(
  persist(
    (set, get) => ({
      user: null,
      token: null,
      isLoading: false,
      error: null,
      login: async (email, password) => {
        set({ isLoading: true, error: null });
        try {
          // API call to login
          const { user, token } = await loginUser(email, password);
          set({ user, token, isLoading: false });
        } catch (error) {
          set({ error: error.message, isLoading: false });
        }
      },
      logout: () => {
        set({ user: null, token: null });
      },
      // other actions...
```

```
    }),
    {
      name: 'auth-storage',
      partialize: (state) => ({ user: state.user, token: state.token }),
    }
  )
);
```

Kling AI Integration Components

Virtual Try-On Component

1. **KlingTryOnViewer**:
   - Core component for displaying try-on results
   - Handles loading, error, and success states
   - Provides controls for view manipulation

2. **Implementation Guidelines**:

tsx

Copy

```tsx
// Example implementation structure
function KlingTryOnViewer({
  productId,
  userImageId,
  onSuccess,
  onError,
  ...props
}) {
  const [status, setStatus] = useState('idle'); // idle, loading, success, error
  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (productId && userImageId) {
      setStatus('loading');

      // Call to our backend service that interfaces with Kling API
      tryOnService
        .createTryOnTask(productId, userImageId)
        .then(taskId => pollTaskStatus(taskId))
        .then(result => {
          setResult(result);
          setStatus('success');
          onSuccess?.(result);
        })
        .catch(err => {
          setError(err);
          setStatus('error');
          onError?.(err);
        });
    }
  }, [productId, userImageId]);

  // Render appropriate UI based on status
  if (status === 'loading') {
    return <TryOnLoadingState />;
  }

  if (status === 'error') {
    return <TryOnErrorState error={error} />;
  }
```

```
  if (status === 'success') {
    return <TryOnResultView result={result} {...props} />;
  }

  return <TryOnInitialState />;
}
```

3. **Loading States**:
   - Implement visually appealing loading indicators
   - Provide status updates during the try-on process
   - Consider skeleton loaders for a smoother experience
4. **Error Handling**:
   - Display user-friendly error messages
   - Provide appropriate recovery actions
   - Log detailed errors for debugging

Performance Optimization

Code Splitting

1. **Route-Based Splitting**:
   - Leverage Next.js automatic route-based code splitting
   - Use dynamic imports for large component trees
2. **Component-Level Splitting**:
   - Use dynamic imports for large, non-critical components

tsx Copy

```
const KlingTryOnViewer = dynamic(() => import('@/components/KlingTryOnViewer'), {
  loading: () => <TryOnSkeleton />,
  ssr: false, // Disable for components that only work client-side
});
```

Rendering Optimization

1. **Memoization**:
   - Use React.memo for components that render often but with the same props
   - Use useMemo for expensive calculations
   - Use useCallback for functions passed as props

2. **Virtualization**:
    - Implement virtualized lists for long scrollable content
    - Consider react-virtualized or react-window for efficient rendering
3. **Image Optimization**:
    - Use Next.js Image component for automatic optimization
    - Implement lazy loading for images below the fold
    - Provide appropriate sizes and formats
    - Optimize product images before sending to Kling AI

Data Fetching

1. **GraphQL Practices**:
    - Request only the data you need
    - Use fragments to share common field selections
    - Implement Apollo Client cache effectively
2. **SWR/React Query**:
    - Use for REST API calls with automatic caching and revalidation
    - Implement optimistic updates for better UX
3. **Kling API Integration**:
    - Implement request batching when possible
    - Cache try-on results to prevent redundant requests
    - Prioritize critical requests

Accessibility Standards

Core Requirements

1. **Semantic HTML**:
    - Use appropriate HTML elements for their intended purpose
    - Ensure proper heading hierarchy
    - Use landmarks for major page sections (header, main, nav, footer)
2. **Keyboard Navigation**:
    - Ensure all interactive elements are keyboard accessible
    - Implement logical tab order
    - Provide visible focus states for all interactive elements

- Support keyboard shortcuts for power users

3. **Screen Readers**:
   - Add appropriate ARIA attributes when necessary
   - Include alt text for all images
   - Ensure form elements have associated labels
   - Test with screen readers regularly
4. **Focus Management**:
   - Trap focus in modals and dialogs
   - Return focus to triggering element when closing overlays
   - Avoid focus loss during dynamic content updates

Implementation Examples

tsx

Copy

