AI-Driven UI/UX Excellence: A Guide to Dash Mantine Components for Clean and Useful Dash Applications

1. Introduction: Guiding AI for Superior Dash UI/UX with Mantine Components

This document serves as a comprehensive guide for an Artificial Intelligence (AI) to develop clean, useful, and aesthetically pleasing Dash applications, specifically leveraging the dash-mantine-components (DMC) library with dash==3.0.4. The core objective is to translate established UI/UX best practices, as detailed in "The UI_UX Playbook", into actionable directives for AI-driven Dash development using DMC.

The user interface (UI) is paramount as it establishes the initial tone for the entire user experience (UX). Achieving a high-quality UI/UX is not a matter of chance but the outcome of meticulously applying fundamental design principles. For an AI, these principles, when mapped to specific DMC component usage and theming strategies, form the foundational "rules" for generating superior interfaces. It's crucial to recognize that an aesthetically stunning UI can be rendered ineffective by poor UX, such as convoluted navigation. Conversely, excellent UX paired with a subpar UI can make an application feel archaic and uninviting. The optimal user experience emerges when both UI and UX are executed proficiently. This guide aims to equip an AI with the knowledge to achieve this synergy.

The explicit instruction to target an AI necessitates a "declarative" style of guidance. This means the document will not only explain *what* constitutes good UI but will meticulously detail *how* to achieve it through specific DMC code patterns and prop configurations. This approach provides the AI with clear, unambiguous instructions, which is crucial for code generation. The user's request for a ".md file that can be used as useful project knowledge to help an ai create clean, and useful dash applications" underscores this need for a practical translation layer from abstract principles to concrete DMC implementations. For instance, the principle of "Visual Hierarchy" translates into directives such as "Utilize dmc.Title with the order prop and dmc.Text with appropriate size and weight props, ensuring the theme.headings object is configured for consistent heading styles."

Dash Mantine Components offers a rich suite of components and a robust theming system that aligns seamlessly with modern UI/UX paradigms. The focus on dash==3.0.4 and the version of DMC current at the time of documentation Browse implies that the AI's operational knowledge should be grounded in the features and props available within this specific technological context. Future updates to Dash or DMC may necessitate revisions to this guidance to maintain its utility.

This document will systematically explore key UI/UX domains—Theming, Layout, Data Display, Typography, and Data Visualization. Each section will reference principles from "The UI_UX Playbook" and map them to tangible DMC components, props, and theming configurations, thereby providing a structured and actionable knowledge base for the AI.

---

2. Core Theming with Dash Mantine Components

Theming is the cornerstone of a visually consistent and appealing application. Dash Mantine Components (DMC) provides a powerful and flexible theming system, primarily managed through the dmc.MantineProvider component. A well-defined theme is crucial for an AI to generate UIs that are not only aesthetically pleasing but also adhere to UX principles like readability and visual hierarchy with minimal per-instance component configuration.

At the heart of DMC theming is the dmc.MantineProvider. Every DMC application *must* be enveloped by a single instance of this component. dmc.MantineProvider is responsible for disseminating the theme throughout the application, managing color schemes (light/dark modes), and injecting global styles and CSS variables.

Key props for dmc.MantineProvider include:

- `theme` (dict): This is the central theme object where all design tokens and customizations are defined. It is deeply merged with Mantine's default theme, allowing for selective overrides.
- `forceColorScheme` (string: "light" or "dark"): This prop allows the AI to programmatically set the color scheme, which can be useful for generating consistent previews or for applications with specific display requirements, overriding user or system preferences.
- `withGlobalStyles` (boolean): When set to True (recommended), it applies Mantine's base global styles, contributing to a consistent look and feel.
- `withNormalizeCSS` (boolean): When set to True (recommended), it includes a CSS normalization stylesheet to reduce browser inconsistencies.
- `inherit` (boolean): If True, MantineProvider will attempt to inherit font and color styles from its parent element in the DOM tree.

2.1. Defining Your Color System

Color is a fundamental aspect of UI design, influencing mood, brand perception, and usability. Effective color systems prioritize clarity, contrast, and consistency.

