Future-Proof Context System - Infinite Scalability Architecture

🎯 CONTEXT SYSTEM PHILOSOPHY

Infinite Scalability: The system automatically adapts to any number of contexts without requiring code changes to existing components.

Hierarchical Inheritance: Contexts can inherit from parent contexts, enabling sophisticated organizational structures.

Dynamic Registration: New contexts can be registered at runtime, supporting plugin-based architectures and customer-specific customizations.

Backward Compatibility: Existing components continue to work seamlessly as new contexts are added.

🏗️ CORE ARCHITECTURE

Dynamic Context Registry

typescript

```typescript
// src/contexts/ContextRegistry.ts
import { EventEmitter } from 'events'

export interface ContextFeature {
  id: string
  name: string
  description?: string
  permissions?: string[]
  dependencies?: string[]
  enabled?: boolean
}

export interface ContextTheme {
  id: string
  name: string
  colors: Record<string, string>
  typography?: Record<string, any>
  spacing?: Record<string, any>
  components?: Record<string, any>
  dark?: boolean
}

export interface ContextDefinition {
  id: string
  name: string
  description: string
  version: string

  // Visual identity
  icon?: string
  color?: string
  logo?: string

  // Inheritance
  parentContext?: string
  childContexts?: string[]

  // Theming
  themes: ContextTheme[]
  defaultTheme?: string

  // Features and permissions
  features: ContextFeature[]
  permissions: string[]

  // Behavioral configuration
  navigation?: NavigationConfig
  layout?: LayoutConfig
  branding?: BrandingConfig

  // Lifecycle
  createdAt: Date
  updatedAt: Date
  deprecated?: boolean
  deprecationDate?: Date

  // Custom properties
  metadata?: Record<string, any>

  // Validation
  validate?: (context: ContextDefinition) => boolean
}

export interface NavigationConfig {
  primaryNav?: NavigationItem[]
  secondaryNav?: NavigationItem[]
  footerNav?: NavigationItem[]
  breadcrumbStyle?: 'simple' | 'complex' | 'hierarchical'
}

export interface NavigationItem {
  id: string
  label: string
  href?: string
  icon?: string
  children?: NavigationItem[]
  permissions?: string[]
  feature?: string
}

export interface LayoutConfig {
  sidebar?: {
    position: 'left' | 'right' | 'none'
    collapsible: boolean
    defaultCollapsed?: boolean
    width?: number
  }
```

```typescript
  header?: {
    height: number
    fixed: boolean
    showLogo: boolean
    showUserMenu: boolean
  }
  footer?: {
    enabled: boolean
    height?: number
    content?: string
  }
}

export interface BrandingConfig {
  logo?: string
  favicon?: string
  primaryColor?: string
  secondaryColor?: string
  fontFamily?: string
  customCSS?: string
}

class ContextRegistry extends EventEmitter {
  private contexts: Map<string, ContextDefinition> = new Map()
  private contextHierarchy: Map<string, string[]> = new Map()
  private themeCache: Map<string, ContextTheme> = new Map()
  private featureCache: Map<string, ContextFeature[]> = new Map()
  private validationRules: Map<string, (context: ContextDefinition) => boolean> = new Map()

  constructor() {
    super()
    this.initializeCore()
  }

  private initializeCore() {
    // Register core validation rules
    this.validationRules.set('id', (ctx) => /^[a-z][a-z0-9-]*$/.test(ctx.id))
    this.validationRules.set('name', (ctx) => ctx.name.length > 0)
    this.validationRules.set('themes', (ctx) => ctx.themes.length > 0)
    this.validationRules.set('circular', (ctx) => !this.hasCircularDependency(ctx))
  }

  // Registration methods
  register(context: ContextDefinition): void {
    this.validateContext(context)

    const existingContext = this.contexts.get(context.id)
    if (existingContext) {
      console.warn(`Context ${context.id} already exists, updating...`)
    }

    this.contexts.set(context.id, {
      ...context,
      updatedAt: new Date(),
      createdAt: existingContext?.createdAt || new Date()
    })

    this.updateHierarchy(context)
    this.updateCaches(context)

    this.emit('contextRegistered', context)
    this.emit('contextsChanged', this.getAllContexts())
  }

  unregister(contextId: string): void {
    const context = this.contexts.get(contextId)
    if (!context) {
      console.warn(`Context ${contextId} does not exist`)
      return
    }

    // Check for dependent contexts
    const dependents = this.getDependentContexts(contextId)
    if (dependents.length > 0) {
      console.warn(`Cannot unregister ${contextId}: has dependent contexts:`, dependents)
      return
    }

    this.contexts.delete(contextId)
    this.cleanupHierarchy(contextId)
    this.cleanupCaches(contextId)