```tsx
// Accessible Button Component
function Button({ children, isLoading, ...props }) {
  return (
    <button
      className="..."
      disabled={isLoading}
      aria-busy={isLoading}
      {...props}
    >
      {isLoading ? (
        <>
          <span className="sr-only">Loading</span>
          <Spinner aria-hidden="true" />
        </>
      ) : (
        children
      )}
    </button>
  );
}

// Accessible Modal Component
function Modal({ isOpen, onClose, title, children }) {
  const modalRef = useRef(null);

  useEffect(() => {
    const handleEscape = (e) => {
      if (e.key === 'Escape' && isOpen) {
        onClose();
      }
    };

    document.addEventListener('keydown', handleEscape);
    return () => document.removeEventListener('keydown', handleEscape);
  }, [isOpen, onClose]);

  useEffect(() => {
    if (isOpen && modalRef.current) {
      const focusableElements = modalRef.current.querySelectorAll(
        'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
      );
      if (focusableElements.length) {
```

```
        focusableElements[0].focus();
      }
    }
  }, [isOpen]);

  if (!isOpen) return null;

  return (
    <div
      className="modal-backdrop"
      onClick={onClose}
      role="presentation"
    >
      <div
        ref={modalRef}
        className="modal-content"
        role="dialog"
        aria-modal="true"
        aria-labelledby="modal-title"
        onClick={(e) => e.stopPropagation()}
      >
        <h2 id="modal-title">{title}</h2>
        <div>{children}</div>
        <button
          className="close-button"
          onClick={onClose}
          aria-label="Close"
        >
          <CloseIcon aria-hidden="true" />
        </button>
      </div>
    </div>
  );
}
```

Responsive Design

Mobile-First Approach

1. **Base Styles**:
   - Start with mobile layouts and progressively enhance for larger screens
   - Use Tailwind's responsive modifiers (sm:, md:, lg:, xl:)

- Implement critical styles for mobile experience first

2. **Breakpoints**:
   - Follow standard Tailwind breakpoints or customize if needed:
     - sm: 640px (small devices)
     - md: 768px (medium devices)
     - lg: 1024px (large devices)
     - xl: 1280px (extra large devices)
     - 2xl: 1536px (2x extra large devices)

3. **Responsive Typography**:
   - Implement fluid typography that scales with viewport
   - Use clamp() for font-size where appropriate
   - Maintain reasonable line lengths on all screen sizes

Touch Optimization

1. **Touch Targets**:
   - Minimum touch target size of 44x44 pixels
   - Adequate spacing between interactive elements
   - Consider thumb zones on mobile devices

2. **Touch-Friendly Interactions**:
   - Implement swipe gestures for key interactions
   - Provide haptic feedback where appropriate
   - Optimize for both touch and pointer devices

Style Stream Implementation

Video Feed Components

1. **Structure**:
   - Implement using virtualized lists for performance
   - Use intersection observers for autoplay management
   - Support vertical swipe navigation

2. **Performance Considerations**:
   - Lazy load videos
   - Unload off-screen content

- Use appropriate video compression

3. **Component Example**:

tsx Copy

```tsx
function StyleStreamFeed() {
  const { data, fetchNextPage } = useInfiniteQuery(
    'styleStreamFeed',
    ({ pageParam = 1 }) => fetchFeedItems(pageParam),
    {
      getNextPageParam: (lastPage) => lastPage.nextCursor,
    }
  );

  const items = useMemo(() =>
    data?.pages.flatMap(page => page.items) || [],
    [data]
  );

  return (
    <div className="style-stream-container h-full w-full">
      <FeedVirtualList
        items={items}
        onEndReached={() => fetchNextPage()}
        renderItem={(item) => (
          <StyleStreamItem
            key={item.id}
            item={item}
            onTryOn={() => handleTryOn(item)}
          />
        )}
      />
    </div>
  );
}
```

Try-On Integration

1. **Quick Try-On Action**:
   - Implement one-tap try-on from Style Stream
   - Show try-on result as overlay or modal
   - Provide seamless transition to product detail

2. **Implementation Example**:

tsx Copy

```tsx
function StyleStreamItem({ item, onTryOn }) {
  // Component implementation
  return (
    <div className="relative h-full w-full">
      <video
        src={item.videoUrl}
        className="h-full w-full object-cover"
        loop
        muted
        playsInline
        // Other video props
      />