Principles for Color Usage:

- **Moderation**: "Less is often more" is a key principle. Primary colors should be used thoughtfully, primarily to highlight interactive elements like buttons and links, rather than overwhelming the interface.
- **Backgrounds**: Neutral background colors (shades of grey, beige, or soft whites) often provide the best canvas, allowing content and primary colors to stand out. Dark backgrounds can also be highly effective, particularly for reducing eye strain in low-light conditions and making text pop.
- **Contrast**: Sufficient contrast between text and its background, and between different UI elements, is crucial for readability, guiding user attention, and ensuring accessibility for users with visual impairments.
- **Semantic Colors**: Adhere to established color conventions for system statuses: red for errors, green for success, and yellow for warnings, to provide immediate and intuitive feedback to the user.

DMC Implementation for Color Systems:

- `theme.primaryColor` (string): This theme property sets the main accent color for the application. Its value must be a key defined within `theme.colors` (e.g., "blue", or a custom-defined

"myBrandGreen"). This color will be used by default for many interactive components like `dmc.Button` when no specific color is provided.
    - **AI Directive**: The AI should define `primaryColor` based on the application's brand identity. This color should be consistently applied to primary call-to-action buttons, active navigation elements, and other key interactive components to guide user focus.
- **`theme.colors`** (dict): This dictionary is where custom color palettes are defined or existing Mantine default colors (like "blue", "red", "green") are overridden. Each entry in `theme.colors` should be a color name (string key) associated with an array of 10 string shades, indexed from 0 (lightest) to 9 (darkest). Providing all 10 shades is essential for components to render their variants (e.g., light, filled, outline) correctly across different states (hover, active).
    - Example: `theme={"colors": {"myBrandBlue": ["#E7F5FF", ..., "#1864AB"], "deepPurple": [...]}}`
    - **AI Directive**: For brand colors not available in Mantine's defaults, the AI must define them within `theme.colors`. It should ensure that each custom color array contains exactly 10 shades. If only a base hex color is available, tools like the Mantine color generator (mentioned in DMC documentation) can be used to generate the 10 shades.
- **`theme.primaryShade`** (number or dict): This property specifies which shade (index 0-9) of the `theme.primaryColor` is used by default. It can be a single number (e.g., `6`) to apply to both light and dark color schemes, or a dictionary (e.g., `{light: 6, dark: 8}`) to specify different shades for light and dark modes.
    - **AI Directive**: The AI should adjust `primaryShade` to ensure the primary color offers good visual appeal and, critically, sufficient contrast for text and icons on primary-colored backgrounds in both light and dark themes. This is vital for accessibility.
- **Applying Colors to Components**:
    - Individual components can have their color set using the `color` prop. This prop can accept a theme color name (e.g., `color="myBrandBlue"`), a theme color name with a specific shade index (e.g., `color="red.7"` for a darker red), or a direct CSS color value (e.g., `color="#FF5733"`).
    - **AI Directive**: For primary actions, the AI should use `color=theme.primaryColor` (or simply omit the `color` prop on components like `dmc.Button` if the `primaryColor` is intended). For semantic coloring (errors, warnings, success messages), the AI must use appropriate theme colors like `color="red"`, `color="yellow"`, `color="green"`, adhering to the color conventions outlined.
- **Mantine CSS Variables for Colors**: All theme colors, including custom ones, are automatically exposed as CSS variables. These variables follow a pattern like `var(--mantine-color-{colorName}-{shadeIndex})` (e.g., `var(--mantine-color-myBrandBlue-5)`) or `var(--mantine-primary-color-filled)` for variants of the primary color.
    - **AI Directive**: For custom CSS styling needs that go beyond standard component props, or when styling non-Mantine elements to match the theme, the AI should utilize these CSS variables to maintain color consistency.
- **Base and Contrast Colors**:
    - `theme.black` and `theme.white`: Define the absolute black and white colors for the theme.
    - `theme.autoContrast` (boolean) and `theme.luminanceThreshold` (number): When `autoContrast` is true, Mantine attempts to automatically adjust the text color (to black or white) on colored backgrounds (like filled buttons) to ensure sufficient contrast, based on the `luminanceThreshold`.