    this.emit('contextUnregistered', contextId)
    this.emit('contextsChanged', this.getAllContexts())
  }

  // Retrieval methods
  getContext(contextId: string): ContextDefinition | undefined {
    return this.contexts.get(contextId)
  }

  getAllContexts(): ContextDefinition[] {
    return Array.from(this.contexts.values())
      .filter(context => !context.deprecated)
      .sort((a, b) => a.name.localeCompare(b.name))
  }

  getActiveContexts(): ContextDefinition[] {
    return this.getAllContexts().filter(context =>
      !context.deprecated &&
      context.features.some(f => f.enabled !== false)
    )
  }

  getContextsByFeature(featureId: string): ContextDefinition[] {
    return this.getAllContexts().filter(context =>
      context.features.some(f => f.id === featureId)
    )
  }

  // Hierarchy methods
  getContextHierarchy(contextId: string): ContextDefinition[] {
    const hierarchy: ContextDefinition[] = []
    let current = this.getContext(contextId)

    while (current) {
      hierarchy.unshift(current)
      current = current.parentContext ? this.getContext(current.parentContext) : undefined
    }

    return hierarchy
  }

  getChildContexts(contextId: string): ContextDefinition[] {
    const children = this.contextHierarchy.get(contextId) || []
    return children.map(id => this.getContext(id)).filter(Boolean) as ContextDefinition[]
  }

  getAllDescendants(contextId: string): ContextDefinition[] {
    const descendants: ContextDefinition[] = []
    const children = this.getChildContexts(contextId)
    children.forEach(child => {
      descendants.push(child)
      descendants.push(...this.getAllDescendants(child.id))
    })

    return descendants
  }

  // Theme methods
  getContextTheme(contextId: string, themeId?: string): ContextTheme | undefined {
    const context = this.getContext(contextId)
    if (!context) return undefined

    const targetThemeId = themeId || context.defaultTheme || context.themes[0]?.id
    return context.themes.find(theme => theme.id === targetThemeId)
  }

  getInheritedTheme(contextId: string, themeId?: string): ContextTheme {
    const hierarchy = this.getContextHierarchy(contextId)
    const mergedTheme: ContextTheme = {
      id: themeId || 'inherited',
      name: 'Inherited Theme',
      colors: {}
    }

    // Merge themes from root to leaf
    hierarchy.forEach(context => {
      const theme = this.getContextTheme(context.id, themeId)
      if (theme) {
        Object.assign(mergedTheme.colors, theme.colors)
        Object.assign(mergedTheme, {
          typography: { ...mergedTheme.typography, ...theme.typography },
          spacing: { ...mergedTheme.spacing, ...theme.spacing },
          components: { ...mergedTheme.components, ...theme.components }
        })
      }
    })

    return mergedTheme
  }

  // Feature methods
  getContextFeatures(contextId: string): ContextFeature[] {
    const cached = this.featureCache.get(contextId)
```

```
    if (cached) return cached

    const hierarchy = this.getContextHierarchy(contextId)
    const features: ContextFeature[] = []
    const featureMap = new Map<string, ContextFeature>()

    // Collect features from hierarchy (parent to child)
    hierarchy.forEach(context => {
      context.features.forEach(feature => {
        featureMap.set(feature.id, { ...feature })
      })
    })

    const result = Array.from(featureMap.values())
    this.featureCache.set(contextId, result)
    return result
  }

  hasFeature(contextId: string, featureId: string): boolean {
    const features = this.getContextFeatures(contextId)
    return features.some(f => f.id === featureId && f.enabled !== false)
  }

  // Permission methods
  getContextPermissions(contextId: string): string[] {
    const hierarchy = this.getContextHierarchy(contextId)
    const permissions = new Set<string>()

    hierarchy.forEach(context => {
      context.permissions.forEach(permission => permissions.add(permission))
    })

    return Array.from(permissions)
  }

  hasPermission(contextId: string, permission: string): boolean {
    const permissions = this.getContextPermissions(contextId)
    return permissions.includes(permission)
  }

  // Validation methods
  private validateContext(context: ContextDefinition): void {
    const errors: string[] = []
    this.validationRules.forEach((rule, ruleName) => {
      try {
        if (!rule(context)) {
          errors.push(`Validation failed for rule: ${ruleName}`)
        }
      } catch (error) {
        errors.push(`Error in validation rule ${ruleName}: ${error.message}`)
      }
    })

    if (errors.length > 0) {
      throw new Error(`Context validation failed: ${errors.join(', ')}`)
    }
