Epic 1.1: Monorepo Architecture Setup

Epic Overview

This epic establishes the foundational monorepo structure for THE WHEEL design system, organizing the existing 85% complete component library into a scalable, maintainable architecture.

Priority: P0 (Critical)
Timeline: 2-3 weeks
Dependencies: None (Foundation Epic)

Story 1.1.1: Package Structure Migration

Overview

Restructure the existing component library into a well-organized monorepo with clear package boundaries and dependencies.

AI Developer Prompt

You are implementing the monorepo restructuring for THE WHEEL design system. You have an existing 85% complete component library that needs to be extracted into a proper monorepo structure.

Context

- Existing codebase has components in src/components/ui/, src/components/common/, src/components/layout/
- You have a sophisticated theming system already implemented
- Real-time collaboration features are already built
- User context management is already working

Requirements

1. Create Nx workspace monorepo structure with these packages:

- **packages/ui/** - Extract from src/components/ui/
  - Core UI components (buttons, inputs, cards, etc.)
  - Base component interfaces and types
  - Component-specific styles and assets
- **packages/patterns/** - Extract from src/components/common/

    - Common design patterns
    - Composite components
    - Reusable component combinations
- **packages/layouts/** - Extract from src/components/layout/
    - Layout components and systems
    - Grid and spacing utilities
    - Responsive layout managers
- **packages/themes/** - Extract theming system
    - Theme provider and context
    - CSS variable management
    - Theme switching logic
    - Workspace-specific theme variations
- **packages/workspace/** - New workspace-specific components
    - Workspace context providers
    - Permission management components
    - Workspace-specific UI elements
- **packages/shared/** - Utilities and contexts
    - Shared TypeScript types
    - Common utilities and helpers
    - Shared contexts and hooks

2. Preserve all existing functionality during migration

- No breaking changes to component APIs
- Maintain all existing props and behaviors
- Preserve theme system functionality
- Keep real-time features working

3. Maintain TypeScript strict mode compliance

- All packages must use strict TypeScript
- Proper type exports from each package
- No implicit any types
- Complete type coverage

4. Ensure all imports are updated throughout codebase

- Update relative imports to package imports
- Fix all import paths in components
- Update test imports
- Update Storybook story imports

5. Configure proper package.json for each package

- Correct dependencies and peerDependencies
- Proper build scripts
- Package metadata
- Export configurations

Specific Tasks

typescript

```typescript
// Example package.json structure for packages/ui
{
  "name": "@wheel/ui",
  "version": "0.0.1",
  "main": "./dist/index.js",
  "types": "./dist/index.d.ts",
  "exports": {
    ".": {
      "import": "./dist/index.mjs",
      "require": "./dist/index.js"
    },
    "./styles": "./dist/styles.css"
  },
  "scripts": {
    "build": "tsup src/index.ts --format cjs,esm --dts",
    "dev": "tsup src/index.ts --format cjs,esm --dts --watch",
    "test": "jest"
  },
  "peerDependencies": {
    "react": "^18.0.0",
    "react-dom": "^18.0.0"
  },
  "dependencies": {
    "@wheel/themes": "workspace:*",
    "@wheel/shared": "workspace:*"
  }
}
```

Documentation Required

- README.md for each package explaining:
  - Package purpose and scope
  - Installation instructions
  - Usage examples
  - API documentation
  - Contributing guidelines
- Migration guide documenting:
  - What components moved where
  - Import path changes

    - Breaking changes (if any)
    - Migration scripts available
- Updated development setup instructions
- Package dependency diagram
- Import/export reference guide

Testing Requirements

- All existing tests must pass after migration
- Package isolation tests
- Build system tests for each package
- Import/export validation tests
- Circular dependency detection tests
- Cross-package integration tests

Integration Points

- Maintain compatibility with existing Storybook setup
- Ensure theme system continues to work across packages
- Preserve real-time collaboration context providers
- Keep user management system functioning
- Maintain existing CI/CD pipeline compatibility