    - **AI Directive**: The AI should consider enabling `autoContrast` if the application uses a wide variety of background colors for interactive elements, as this helps maintain readability and accessibility, aligning with the contrast principles.

The careful definition and application of `theme.primaryColor`, `theme.colors`, and `theme.primaryShade` are fundamental. This allows the AI to achieve brand alignment while ensuring accessibility through appropriate color contrasts, especially when considering light and dark modes. The AI must understand that defining a comprehensive brand color palette in `theme.colors` (with 10 shades per color) is the first step. Then, selecting the `primaryColor` from these and fine-tuning `primaryShade` for different modes ensures that primary interactive elements are both on-brand and usable.

2.2. Typography System

Typography is a critical element of UI design, significantly impacting readability, user engagement, and the overall aesthetic appeal of an application.

Principles for Typography:

- **Font Selection**: The choice of font should be deliberate, considering the application's context (e.g., formal, tech-focused, creative). Prioritize readability and visual appeal. It's generally best to limit font choices to one or two families to maintain consistency and avoid a cluttered look.
- **Line Length**: Optimal line length enhances readability. For desktop interfaces, aim for 45-75 characters per line; for mobile, 30-40 characters is more appropriate.
- **Text Contrast**: Avoid using pure black text (`#000000`) on a white background or vice-versa. Opt for off-black or dark gray for headings and a slightly lighter gray for body text to reduce eye strain and improve visual harmony.
- **Font Weights**: Use heavier font weights (e.g., bold, semibold) for headings to establish hierarchy and draw attention. Lighter weights (e.g., regular) are suitable for paragraph text to ensure readability without competing with headings.
- **Size and Hierarchy**: Font size is a primary tool for communicating importance. A common standard for body text is 16px. Headings should be proportionally larger.
- **Line Height (Leading)**: Adequate line height is crucial for readability. For body text (around 16px), a line height of 1.5 to 1.6 times the font size (150%-160%) is recommended. Headings can often use slightly tighter line heights, while paragraphs benefit from more generous spacing.
- **Text Alignment**: For longer passages of text, left alignment is generally preferred in left-to-right languages as it provides a consistent starting point for each line, aiding readability.

DMC Implementation for Typography:

- **`theme.fontFamily`** (string): Defines the global font family for most components in the application. A common practice is to use a system font stack for broad compatibility and performance, or a specific web font like 'Inter'.
    - Example: `theme={"fontFamily": "Inter, sans-serif"}`
    - **AI Directive**: The AI should set `fontFamily` to a highly readable sans-serif font. Using system fonts (e.g., `-apple-system, BlinkMacSystemFont, Segoe UI, Roboto, ...`) is a good default for performance and native feel.
- **`theme.fontFamilyMonospace`** (string): Specifies the font family for components that display code, such as `dmc.Code`, `dmc.Kbd`, and `dmc.CodeHighlight`.

    - **AI Directive**: The AI should select a clear and legible monospace font stack (e.g., `ui-monospace, SFMono-Regular, Menlo,...`).
- **`theme.headings`** (dict): This object allows for global styling of `dmc.Title` components (which render as `<h1>`-`<h6>` HTML tags). It can include:
    - `fontFamily` (string): A specific font family for all headings, overriding `theme.fontFamily` if desired.
    - `fontWeight` (string | number): A default font weight for all headings.
    - `sizes` (dict): A nested dictionary to define specific `fontSize` and `lineHeight` (and optionally `fontWeight`) for each heading level from `h1` to `h6`.
        - Example: `theme={"headings": {"fontFamily": "Georgia, serif", "fontWeight": 700, "sizes": {"h1": {"fontSize": "2.5rem", "lineHeight": 1.3}, "h2": {"fontSize": "2rem", "lineHeight": 1.35, "fontWeight": 600},...}}}`
    - **AI Directive**: The AI must configure `theme.headings.sizes` to establish a clear visual hierarchy as emphasized in the design principles. Font sizes should decrease progressively from `h1` to `h6`. Line heights should generally be tighter for larger font sizes. Font weights can also be varied per heading level for further differentiation.
- **`theme.fontSizes`** (dict): Defines a scale of predefined font size tokens (typically 'xs', 'sm', 'md', 'lg', 'xl') that can be used by `dmc.Text` and other components via their `size` prop.
    - Example: `theme={"fontSizes": {"xs": "0.75rem", "sm": "0.875rem", "md": "1rem", "lg": "1.125rem", "xl": "1.25rem"}}` (Default values from)
    - **AI Directive**: The AI should ensure `theme.fontSizes.md` corresponds to approximately 16px (1rem), as this is a widely accepted standard for body text readability. Other sizes should be set proportionally to create a harmonious typographic scale.
- **`theme.lineHeights`** (dict): Defines a scale of predefined line height tokens (typically 'xs', 'sm', 'md', 'lg', 'xl').
    - Example: `theme={"lineHeights": {"xs": 1.4, "sm": 1.45, "md": 1.55, "lg": 1.6, "xl": 1.65}}` (Default values from)
    - **AI Directive**: The AI should set `theme.lineHeights.md` (for standard body text) to a value between 1.5 and 1.6 to ensure optimal readability, as recommended.