  }

  private hasCircularDependency(context: ContextDefinition): boolean {
    const visited = new Set<string>()
    const recursionStack = new Set<string>()

    const hasCycle = (contextId: string): boolean => {
      if (recursionStack.has(contextId)) return true
      if (visited.has(contextId)) return false

      visited.add(contextId)
      recursionStack.add(contextId)

      const ctx = this.getContext(contextId)
      if (ctx?.parentContext) {
        if (hasCycle(ctx.parentContext)) return true
      }

      recursionStack.delete(contextId)
      return false
    }

    return hasCycle(context.id)
  }

  // Utility methods
  private updateHierarchy(context: ContextDefinition): void {
    if (context.parentContext) {
      const siblings = this.contextHierarchy.get(context.parentContext) || []
      if (!siblings.includes(context.id)) {
        siblings.push(context.id)
        this.contextHierarchy.set(context.parentContext, siblings)
      }
    }
  }

  private updateCaches(context: ContextDefinition): void {
    // Clear related caches
    this.themeCache.delete(context.id)
    this.featureCache.delete(context.id)

    // Clear descendant caches
    this.getAllDescendants(context.id).forEach(descendant => {
      this.themeCache.delete(descendant.id)
      this.featureCache.delete(descendant.id)
    })
  }

  private cleanupHierarchy(contextId: string): void {
    // Remove from parent's children
    this.contextHierarchy.forEach((children, parentId) => {
      const index = children.indexOf(contextId)
      if (index > -1) {
        children.splice(index, 1)
        if (children.length === 0) {
          this.contextHierarchy.delete(parentId)
        }
      }
    })

    // Remove own hierarchy entry
    this.contextHierarchy.delete(contextId)
  }

  private cleanupCaches(contextId: string): void {
    this.themeCache.delete(contextId)
    this.featureCache.delete(contextId)
  }

  private getDependentContexts(contextId: string): string[] {
    return Array.from(this.contexts.values())
      .filter(context => context.parentContext === contextId)
      .map(context => context.id)
  }
```

```
// Query methods
query(filters: {
  feature?: string
  permission?: string
  parent?: string
  theme?: string
  metadata?: Record<string, any>
}): ContextDefinition[] {
  return this.getAllContexts().filter(context => {
    if (filters.feature && !this.hasFeature(context.id, filters.feature)) {
      return false
    }

    if (filters.permission && !this.hasPermission(context.id, filters.permission)) {
      return false
    }

    if (filters.parent && context.parentContext !== filters.parent) {
      return false
    }

    if (filters.theme && !context.themes.some(t => t.id === filters.theme)) {
      return false
    }

    if (filters.metadata) {
      for (const [key, value] of Object.entries(filters.metadata)) {
        if (context.metadata?.[key] !== value) {
          return false
        }
      }
    }

    return true
  })
}

// Bulk operations
bulkRegister(contexts: ContextDefinition[]): void {
  // Sort by dependency order
  const sortedContexts = this.topologicalSort(contexts)

  sortedContexts.forEach(context => {
    try {
      this.register(context)
    } catch (error) {
      console.error(`Failed to register context ${context.id}:`, error)
    }
  })
}

private topologicalSort(contexts: ContextDefinition[]): ContextDefinition[] {
  const visited = new Set<string>()
  const result: ContextDefinition[] = []
  const contextMap = new Map(contexts.map(c => [c.id, c]))

  const visit = (context: ContextDefinition) => {
    if (visited.has(context.id)) return

    visited.add(context.id)

    if (context.parentContext) {
      const parent = contextMap.get(context.parentContext)
      if (parent) {
        visit(parent)
      }
    }

    result.push(context)
  }

  contexts.forEach(visit)
  return result
}

// Serialization
export(): string {
  const data = {
    contexts: Array.from(this.contexts.values()),
    hierarchy: Object.fromEntries(this.contextHierarchy),
    exportedAt: new Date().toISOString()
  }

  return JSON.stringify(data, null, 2)
}

import(data: string): void {
  try {
      const parsed = JSON.parse(data)

      if (parsed.contexts) {
        this.bulkRegister(parsed.contexts)
      }

      if (parsed.hierarchy) {
        this.contextHierarchy = new Map(Object.entries(parsed.hierarchy))
      }

      this.emit('contextsImported', parsed)
    } catch (error) {
      console.error('Failed to import contexts:', error)
      throw error
    }
  }
}

export const contextRegistry = new ContextRegistry()

// Export singleton instance
export { contextRegistry as default }
```