Deliverables

- Fully functional monorepo with proper package structure
- All existing components working in new structure
- Updated documentation and development workflow
- Comprehensive test suite validating migration
- Performance benchmarks showing no regression

Error Handling

- Graceful handling of missing imports during migration
- Fallback mechanisms for theme system during package restructuring
- Rollback plan if migration fails
- Import validation during build process

Performance Requirements

- Build time for full workspace under 2 minutes
- Individual package build under 30 seconds
- No runtime performance regression
- Tree-shaking effectiveness maintained
- Bundle size optimization preserved

Story 1.1.2: Build System Configuration

Overview

Configure a robust build system for the monorepo that handles TypeScript compilation, bundling, and optimization across all packages.

AI Developer Prompt

You are configuring the build system for THE WHEEL design system monorepo. Building on the package structure from Story 1.1.1, you need to create a robust build system that handles TypeScript compilation, bundling, and optimization.

Context

- Monorepo structure is now established with 6 packages
- Existing components are TypeScript-based with strict mode
- You have complex theming system with CSS variables
- Real-time features require WebSocket and event handling
- Performance is critical for consultant workspace applications

Requirements

1. Configure Nx build targets for each package:

- TypeScript compilation with project references
- Rollup/Vite bundling for production
- Development build with hot reloading
- CSS processing pipeline with PostCSS
- Asset optimization and tree shaking

2. Set up package-specific build configurations:

- **packages/ui/**: Pure component library build
- **packages/patterns/**: Complex component patterns
- **packages/layouts/**: Layout component builds
- **packages/themes/**: CSS variable processing
- **packages/workspace/**: Business logic components
- **packages/shared/**: Utility library builds

3. Configure environment-specific builds:

- **Development**: Source maps, hot reload, debug info
- **Production**: Minification, compression, tree shaking
- **Testing**: Coverage instrumentation, test utilities

Specific Tasks

javascript

```javascript
// Example Nx workspace configuration
{
  "version": 2,
  "projects": {
    "ui": {
      "root": "packages/ui",
      "sourceRoot": "packages/ui/src",
      "projectType": "library",
      "targets": {
        "build": {
          "executor": "@nrwl/vite:build",
          "options": {
            "outputPath": "dist/packages/ui",
            "main": "packages/ui/src/index.ts",
            "tsConfig": "packages/ui/tsconfig.lib.json",
            "assets": ["packages/ui/*.md"]
          }
        },
        "test": {
          "executor": "@nrwl/jest:jest",
          "options": {
            "jestConfig": "packages/ui/jest.config.js",
            "passWithNoTests": true
          }
        }
      }
    }
  }
}
```

Documentation Required

- Build system architecture documentation
- Package-specific build configuration guide
- Development vs production build differences
- Performance optimization techniques used
- Troubleshooting guide for common build issues
- Build artifacts and output structure explanation

Testing Requirements

- Build validation tests for each package
- Bundle size regression tests
- Tree shaking effectiveness tests
- Cross-package dependency resolution tests
- Build performance benchmarks
- Source map validation tests

Integration Points

- Integrate with existing Storybook build process
- Support for existing theme system CSS variables
- Maintain compatibility with real-time WebSocket features
- Support for existing testing framework
- CI/CD pipeline integration for automated builds

Deliverables

- Fully configured build system for all packages
- Package-specific build configurations
- Development and production build workflows
- Build performance optimization
- Comprehensive build documentation and tests

Performance Requirements

- Build time under 30 seconds for full workspace
- Bundle size under 2MB for complete UI package
- Tree shaking reduces bundle by 40%+ for typical usage
- Hot reload under 1 second for development
- Build caching reduces rebuild time by 70%+

Story 1.1.3: Component Inventory & Audit

Overview

Conduct a comprehensive audit of all existing components to identify gaps, enhancement opportunities, and create a complete inventory for the design system.

AI Developer Prompt

You are conducting a comprehensive audit of THE WHEEL design system components. Building on the monorepo structure from previous stories, you need to catalog all existing components and create an enhancement plan.