2.3. Spacing, Radius, and Shadows

Consistent spacing, appropriate corner rounding, and subtle use of shadows contribute significantly to a polished, organized, and easy-to-navigate UI.

Principles for Spacing, Radius, and Shadows:

- **Whitespace**: Also known as negative space, whitespace is crucial for giving design elements room to breathe. It enhances readability, guides the user's focus, and helps in segmenting content. A well-defined spacing system, often based on multiples of a base unit (e.g., 4px or 8px), should be established and used consistently.
- **Depth and Texture**: Shadows can be used to create a sense of depth, making interactive elements more noticeable and establishing a visual hierarchy. Soft shadows are generally preferred over harsh ones. Matching shadow color to the background can enhance cohesiveness. A consistent shadow system (e.g., different shadow strengths for different elevations) should be defined.
- **Shape Consistency**: Maintaining consistency in corner radii across elements like buttons, cards, and inputs contributes to a harmonious and professional look.

DMC Implementation for Spacing, Radius, and Shadows:

- `theme.spacing` (dict): This dictionary defines a scale of spacing tokens (e.g., 'xs', 'sm', 'md', 'lg', 'xl') that are used for margins, paddings, and gaps in layout components. These tokens typically map to pixel or rem values.
  - Example: `theme={"spacing": {"xs": "0.25rem", "sm": "0.5rem", "md": "1rem", "lg": "1.5rem", "xl": "2rem"}}` (1rem = 16px if base font size is 16px)
  - **AI Directive**: The AI must define `theme.spacing` using a consistent scale, preferably based on multiples of a base unit like 4px or 8px, as advocated for establishing a spacing system. These tokens should then be used for props like `m` (margin), `p` (padding) on various components, and `gap` or `gutter` in layout components like `dmc.Stack`, `dmc.Group`, and `dmc.Grid`.
- `theme.radius` (dict) and `theme.defaultRadius` (string): `theme.radius` defines a scale of border-radius tokens (e.g., 'xs', 'sm', 'md', 'lg', 'xl'), while `theme.defaultRadius` sets the default border-radius value (a key from `theme.radius`) that will be applied to most components unless overridden.
  - Example: `theme={"radius": {"sm": "0.25rem", "md": "0.5rem", "lg": "1rem"}, "defaultRadius": "md"}`
  - **AI Directive**: The AI should set a `defaultRadius` to ensure overall consistency in corner rounding across the application. These radius tokens should be applied to components like `dmc.Card`, `dmc.Button`, `dmc.Input`, and `dmc.Paper` via their respective `radius` prop. This directly supports the principle of shape and corner radius consistency.
- `theme.shadows` (dict): This dictionary defines a scale of box-shadow tokens (e.g., 'xs', 'sm', 'md', 'lg', 'xl') that can be applied to components to create an illusion of depth.
  - Example: `theme={"shadows": {"sm": "0 1px 3px rgba(0,0,0,0.05), 0 1px 2px rgba(0,0,0,0.1)", "md": "0 4px 6px -1px rgba(0,0,0,0.1), 0 2px 4px -1px rgba(0,0,0,0.06)"}}` (Values from Mantine's default shadows)
  - **AI Directive**: The AI should define a subtle and consistent shadow system within `theme.shadows`. These tokens can then be applied to components like `dmc.Card(shadow="sm")` or `dmc.Paper(shadow="md")` to create visual depth and hierarchy, as suggested for creating depth and texture. The AI should be guided to use softer shadows. While Mantine's default shadows are generally neutral, if highly colored backgrounds are used, the shadow definitions themselves might need to include a color tint to match the background, a more advanced technique.