Context Provider System

typescript

```tsx
// src/contexts/WorkspaceContext.tsx
import React, { createContext, useContext, useEffect, useState, useCallback } from 'react'
import { contextRegistry, ContextDefinition, ContextTheme, ContextFeature } from './ContextRegistry'

interface WorkspaceContextValue {
  // Current state
  currentContext: string
  contextDefinition: ContextDefinition | null
  currentTheme: string
  theme: ContextTheme | null

  // Available options
  availableContexts: ContextDefinition[]
  availableThemes: ContextTheme[]

  // Actions
  switchContext: (contextId: string) => Promise<void>
  switchTheme: (themeId: string) => void

  // Utilities
  isContextAvailable: (contextId: string) => boolean
  hasFeature: (featureId: string) => boolean
  hasPermission: (permission: string) => boolean
  getContextHierarchy: () => ContextDefinition[]
  getInheritedTheme: () => ContextTheme

  // Advanced features
  canSwitchTo: (contextId: string) => boolean
  getContextMetadata: (key: string) => any
  isDescendantOf: (ancestorId: string) => boolean
}

const WorkspaceContext = createContext<WorkspaceContextValue | null>(null)

export const useWorkspaceContext = () => {
  const context = useContext(WorkspaceContext)
  if (!context) {
    throw new Error('useWorkspaceContext must be used within a WorkspaceProvider')
  }
  return context
}

interface WorkspaceProviderProps {
  children: React.ReactNode
  initialContext?: string
  initialTheme?: string
  availableContexts?: string[]
  contextSwitchValidator?: (fromContext: string, toContext: string) => boolean
  onContextSwitch?: (fromContext: string, toContext: string) => void
  onThemeSwitch?: (fromTheme: string, toTheme: string) => void
}

export const WorkspaceProvider: React.FC<WorkspaceProviderProps> = ({
  children,
  initialContext = 'default',
  initialTheme = 'light',
  availableContexts,
  contextSwitchValidator,
  onContextSwitch,
  onThemeSwitch
}) => {
  const [currentContext, setCurrentContext] = useState(initialContext)
  const [currentTheme, setCurrentTheme] = useState(initialTheme)
  const [allContexts, setAllContexts] = useState<ContextDefinition[]>([])
  const [loading, setLoading] = useState(false)

  // Update contexts when registry changes
  useEffect(() => {
    const updateContexts = (contexts: ContextDefinition[]) => {
      setAllContexts(contexts)
    }

    updateContexts(contextRegistry.getAllContexts())

    const handleContextsChanged = (contexts: ContextDefinition[]) => {
      updateContexts(contexts)
    }

    contextRegistry.on('contextsChanged', handleContextsChanged)
    return () => {
      contextRegistry.off('contextsChanged', handleContextsChanged)
    }
  }, [])

  // Validate current context exists
  useEffect(() => {
    const contextExists = contextRegistry.getContext(currentContext)
  if (!contextExists && allContexts.length > 0) {
    console.warn(`Context ${currentContext} not found, falling back to first available`)
    setCurrentContext(allContexts[0].id)
  }
}, [currentContext, allContexts])

// Computed values
const contextDefinition = contextRegistry.getContext(currentContext)
const theme = contextRegistry.getInheritedTheme(currentContext, currentTheme)

const availableContextDefinitions = availableContexts
  ? allContexts.filter(ctx => availableContexts.includes(ctx.id))
  : allContexts

const availableThemes = contextDefinition?.themes || []

// Actions
const switchContext = useCallback(async (contextId: string) => {
  if (contextId === currentContext) return

  const targetContext = contextRegistry.getContext(contextId)
  if (!targetContext) {
    console.warn(`Context ${contextId} not found`)
    return
  }

  // Validate switch if validator provided
  if (contextSwitchValidator && !contextSwitchValidator(currentContext, contextId)) {
    console.warn(`Context switch from ${currentContext} to ${contextId} not allowed`)
    return
  }

  setLoading(true)

  try {
    const oldContext = currentContext
    setCurrentContext(contextId)

    // Update theme if current theme not available in new context
    const newContextThemes = targetContext.themes.map(t => t.id)
    if (!newContextThemes.includes(currentTheme)) {
      const newTheme = targetContext.defaultTheme || targetContext.themes[0]?.id
      if (newTheme) {
        setCurrentTheme(newTheme)
        onThemeSwitch?.(currentTheme, newTheme)
      }
    }

    onContextSwitch?.(oldContext, contextId)
  } finally {
    setLoading(false)
  }
}, [currentContext, currentTheme, contextSwitchValidator, onContextSwitch, onThemeSwitch])

const switchTheme = useCallback((themeId: string) => {
  if (themeId === currentTheme) return

  const contextThemes = contextDefinition?.themes || []
  const themeExists = contextThemes.some(t => t.id === themeId)

  if (!themeExists) {
    console.warn(`Theme ${themeId} not available in context ${currentContext}`)
    return
  }

  const oldTheme = currentTheme
  setCurrentTheme(themeId)
  onThemeSwitch?.(oldTheme, themeId)
}, [currentTheme, contextDefinition, currentContext, onThemeSwitch])

// Utilities
const isContextAvailable = useCallback((contextId: string) => {
  return availableContextDefinitions.some(ctx => ctx.id === contextId)
}, [availableContextDefinitions])

const hasFeature = useCallback((featureId: string) => {
  return contextRegistry.hasFeature(currentContext, featureId)
}, [currentContext])

const hasPermission = useCallback((permission: string) => {
  return contextRegistry.hasPermission(currentContext, permission)
}, [currentContext])

const getContextHierarchy = useCallback(() => {
  return contextRegistry.getContextHierarchy(currentContext)
}, [currentContext])

const getInheritedTheme = useCallback(() => {
    return contextRegistry.getInheritedTheme(currentContext, currentTheme)
  }, [currentContext, currentTheme])

  const canSwitchTo = useCallback((contextId: string) => {
    if (!isContextAvailable(contextId)) return false
    if (contextSwitchValidator) {
      return contextSwitchValidator(currentContext, contextId)
    }
    return true
  }, [currentContext, isContextAvailable, contextSwitchValidator])

  const getContextMetadata = useCallback((key: string) => {
    return contextDefinition?.metadata?.[key]
  }, [contextDefinition])

  const isDescendantOf = useCallback((ancestorId: string) => {
    const hierarchy = getContextHierarchy()
    return hierarchy.some(ctx => ctx.id === ancestorId)
  }, [getContextHierarchy])

  const contextValue: WorkspaceContextValue = {
    currentContext,
    contextDefinition,
    currentTheme,
    theme,
    availableContexts: availableContextDefinitions,
    availableThemes,
    switchContext,
    switchTheme,
    isContextAvailable,
    hasFeature,
    hasPermission,
    getContextHierarchy,
    getInheritedTheme,
    canSwitchTo,
    getContextMetadata,
    isDescendantOf
  }

  return (
    <WorkspaceContext.Provider value={contextValue}>
      <div
        className={`workspace-context workspace-${currentContext}`}
        data-context={currentContext}
        data-theme={currentTheme}
        data-loading={loading}
        style={{
          // Apply CSS variables from theme
          ...Object.entries(theme?.colors || {}).reduce((acc, [key, value]) => {
            acc[`--color-${key}`] = value
            return acc
          }, {} as Record<string, string>)
        }}
      >
        {children}
      </div>
    </WorkspaceContext.Provider>
  )
}
```