Context

- Monorepo structure is established with components distributed across packages
- You have 85% of components already built to production quality
- Existing components have sophisticated theming and real-time features
- Need to identify missing components and enhancement opportunities
- Components need workspace context awareness for multi-tenant functionality

Requirements

1. Create comprehensive component inventory:

- Catalog all components in each package
- Document current props, methods, and functionality
- Identify component maturity levels (alpha, beta, stable)
- Map component dependencies and relationships
- Assess workspace context compatibility

2. Gap analysis and enhancement planning:

- Identify 23 missing components from atomic design system
- Document enhancement needs for 89 existing components
- Create priority matrix (P0-P3) for development work
- Estimate effort required for each enhancement
- Plan workspace-specific component variants

3. Quality assessment:

- Evaluate accessibility compliance
- Check responsive design implementation

- Assess performance characteristics
- Review TypeScript type safety
- Validate design system consistency

Specific Tasks

typescript

```typescript
// Example component inventory structure
interface ComponentInventory {
  component: {
    name: string;
    package: string;
    category: 'atom' | 'molecule' | 'organism';
    status: 'alpha' | 'beta' | 'stable';
    workspaceContextSupport: boolean;
    accessibility: 'full' | 'partial' | 'none';
    responsive: boolean;
    performance: 'optimized' | 'needs-work' | 'not-tested';
    dependencies: string[];
    enhancements: Enhancement[];
    missingFeatures: string[];
  }
}

interface Enhancement {
  description: string;
  priority: 'P0' | 'P1' | 'P2' | 'P3';
  effort: 'S' | 'M' | 'L' | 'XL';
  category: 'feature' | 'performance' | 'accessibility' | 'design';
}
```

Documentation Required

- Complete component inventory spreadsheet
- Component dependency diagram
- Enhancement backlog with effort estimates
- Missing component specifications
- Quality assessment report
- Development roadmap based on findings

Testing Requirements

- Component API validation tests
- Dependency cycle detection tests
- Workspace context compatibility tests
- Performance benchmarks for each component
- Accessibility compliance audit
- Cross-browser compatibility assessment

Integration Points

- Document integration with existing theme system
- Identify real-time collaboration integration points
- Map user context dependencies
- Document Storybook integration requirements
- Assess CI/CD pipeline integration needs

Deliverables

- Comprehensive component inventory (156 components)
- Enhancement backlog with priorities and estimates
- Missing component specifications (23 components)
- Quality assessment report with recommendations
- Development roadmap for next 8 weeks
- Component documentation templates

Audit Criteria

- Component completeness and functionality
- Workspace context integration readiness
- Performance and accessibility compliance
- Design system consistency
- Documentation quality and completeness
- Test coverage and quality

Expected Findings

- **156 total components** identified across all packages

    - 45 atoms (buttons, inputs, typography, etc.)
    - 44 molecules (form fields, cards, etc.)
    - 48 organisms (navigation, tables, forms, etc.)
    - 19 workspace-specific components
- **23 missing components** to reach 100% coverage
- **89 components** needing enhancements
- **67 components** requiring accessibility improvements
- **45 components** needing performance optimization

Timeline and Dependencies

Timeline

- **Week 1-2**: Story 1.1.1 - Package Structure Migration
- **Week 2-3**: Story 1.1.2 - Build System Configuration
- **Week 3**: Story 1.1.3 - Component Inventory & Audit

Dependencies

- No external dependencies (foundation epic)
- Stories 1.1.2 and 1.1.3 depend on Story 1.1.1 completion

Success Metrics

- All components successfully migrated to monorepo
- Build times meet performance requirements
- Zero regression in functionality
- Complete component inventory delivered
- Clear roadmap for remaining 15% of components

Risk Mitigation

- Create backup of existing codebase before migration
- Implement migration in phases with validation
- Maintain parallel development environment
- Have rollback procedures ready
- Regular checkpoints with stakeholder approval