2.4. Global Component Styling

Achieving consistency and simplicity in UI design is greatly aided by the ability to set global styles or default properties for components. This ensures that elements like buttons or cards have a uniform appearance and behavior throughout the application without repetitive styling.

Principles for Global Styling:

- **Consistency**: UI elements should function and appear in a similar manner across all parts of the application. This builds user trust, reduces cognitive load by making the interface predictable, and contributes to a cohesive aesthetic.
- **Simplicity**: Designs should be clean, prioritizing essential features and information. Reducing unnecessary variations in component styling contributes to this simplicity.

DMC Implementation via `theme.components` (dict): The `theme.components` key in the MantineProvider's `theme` object is a powerful mechanism for global component customization. It allows specifying default props and base styles for any DMC component.

- **Setting `defaultProps`**: This allows the AI to define default values for the props of any DMC component. When the component is used without these props explicitly set, the defaults from the theme will apply.
    - Example:

```
theme = {
    "components": {
        "Button": {
            "defaultProps": {
                "variant": "filled",
                "color": "primary", # Assumes primaryColor is
defined
                "radius": "md"      # Assumes 'md' is a key in
theme.radius
            }
        },
        "Card": {
            "defaultProps": {
                "shadow": "sm",     # Assumes 'sm' is a key in
theme.shadows
                "withBorder": True,
                "padding": "lg"     # Assumes 'lg' is a key in
theme.spacing
            }
        }
    }
}
```

    - **AI Directive**: To enforce consistency and simplicity, the AI should define `defaultProps` for frequently used components such as `dmc.Button`, `dmc.Card`, `dmc.TextInput`, `dmc.Title`, and `dmc.Text`. For instance, setting a default variant and radius for all buttons, or a default shadow and padding for all cards, ensures a uniform base look and feel. This reduces the amount of code the AI needs to generate for each component instance and makes global style changes easier by modifying the theme in one place.
- **Applying Global `styles` or `classNames` (Styles API)**: Beyond `defaultProps`, the `theme.components` key can also be used to apply global styles or CSS class names to specific internal selectors of components, leveraging Mantine's Styles API. This allows for more fine-grained control over the appearance of component parts.
    - Example:

```
theme = {
    "spacing": {"sm": "0.5rem"}, # Define for use below
    "colors": {"gray": ["#F8F9FA", ..., "#373A40"]}, # Define for
use below
```

```
    "components": {
        "Title": {
            # Apply lambda for dynamic styles or direct dict
            "styles": lambda theme, props: {
                "root": {"marginBottom": theme["spacing"]["sm"]}
            }
        },
        "TextInput": {
            # Apply lambda for dynamic styles or direct dict
            "styles": lambda theme, props: {
                "input": {"borderColor": theme["colors"]["gray"]
[4]}
            }
        }
    }
}
```

- **AI Directive**: For global styling of specific internal parts of components (e.g., uniformly changing the border color of all `dmc.TextInput` input elements, or adding consistent bottom margin to all `dmc.Title` components), the AI should use `theme.components.ComponentName.styles`. The `styles` value can be a dictionary or a function that receives the theme and component props, returning a style dictionary. The AI must refer to the individual DMC component documentation to find the available selectors for each component (e.g., `root`, `input`, `label` for `dmc.TextInput`; `root`, `label`, `inner` for `dmc.Button`).

The `theme.components` key is a particularly effective tool for AI-driven UI generation. By establishing "sane defaults," it significantly reduces the boilerplate code the AI needs to generate for each component instance. This not only leads to cleaner and more maintainable code but also inherently promotes a consistent UI, directly addressing the user's request for "clean and useful" applications.