🎨 DYNAMIC STYLING SYSTEM

Context-Aware Styling Engine

typescript

```
// src/styling/ContextStyling.ts
import { contextRegistry } from '../contexts/ContextRegistry'

export interface StyleVariant {
  name: string
  className: string
  styles: Record<string, string>
  conditions?: {
    context?: string[]
    theme?: string[]
    feature?: string[]
    permission?: string[]
  }
}

export interface ComponentStyleConfig {
  base: string
  variants: Record<string, StyleVariant[]>
  contextOverrides: Record<string, {
    base?: string
    variants?: Record<string, Partial<StyleVariant>>
  }>
}

class ContextStylingEngine {
  private styleConfigs: Map<string, ComponentStyleConfig> = new Map()
  private generatedStyles: Map<string, string> = new Map()

  registerComponent(componentName: string, config: ComponentStyleConfig): void {
    this.styleConfigs.set(componentName, config)
    this.generateContextStyles(componentName)
  }

  getComponentClasses(
    componentName: string,
    variant: string,
    context: string,
    theme: string = 'light'
  ): string {
    const config = this.styleConfigs.get(componentName)
    if (!config) return ''

    const classes: string[] = []
    // Base classes
    classes.push(config.base)

    // Context-specific base override
    const contextOverride = config.contextOverrides[context]
    if (contextOverride?.base) {
      classes.push(contextOverride.base)
    }

    // Variant classes
    const variants = config.variants[variant] || []
    const applicableVariants = variants.filter(v =>
      this.isVariantApplicable(v, context, theme)
    )

    applicableVariants.forEach(v => {
      classes.push(v.className)
    })

    // Context-specific variant overrides
    const variantOverride = contextOverride?.variants?.[variant]
    if (variantOverride?.className) {
      classes.push(variantOverride.className)
    }

    return classes.join(' ')
  }

  private isVariantApplicable(
    variant: StyleVariant,
    context: string,
    theme: string
  ): boolean {
    const conditions = variant.conditions
    if (!conditions) return true

    if (conditions.context && !conditions.context.includes(context)) {
      return false
    }

    if (conditions.theme && !conditions.theme.includes(theme)) {
      return false
    }
    if (conditions.feature) {
      const hasAllFeatures = conditions.feature.every(feature =>
        contextRegistry.hasFeature(context, feature)
      )
      if (!hasAllFeatures) return false
    }

    if (conditions.permission) {
      const hasAllPermissions = conditions.permission.every(permission =>
        contextRegistry.hasPermission(context, permission)
      )
      if (!hasAllPermissions) return false
    }

    return true
  }

  private generateContextStyles(componentName: string): void {
    const config = this.styleConfigs.get(componentName)
    if (!config) return

    const contexts = contextRegistry.getAllContexts()
    const cssRules: string[] = []

    contexts.forEach(context => {
      const contextThemes = context.themes

      contextThemes.forEach(theme => {
        const selector = `.workspace-${context.id}[data-theme="${theme.id}"]`

        // Generate CSS variables for theme
        const cssVariables = Object.entries(theme.colors || {})
          .map(([key, value]) => `--color-${key}: ${value}`)
          .join('; ')

        if (cssVariables) {
          cssRules.push(`${selector} { ${cssVariables} }`)
        }

        // Generate component-specific styles
        Object.entries(config.variants).forEach(([variantName, variants]) => {
          variants.forEach(variant => {
            if (this.isVariantApplicable(variant, context.id, theme.id)) {
              const componentSelector = `${selector} .${componentName}-${variantName}`
              const styles = Object.entries(variant.styles)
                .map(([property, value]) => `${property}: ${value}`)
                .join('; ')

              if (styles) {
                cssRules.push(`${componentSelector} { ${styles} }`)
              }
            }
          })
        })
      })
    })

    this.generatedStyles.set(componentName, cssRules.join('\n'))
   }

   generateCSS(): string {
    const allStyles: string[] = []

    this.generatedStyles.forEach((styles, componentName) => {
      allStyles.push(`/* ${componentName} */`)
      allStyles.push(styles)
      allStyles.push('')
    })

    return allStyles.join('\n')
   }

   regenerateStyles(): void {
    this.styleConfigs.forEach((config, componentName) => {
      this.generateContextStyles(componentName)
    })
   }
  }

  export const contextStyling = new ContextStylingEngine()

  // Listen for context changes and regenerate styles
  contextRegistry.on('contextsChanged', () => {
   contextStyling.regenerateStyles()
  })
```

