Technical Documentation: Presentation Template

Overview

The presentation template system is a sophisticated web application that allows users to create, edit, and manage presentation templates with customizable sections and pages. The system is built using Next.js, React, and Firebase, implementing a modular and flexible architecture.

Architecture

Core Components

1. Presentation Template Editor

- **Location**: `app/company-settings/presentations/[id]/EditPresentation.tsx`
- **Purpose**: Main interface for managing presentation templates
- **Key Features**:
  - Multi-page management
  - Drag-and-drop section reordering
  - Real-time preview
  - Unsaved changes detection
  - Page and section duplication

2. Section Templates

- **Location**: `app/components/presentation-templates/`
- **Structure**:

  ```
  presentation-templates/
  ├── SectionTemplatesEditor.tsx
  ├── section-templates/
  │   ├── BeforeAfterSlider.tsx
  │   ├── CenteredText.tsx
  │   ├── CoverPage.tsx
  │   ├── CustomHtmlCss.tsx
  │   └── ... (other section types)
  ├── StyledTextArea.tsx
  └── StyledInput.tsx
  ```

Data Structure

Presentation Template

```
interface OriginalData {
  name: string;
  quoteType: string;
  pages: Page[];
  sections: Section[];
}
```

Page

```
interface Page {
  id: string;
  title: string;
  sequence: number;
}
```

Section

```
interface Section {
  id: string;
  title: string;
  content: string;
  pageId: string;
  type: string;
  sequence: number;
  templateMetadata?: any;
  stylingMetadata?: any;
}
```

Key Technical Approaches

1. Shadow DOM for Preview

- Implementation of `ShadowDomPreview` component for isolated preview rendering
- Prevents style leakage between preview and editor
- Enables safe rendering of custom HTML/CSS

2. Security Measures

```
// DOMPurify Configuration
DOMPurify.setConfig({
  ADD_ATTR: ['target', 'viewBox', 'fill', 'd', 'xmlns', 'width', 'height'],
  ADD_TAGS: ['iframe', 'svg', 'path'],
  ALLOWED_TAGS: [...],
  FORBID_TAGS: ['script', 'style', 'form', 'input'],
  FORBID_ATTR: ['onerror', 'onload', 'onclick', 'onmouseover'],
  // ... other security configurations
});
```

3. State Management

- Complex state management using React hooks
- Implementation of unsaved changes detection
- Optimistic updates for better UX
- State persistence in Firebase

4. Drag and Drop Implementation

- Using `@hello-pangea/dnd` for drag-and-drop functionality
- Custom implementation for page and section reordering
- Sequence management for maintaining order

5. Modular Section System

- Each section type is a separate component
- Common styling and behavior patterns
- Extensible architecture for new section types
- Template metadata for section-specific configurations

Common Components

1. UI Components

- **Modal**: Reusable modal dialog
- **DropdownMenu**: Custom dropdown implementation
- **Confirm**: Confirmation dialog
- **Tabs**: Tabbed interface
- **Loader**: Loading state indicator

2. Form Components

- **StyledInput**: Custom styled input field
- **StyledTextArea**: Enhanced text area with formatting
- **Button**: Reusable button component

Best Practices Implemented

1. **Type Safety**

    - Comprehensive TypeScript interfaces
    - Strict type checking
    - Proper error handling

2. **Performance Optimization**

    - Memoization of expensive computations
    - Efficient state updates
    - Optimized re-rendering

3. **Security**

    - Input sanitization
    - XSS prevention
    - Safe HTML rendering

4. **User Experience**

    - Unsaved changes warning
    - Real-time preview
    - Intuitive drag-and-drop interface
    - Responsive design

Firebase Integration

Data Structure

```
companies/
    └── {companyId}/
    └── presentation_templates/
    └── {templateId}/
        ├── pages/
            └── sections/
```

Operations

- CRUD operations for templates, pages, and sections
- Real-time updates
- Efficient data fetching
- Proper error handling

Future Considerations

1. **Performance**

   - Implement virtualization for large presentations
   - Optimize image loading and processing
   - Add caching mechanisms