Table 2.1: Core `dmc.MantineProvider` Theme Properties

Theme Key	Description	Example Value (Illustrative)	Relevant PDF Principle(s)
`primaryColor`	Sets the main accent color for the application. Must be a key from `theme.colors`.	`"blue"`, `"myCustomGreen"`	Color (p.65), Contrast (p.20), Visual Hierarchy (p.7)
`colors`	Defines custom color palettes or overrides existing ones. Each color is an array of 10 shades.	`{"myBrandBlue": ["#E7F5FF", ..., "#1864AB"]}`	Color (p.65-70), Contrast (p.21), Consistency (p.37)
`primaryShade`	Specifies which shade of `primaryColor` is used for light and dark modes.	`6` or `{"light": 6, "dark": 8}`	Color (p.65), Contrast (p.21)

Theme Key	Description	Example Value (Illustrative)	Relevant PDF Principle(s)
`fontFamily`	Global font family for most components.	`"Inter, sans-serif"`	Typography (p.71-74, Readability)
`headings`	Global styles for `dmc.Title` (h1-h6), including `fontFamily`, `fontWeight`, and `sizes` (for `fontSize`, `lineHeight` per level).	`{"fontFamily": "Roboto", "sizes": {"h1": {"fontSize": "2.5rem"}}}`	Typography (p.71, p.83, p.87-92, Visual Hierarchy, Readability)
`fontSizes`	Predefined font size tokens (xs, sm, md, lg, xl).	`{"md": "1rem"}`	Typography (p.89, Readability, Visual Hierarchy)
`lineHeights`	Predefined line height tokens (xs, sm, md, lg, xl).	`{"md": 1.55}`	Typography (p.90-92, Readability)
`spacing`	Defines spacing tokens (xs, sm, md, lg, xl) for margins, paddings, gaps.	`{"md": "1rem"}`	Whitespace (p.24-29), Proximity (p.12), Alignment (p.16)
`radius`	Defines border radius tokens (xs, sm, md, lg, xl).	`{"md": "0.5rem"}`	Consistency (p.38, Shape/Corner Radius), Depth & Texture
`defaultRadius`	Default border-radius key from `theme.radius` used by most components.	`"md"`	Consistency (p.38, Shape/Corner Radius)
`shadows`	Defines box-shadow tokens (xs, sm, md, lg, xl).	`{"sm": "0 1px 3px rgba(0,0,0,0.1)"}`	Depth & Texture (p.49-55), Visual Hierarchy

Theme Key	Description	Example Value (Illustrative)	Relevant PDF Principle(s)
`components`	Allows setting global `defaultProps` or `styles` for specific DMC components.	`{"Button": {"defaultProps": {"size": "lg"}}}`	Consistency (p.37-45), Simplicity (p.22), Interaction Cost (by reducing boilerplate)
`other`	A dictionary for any other custom properties the user wants to store in the theme, accessible via `theme["other"]["myCustomValue"]`.	`{"myCustomValue": "any data"}`	Extensibility, Custom Logic
`black`, `white`	Defines the base black and white colors for the theme.	`"#000000"`, `"#FFFFFF"`	Color (p.67, p.82), Contrast
`autoContrast`	If true, automatically adjusts text color for better contrast on colored backgrounds.	`True` or `False`	Contrast (p.21), Accessibility, Readability
`luminanceThreshold`	Threshold used by `autoContrast` to determine if text should be light or dark.	`0.3` (default)	Contrast (p.21), Accessibility
`fontFamilyMonospace`	Font family for monospace elements like `dmc.Code`.	`"ui-monospace, SFMono-Regular,..."`	Typography (p.71, Readability for code)
`breakpoints`	Defines responsive breakpoints (xs, sm, md, lg, xl) in em units.	`{"sm": "48em"}`	Layout (Responsive Design), indirectly related to Clarity and Usability across devices.

This structured approach to theming, particularly through `dmc.MantineProvider` and its `theme` object, empowers the AI to generate Dash applications that are not only visually appealing and brand-aligned but also inherently embody core UI/UX principles, leading to cleaner code and more useful, consistent user experiences.