Enhanced CVA Integration

typescript

```typescript
// src/styling/ContextAwareVariants.ts
import { cva, type VariantProps } from 'class-variance-authority'
import { contextStyling } from './ContextStyling'
import { useWorkspaceContext } from '../contexts/WorkspaceContext'

export interface ContextAwareVariantConfig {
  base: string | string[]
  variants?: Record<string, Record<string, string | string[]>>
  contextVariants?: Record<string, {
    base?: string | string[]
    variants?: Record<string, Record<string, string | string[]>>
  }>
  compoundVariants?: Array<{
    context?: string | string[]
    theme?: string | string[]
    [key: string]: any
    class: string | string[]
  }>
  defaultVariants?: Record<string, any>
}

export const contextAwareCva = (config: ContextAwareVariantConfig) => {
  const baseVariants = cva(config.base, {
    variants: config.variants,
    compoundVariants: config.compoundVariants,
    defaultVariants: config.defaultVariants
  })

  return (props: any & { context?: string; theme?: string }) => {
    const { context, theme, ...variantProps } = props

    // Get base classes
    let classes = baseVariants(variantProps)

    // Add context-specific classes
    if (context && config.contextVariants?.[context]) {
      const contextConfig = config.contextVariants[context]

      if (contextConfig.base) {
        classes += ` ${Array.isArray(contextConfig.base) ? contextConfig.base.join(' ') : contextConfig.base}`
      }

      if (contextConfig.variants) {
        Object.entries(variantProps).forEach(([key, value]) => {
          const contextVariant = contextConfig.variants?.[key]?.[value as string]
          if (contextVariant) {
            classes += ` ${Array.isArray(contextVariant) ? contextVariant.join(' ') : contextVariant}`
          }
        })
      }
    }

    // Add compound variant classes
    if (config.compoundVariants) {
      config.compoundVariants.forEach(compound => {
        const { context: contextMatch, theme: themeMatch, class: compoundClass, ...compoundProps } = compound

        // Check if compound variant applies
        let applies = true

        if (contextMatch) {
          const contexts = Array.isArray(contextMatch) ? contextMatch : [contextMatch]
          applies = applies && contexts.includes(context)
        }

        if (themeMatch) {
          const themes = Array.isArray(themeMatch) ? themeMatch : [themeMatch]
          applies = applies && themes.includes(theme)
        }

        // Check other variant conditions
        Object.entries(compoundProps).forEach(([key, value]) => {
          applies = applies && variantProps[key] === value
        })

        if (applies) {
          classes += ` ${Array.isArray(compoundClass) ? compoundClass.join(' ') : compoundClass}`
        }
      })
    }

    return classes
  }
}

// Hook for using context-aware variants
export const useContextAwareVariants = () => {
  const { currentContext, currentTheme } = useWorkspaceContext()

  return {
    currentContext,
    currentTheme,
    getVariantClasses: (config: ContextAwareVariantConfig, props: any) => {
      const variantFunction = contextAwareCva(config)
      return variantFunction({ ...props, context: currentContext, theme: currentTheme })
    }
  }
}
```

🔧 UTILITY FUNCTIONS

Context Management Utilities

typescript

```typescript
// src/utils/contextUtils.ts
import { contextRegistry, ContextDefinition } from '../contexts/ContextRegistry'

export const contextUtils = {
  // Context queries
  findContexts: (query: {
    name?: string
    feature?: string
    permission?: string
    parentId?: string
  }) => {
    return contextRegistry.query(query)
  },

  // Context relationships
  getContextPath: (contextId: string): string => {
    const hierarchy = contextRegistry.getContextHierarchy(contextId)
    return hierarchy.map(ctx => ctx.name).join(' > ')
  },

  getContextDepth: (contextId: string): number => {
    const hierarchy = contextRegistry.getContextHierarchy(contextId)
    return hierarchy.length
  },

  // Context validation
  isValidContextTransition: (fromContext: string, toContext: string): boolean => {
    const from = contextRegistry.getContext(fromContext)
    const to = contextRegistry.getContext(toContext)

    if (!from || !to) return false

    // Custom validation logic can be added here
    return true
  },

  // Context features
  getSharedFeatures: (contextIds: string[]): string[] => {
    if (contextIds.length === 0) return []

    const featureSets = contextIds.map(id =>
      new Set(contextRegistry.getContextFeatures(id).map(f => f.id))
    )
    return Array.from(featureSets[0]).filter(feature =>
      featureSets.every(set => set.has(feature))
    )
  },

  // Context permissions
  getEffectivePermissions: (contextId: string, userId?: string): string[] => {
    const contextPermissions = contextRegistry.getContextPermissions(contextId)

    // In a real implementation, you'd also check user-specific permissions
    // For now, just return context permissions
    return contextPermissions
  },

  // Context metadata
  getContextMetadata: (contextId: string, key?: string): any => {
    const context = contextRegistry.getContext(contextId)
    if (!context) return null

    return key ? context.metadata?.[key] : context.metadata
  },

  // Context search
  searchContexts: (searchTerm: string): ContextDefinition[] => {
    const term = searchTerm.toLowerCase()
    return contextRegistry.getAllContexts().filter(context =>
      context.name.toLowerCase().includes(term) ||
      context.description.toLowerCase().includes(term) ||
      context.features.some(f => f.name.toLowerCase().includes(term))
    )
  },

  // Context analytics
  getContextUsageStats: (contextId: string): {
    totalUsers: number
    activeUsers: number
    lastAccessed: Date
    popularFeatures: string[]
  } => {
    // This would integrate with your analytics system
    return {
      totalUsers: 0,
      activeUsers: 0,
        lastAccessed: new Date(),
        popularFeatures: []
      }
    }
  }
```