2. **Features**

   - Template versioning
   - Collaborative editing
   - Advanced styling options
   - Export functionality

3. **Maintenance**

   - Component documentation
   - Unit testing
   - Performance monitoring
   - Error tracking

Modular Architecture: Preview and Content Editing System

Design Philosophy

The Preview and content editing system was intentionally designed with modularity as a core principle. This architectural decision was driven by several key considerations:

1. Separation of Concerns

- **Preview Component (`ShadowDomPreview`)**

  - Isolated rendering environment
  - Independent styling context
  - Clean separation from editor logic
  - Prevents style conflicts with the main application

- **Content Editor (`SectionTemplatesEditor`)**

  - Focused on content manipulation
  - Dedicated state management
  - Specialized validation logic
  - Clean separation from preview rendering

2. Maintenance Benefits

- **Independent Updates**
    - Preview system can be updated without affecting the editor
    - Editor improvements don't require preview changes
    - Easier bug fixing in isolated components
    - Simpler testing and debugging
- **Code Organization**

```
components/
├── presentation-templates/
│   ├── SectionTemplatesEditor.tsx    # Editor logic
│   └── section-templates/            # Section-specific editors
└── common/
    └── ShadowDomPreview.tsx          # Preview rendering
```

3. Scalability Advantages

- **Easy Feature Addition**
    - New preview features can be added without editor changes
    - Editor enhancements don't impact preview functionality
    - Simple integration of new section types
    - Flexible styling system
- **Performance Optimization**
    - Independent rendering cycles
    - Optimized preview updates
    - Reduced unnecessary re-renders
    - Better resource management

4. Development Workflow

- **Parallel Development**
    - Multiple developers can work on different components
    - Reduced merge conflicts
    - Clearer responsibility boundaries
    - Faster development cycles
- **Testing Strategy**
    - Isolated unit tests for each component
    - Easier integration testing
    - Clear test boundaries
    - Simplified test maintenance

5. Future-Proofing

- **Technology Updates**
    - Easy to upgrade individual components
    - Simple to replace outdated technologies
    - Flexible to adopt new features
    - Minimal impact on other parts

- **Feature Evolution**
    - Preview system can evolve independently
    - Editor can be enhanced without preview changes
    - New section types can be added easily
    - Styling system can be updated separately

6. Real-World Benefits

- **Reduced Technical Debt**
    - Clear component boundaries
    - Easier to maintain
    - Simpler to refactor
    - Better code organization
- **Improved Developer Experience**
    - Clear component responsibilities
    - Easier onboarding
    - Better documentation
    - Simpler debugging

7. Implementation Example

```
// ShadowDomPreview.tsx - Isolated Preview Component
export default function ShadowDomPreview({ html, metadata }: PreviewProps) {
  return (
    <div className="preview-container">
      <div id="preview-root" />
      <style>{/* Isolated styles */}</style>
    </div>
  );
}

// SectionTemplatesEditor.tsx - Focused Editor Component
export default function SectionTemplatesEditor({
  sectionData,
  onSave
}: EditorProps) {
  // Editor-specific logic
  return (
    <div className="editor-container">
      {/* Editor UI */}
    </div>
  );
}
```

8. Maintenance Scenarios

1. **Adding New Features**
    - Preview: Add new preview capabilities without touching editor
    - Editor: Implement new editing features independently
    - Both: Maintain clean interfaces between components

2. **Updating Existing Features**

    - Preview: Update rendering without affecting editor
    - Editor: Enhance editing without impacting preview
    - Both: Maintain backward compatibility

3. **Fixing Issues**

    - Preview: Isolate and fix preview-specific bugs
    - Editor: Address editor-specific problems
    - Both: Clear separation of concerns for easier debugging

9. Best Practices Maintained

- **Clear Interfaces**

    - Well-defined props
    - Type-safe communication
    - Consistent patterns
    - Documented contracts

- **State Management**

    - Isolated state
    - Clear data flow
    - Predictable updates
    - Efficient rendering

This modular architecture has proven invaluable for maintaining and evolving the system, allowing for easier updates, better code organization, and improved development efficiency. The separation of preview and editing concerns has created a more maintainable and scalable codebase.