      <div className="absolute bottom-6 right-4 flex flex-col gap-4">
        <Button
          variant="circle"
          aria-label="Like"
          icon={<HeartIcon />}
        />
        <Button
          variant="circle"
          aria-label="Try On"
          icon={<TryOnIcon />}
          onClick={() => onTryOn(item.productId)}
        />
        <Button
          variant="circle"
          aria-label="Share"
          icon={<ShareIcon />}
        />
      </div>

      <div className="absolute bottom-6 left-4">
        <h3 className="text-white text-lg font-semibold">{item.brandName}</h3>
        <p className="text-white text-sm">{item.productName}</p>
        <span className="text-white font-bold">${item.price}</span>
      </div>
    </div>
  );
}
```

Testing Strategy

Unit Testing

1. **Component Testing**:
   - Test each component in isolation
   - Use React Testing Library to focus on user behavior
   - Test all component variations (props, states)
2. **Hook Testing**:
   - Create dedicated tests for custom hooks
   - Test with different input parameters
   - Verify state updates and side effects

Integration Testing

1. **User Flows**:
   - Test complete user journeys (e.g., onboarding, checkout)
   - Verify component interactions
   - Test with realistic data scenarios
2. **API Integration**:
   - Mock API responses for consistent tests
   - Test error handling and loading states
   - Verify that UI updates correctly with data changes
3. **Kling API Testing**:
   - Create mock responses for Kling API
   - Test different response scenarios (success, failure, timeout)
   - Validate UI state during long-running operations

Visual Regression Testing

1. **Implementation**:
   - Use Chromatic or Percy for visual regression tests
   - Take snapshots of key UI states
   - Compare across browsers and viewport sizes
2. **Process**:

- Run visual tests on all PRs
- Review changes before approving
- Maintain a visual history of component evolution

Performance Monitoring

1. **Core Web Vitals**:
   - Monitor LCP (Largest Contentful Paint)
   - Track FID (First Input Delay)
   - Measure CLS (Cumulative Layout Shift)
2. **Custom Metrics**:
   - Time to interactive for key features
   - Animation smoothness (FPS)
   - Try-on request latency and success rate
3. **User-Centric Metrics**:
   - Time to first meaningful action
   - Complete try-on experience time
   - Search to results time

Documentation

Component Documentation

1. **Structure**:
   - Purpose and usage
   - Props API with types and defaults
   - Example usage
   - Accessibility considerations
   - Performance notes
2. **Tools**:
   - Use Storybook for interactive documentation
   - Include code snippets for common use cases
   - Document component variants and states

Code Comments

1. **When to Comment**:
   - Complex business logic
   - Non-obvious optimizations
   - Workarounds or browser-specific code
   - Important architectural decisions
2. **Comment Style**:
   - Be concise and clear
   - Explain why, not what (the code shows what)
   - Use JSDoc for function documentation

Code Review Guidelines

1. **Checklist**:
   - Performance implications
   - Accessibility compliance
   - Mobile responsiveness
   - Browser compatibility
   - Error handling
   - Type safety
   - Integration with Kling API implementation
2. **Process**:
   - Pull request template with key areas to address
   - Required reviewer approvals (2 minimum)
   - Automated checks must pass before review
   - Constructive, specific feedback

Development Workflow

1. **Feature Development**:
   - Create feature branch from main
   - Implement with TDD approach when possible
   - Create pull request with comprehensive description
   - Address review feedback
   - Merge to main upon approval

2. **Release Process**:
    - Automated releases from main branch
    - Feature flags for gradual rollout
    - Canary deployments for high-risk changes
    - Monitoring for post-deployment issues

Collaborative Tools

1. **Design Collaboration**:
    - Figma for UI design and prototyping
    - Design token synchronization
    - Component reference library
2. **Development Tools**:
    - Cursor as primary code editor
    - Shared ESLint and Prettier configurations
    - Husky pre-commit hooks for consistent code quality

Browser Support

- Modern evergreen browsers (Chrome, Firefox, Safari, Edge)
- Latest two major versions of each browser
- iOS Safari 14+
- Android Chrome 90+
- No support for Internet Explorer

Conclusion

These frontend guidelines aim to create a consistent, high-quality, and maintainable codebase for the Colors of Life platform. By following these practices, we'll deliver an exceptional user experience while maintaining development efficiency and code quality.

Remember that guidelines should evolve with the project, and improvements to these standards are encouraged as the platform and team develop.