Component Integration Utilities

typescript

```
// src/utils/componentUtils.ts
import { useWorkspaceContext } from '../contexts/WorkspaceContext'
import { contextUtils } from './contextUtils'

export const useContextAwareComponent = (componentName: string) => {
  const {
    currentContext,
    currentTheme,
    hasFeature,
    hasPermission,
    getContextMetadata
  } = useWorkspaceContext()

  return {
    // Context information
    context: currentContext,
    theme: currentTheme,

    // Feature checking
    hasFeature,
    hasPermission,

    // Metadata access
    getMetadata: getContextMetadata,

    // Component-specific utilities
    getComponentConfig: (configKey: string) => {
      return getContextMetadata(`components.${componentName}.${configKey}`)
    },

    // Styling utilities
    getContextClass: (baseClass: string) => {
      return `${baseClass} ${baseClass}--${currentContext} ${baseClass}--${currentTheme}`
    },

    // Conditional rendering
    renderIfFeature: (featureId: string, component: React.ReactNode) => {
      return hasFeature(featureId) ? component : null
    },

    renderIfPermission: (permission: string, component: React.ReactNode) => {
      return hasPermission(permission) ? component : null
    },
    // Context-aware props
    getContextProps: () => ({
      'data-context': currentContext,
      'data-theme': currentTheme,
      'data-component': componentName
    })
  }
}

// Higher-order component for context awareness
export const withContextAwareness = <P extends object>(
  Component: React.ComponentType<P>
) => {
  return React.forwardRef<any, P>((props, ref) => {
    const contextProps = useContextAwareComponent(Component.displayName || 'Unknown')

    return (
      <Component
        ref={ref}
        {...props}
        {...contextProps.getContextProps()}
      />
    )
  })
}
```

🎯 REAL-WORLD USAGE EXAMPLES

Registering Core Contexts

typescript

```
// src/contexts/coreContexts.ts
import { contextRegistry } from './ContextRegistry'

// Register core contexts
contextRegistry.bulkRegister([
  {
    id: 'consultant',
    name: 'Consultant',
    description: 'Primary business context for consultants',
    version: '1.0.0',
    icon: 'briefcase',
    color: '#3B82F6',
    themes: [
      {
        id: 'light',
        name: 'Light Theme',
        colors: {
          primary: '#3B82F6',
          secondary: '#6B7280',
          background: '#FFFFFF',
          foreground: '#111827'
        }
      },
      {
        id: 'dark',
        name: 'Dark Theme',
        colors: {
          primary: '#60A5FA',
          secondary: '#9CA3AF',
          background: '#111827',
          foreground: '#F9FAFB'
        },
        dark: true
      }
    ],
    defaultTheme: 'light',
    features: [
      {
        id: 'workspace-management',
        name: 'Workspace Management',
        description: 'Create and manage workspaces',
        permissions: ['workspace.create', 'workspace.update', 'workspace.delete'],
        enabled: true
      },
      {
        id: 'client-management',
        name: 'Client Management',
        description: 'Manage client relationships',
        permissions: ['client.create', 'client.update', 'client.view'],
        enabled: true
      }
    ],
    permissions: [
      'workspace.create',
      'workspace.update',
      'workspace.delete',
      'client.create',
      'client.update',
      'client.view'
    ],
    navigation: {
      primaryNav: [
        { id: 'dashboard', label: 'Dashboard', href: '/dashboard', icon: 'home' },
        { id: 'clients', label: 'Clients', href: '/clients', icon: 'users' },
        { id: 'projects', label: 'Projects', href: '/projects', icon: 'folder' }
      ]
    },
    layout: {
      sidebar: {
        position: 'left',
        collapsible: true,
        defaultCollapsed: false,
        width: 240
      },
      header: {
        height: 64,
        fixed: true,
        showLogo: true,
        showUserMenu: true
      }
    },
    branding: {
      primaryColor: '#3B82F6',
      secondaryColor: '#6B7280',
      fontFamily: 'Inter, sans-serif'
    },
    createdAt: new Date(),
      updatedAt: new Date(),
      metadata: {
        tier: 'professional',
        maxWorkspaces: 10,
        maxClients: 100
      }
    },

    {
      id: 'client',
      name: 'Client Portal',
      description: 'Client-facing portal context',
      version: '1.0.0',
      icon: 'user',
      color: '#10B981',
      themes: [
        {
          id: 'light',
          name: 'Light Theme',
          colors: {
            primary: '#10B981',
            secondary: '#6B7280',
            background: '#FFFFFF',
            foreground: '#111827'
          }
        }
      ],
      defaultTheme: 'light',
      features: [
        {
          id: 'project-tracking',
          name: 'Project Tracking',
          description: 'Track project progress',
          permissions: ['project.view'],
          enabled: true
        },
        {
          id: 'document-access',
          name: 'Document Access',
          description: 'Access project documents',
          permissions: ['document.view'],
          enabled: true
        }
      ],
      permissions: ['project.view', 'document.view'],
      navigation: {
        primaryNav: [
          { id: 'overview', label: 'Overview', href: '/overview', icon: 'eye' },
          { id: 'projects', label: 'My Projects', href: '/projects', icon: 'folder' },
          { id: 'documents', label: 'Documents', href: '/documents', icon: 'file' }
        ]
      },
      layout: {
        sidebar: {
          position: 'left',
          collapsible: false,
          width: 200
        },
        header: {
          height: 64,
          fixed: true,
          showLogo: true,
          showUserMenu: true
        }
      },
      branding: {
        primaryColor: '#10B981',
        secondaryColor: '#6B7280',
        fontFamily: 'Inter, sans-serif'
      },
      createdAt: new Date(),
      updatedAt: new Date(),
      metadata: {
        tier: 'client',
        readonly: true
      }
    },

    // Enterprise child context
    {
      id: 'enterprise',
      name: 'Enterprise',
      description: 'Enterprise-level consultant context',
      version: '1.0.0',
      parentContext: 'consultant',
      icon: 'building',
      color: '#7C3AED',
      themes: [
      {
        id: 'light',
        name: 'Enterprise Light',
        colors: {
          primary: '#7C3AED',
          secondary: '#6B7280',
          background: '#FFFFFF',
          foreground: '#111827'
        }
      }
    ],
    defaultTheme: 'light',
    features: [
      {
        id: 'advanced-analytics',
        name: 'Advanced Analytics',
        description: 'Advanced reporting and analytics',
        permissions: ['analytics.advanced'],
        enabled: true
      },
      {
        id: 'white-label',
        name: 'White Label',
        description: 'Custom branding options',
        permissions: ['branding.customize'],
        enabled: true
      }
    ],
    permissions: ['analytics.advanced', 'branding.customize'],
    navigation: {
      primaryNav: [
        { id: 'dashboard', label: 'Dashboard', href: '/dashboard', icon: 'home' },
        { id: 'analytics', label: 'Analytics', href: '/analytics', icon: 'bar-chart' },
        { id: 'branding', label: 'Branding', href: '/branding', icon: 'palette' }
      ]
    },
    layout: {
      sidebar: {
        position: 'left',
        collapsible: true,
        defaultCollapsed: false,
        width: 280
      }
    },
          createdAt: new Date(),
          updatedAt: new Date(),
          metadata: {
            tier: 'enterprise',
            maxWorkspaces: 100,
            maxClients: 1000,
            customBranding: true
          }
        }
      ])
```

Dynamic Context Registration

typescript

```typescript
// src/contexts/dynamicContexts.ts
import { contextRegistry } from './ContextRegistry'

// Function to register contexts from API
export const loadContextsFromAPI = async () => {
  try {
    const response = await fetch('/api/contexts')
    const contexts = await response.json()

    contextRegistry.bulkRegister(contexts)

    console.log(`Loaded ${contexts.length} contexts from API`)
  } catch (error) {
    console.error('Failed to load contexts from API:', error)
  }
}

// Function to register user-specific contexts
export const registerUserContexts = async (userId: string) => {
  try {
    const response = await fetch(`/api/users/${userId}/contexts`)
    const userContexts = await response.json()

    userContexts.forEach(context => {
      contextRegistry.register({
        ...context,
        id: `user-${userId}-${context.id}`,
        name: `${context.name} (Personal)`,
        metadata: {
          ...context.metadata,
          userId,
          personal: true
        }
      })
    })

    console.log(`Registered ${userContexts.length} user contexts`)
  } catch (error) {
    console.error('Failed to register user contexts:', error)
  }
}

// Function to register organization contexts
```

```
export const registerOrgContexts = async (orgId: string) => {
  const orgContexts = [
    {
      id: `org-${orgId}-admin`,
      name: 'Organization Admin',
      description: 'Administrative context for organization',
      version: '1.0.0',
      parentContext: 'admin',
      icon: 'shield',
      color: '#DC2626',
      themes: [
        {
          id: 'light',
          name: 'Admin Light',
          colors: {
            primary: '#DC2626',
            secondary: '#6B7280',
            background: '#FFFFFF',
            foreground: '#111827'
          }
        }
      ],
      defaultTheme: 'light',
      features: [
        {
          id: 'org-management',
          name: 'Organization Management',
          description: 'Manage organization settings',
          permissions: ['org.manage'],
          enabled: true
        }
      ],
      permissions: ['org.manage'],
      createdAt: new Date(),
      updatedAt: new Date(),
      metadata: {
        organizationId: orgId,
        tier: 'admin'
      }
    }
  ]
```

```
    contextRegistry.bulkRegister(orgContexts)
  }
```

This future-proof context system provides infinite scalability while maintaining backward compatibility and performance. New contexts can be added dynamically without affecting existing components, and the hierarchical system allows for sophisticated organizational structures.