3. Layout and Structure in Dash Mantine Components

Effective layout is fundamental to creating interfaces that are both aesthetically pleasing and highly usable. It involves the strategic arrangement of elements to guide the user, establish clear hierarchies, and ensure information is presented in an organized and accessible manner. Dash Mantine Components (DMC) provides a versatile set of layout components that, when used in conjunction with design principles, enable the creation of well-structured and responsive applications.

3.1. Achieving Visual Hierarchy

Visual hierarchy dictates how elements are arranged to convey their relative importance, guiding the user's attention and actions. This is achieved by manipulating size, color, weight, and position. It is crucial to prioritize important information and avoid giving undue prominence to less critical elements.

DMC Implementation for Visual Hierarchy:

- **Typography**: The primary tools for establishing text-based hierarchy are `dmc.Title` and `dmc.Text`.
  - `dmc.Title`: Use with the `order` prop (1-6) to define semantic heading levels. The `size` prop (accepting theme keys like 'h1'-'h6', 'xs'-'xl', or numeric values) and `fw` prop (e.g., 400, 700, 'bold') allow for fine-tuning visual prominence.
  - `dmc.Text`: Use with `size`, `fw`, and `c` props for paragraphs and other textual content.
  - **AI Directive**: For main page or section titles, the AI should use `dmc.Title` with a low `order` value (e.g., `order=1` or `order=2`) and a prominent size. Subheadings and less critical text should use `dmc.Title` with higher `order` values or `dmc.Text` with smaller `size` and potentially `c="dimmed"` or a lighter gray shade from `theme.colors` to de-emphasize, ensuring good contrast.
- **Component Prominence**: The choice and styling of components can significantly influence hierarchy.
  - `dmc.Button`: Primary calls-to-action (CTAs) should be visually distinct. Use `variant="filled"` and `color=theme.primaryColor` for primary buttons. Secondary or tertiary actions can use `variant="light"`, `variant="outline"`, or `variant="subtle"` to appear less prominent, as demonstrated by the principle of button hierarchy.
  - `dmc.Card`: The `shadow` prop can be used to elevate important cards, making them appear closer to the user and thus more significant.
- **Color and Contrast**: As discussed in theming, `theme.primaryColor` should be reserved for key interactive elements. Text must always have sufficient contrast with its background to ensure readability and draw attention appropriately.
- **Size and Position**: Larger elements and those placed in prominent positions (e.g., higher on the page, centrally located) tend to attract more user attention. DMC's layout components (`dmc.Grid`, `dmc.Stack`, `dmc.Group`) are instrumental in strategically positioning elements to reinforce the intended visual hierarchy.

3.2. Grid Systems and Responsive Layouts

Grid systems are essential for creating aligned, consistent, and organized layouts. They provide a structural foundation that helps maintain visual order across different parts of an interface and adapt to various screen sizes.

DMC Implementation for Grid Systems:

- **`dmc.Grid` and `dmc.GridCol`**: These components implement a flexible, column-based layout system, typically based on 12 columns by default.
  - `dmc.Grid` props:
    - `columns` (number): Defines the total number of columns in the grid (default is 12).
    - `gutter` (string | number): Sets the spacing between columns. It can accept theme spacing keys (e.g., "xs", "sm", "md", "lg", "xl") or a numerical pixel value.
    - `justify` (string): Controls horizontal alignment of columns within the grid (e.g., 'flex-start', 'center', 'space-between').
    - `align` (string): Controls vertical alignment of columns within the grid (e.g., 'stretch', 'center', 'flex-start').
    - `grow` (boolean): If True, columns in the last row will expand to fill available space.
  - `dmc.GridCol` props:
    - `span` (number | dict): Specifies how many columns the `GridCol` should occupy out of the total defined in `dmc.Grid.columns`. For responsive layouts, this prop can take a dictionary where keys are breakpoint names (e.g., 'base', 'xs', 'sm', 'md', 'lg', 'xl') and values are the column spans for that breakpoint and above. Example: `span={{ "base": 12, "sm": 6, "lg": 4 }}` means the column takes full width on extra-small screens, half width on small screens, and one-third width on large screens.
    - `offset` (number | dict): Skips a specified number of columns before placing the `GridCol`. Also supports responsive dictionary format.
    - `order` (number | dict): Changes the visual order of columns. Also supports responsive dictionary format.
  - **AI Directive**: The AI should use `dmc.Grid` for structuring the main content areas of a page. It must define `gutter` using `theme.spacing` keys (e.g., `gutter="md"`) for consistent spacing. For `dmc.GridCol` components, the AI should extensively use the responsive `span` prop (and `offset`/`order` if needed) to ensure that content reflows gracefully and maintains usability across different screen sizes.
- **`dmc.SimpleGrid`**: This component is used for creating responsive grid layouts where each item automatically takes an equal amount of width within the available space. It's simpler to configure than `dmc.Grid` for such scenarios.
  - Props:
    - `cols` (number | dict): Defines the number of columns. Supports responsive dictionary format (e.g., `cols={{ "base": 1, "sm": 2, "md": 3 }}`) to change the column count based on screen width.
    - `spacing` (string | number | dict): Sets both horizontal and vertical spacing between grid items, using theme spacing keys or numerical values. Supports responsive dictionary format.
    - `verticalSpacing` (string | number | dict): Sets vertical spacing independently, overriding `spacing` for the vertical axis. Supports responsive dictionary format.
  - **AI Directive**: The AI should employ `dmc.SimpleGrid` for layouts requiring equally sized items, such as displaying a series of `dmc.Card` components, feature highlights, or image galleries. The responsive `cols` and `spacing` props are key to its effectiveness.

The combination of responsive props on layout components like `dmc.GridCol` (`span`), `dmc.SimpleGrid` (`cols`), and `dmc.Flex` (`direction`, `gap`) with theme-defined breakpoints (accessible via `theme.breakpoints` but typically handled implicitly by these props) is fundamental for creating truly adaptive UIs. This ensures that layouts are not only well-structured on desktop environments but also

remain usable and readable on smaller screens, directly contributing to the "clean and useful" application requirement.

3.3. Stacking and Grouping Elements

Properly grouping and aligning related elements is essential for creating interfaces that are intuitive and visually orderly. The principle of Proximity suggests that elements placed close together are perceived as related, while Alignment ensures visual harmony.

DMC Implementation for Stacking and Grouping:

- `dmc.Stack`: Arranges its child components in a vertical flex container.
  - Props:
    - `gap` (string | number | dict): Defines the spacing between stacked items. It accepts theme spacing keys (e.g., "sm", "md") or numerical pixel values. Supports responsive dictionary format.
    - `align` (string): Controls the alignment of items along the cross-axis (horizontal for a vertical stack), e.g., 'stretch', 'center', 'flex-start', 'flex-end'.
    - `justify` (string): Controls the alignment of items along the main-axis (vertical), e.g., 'flex-start', 'center', 'space-between'.
  - **AI Directive**: The AI should use `dmc.Stack` for arranging elements vertically, such as in forms (label-input pairs), lists of items, or sections of text followed by controls. It must set the `gap` prop using `theme.spacing` keys (e.g., `gap="sm"`) to maintain consistent vertical rhythm and ensure proper whitespace, as per design principles.
- `dmc.Group`: Arranges its child components in a horizontal flex container.
  - Props:
    - `gap` (string | number | dict): Defines the spacing between grouped items, accepting theme spacing keys or numerical values. Supports responsive dictionary format.
    - `align` (string): Controls vertical alignment of items within the group, e.g., 'center', 'flex-start'. Default is 'center'.
    - `justify` (string): Controls horizontal alignment of items, e.g., 'flex-start', 'center', 'space-between'. Default is 'flex-start'.
    - `grow` (boolean): If True, child elements will attempt to grow to fill available horizontal space.
    - `wrap` (string): Controls how items wrap if they exceed the container width (e.g., 'wrap', 'nowrap'). Default is 'wrap'.
  - **AI Directive**: The AI should utilize `dmc.Group` for arranging elements horizontally, such as button groups, inline form elements, or icon-text pairings. The `gap` prop is crucial for maintaining consistent spacing, adhering to principles of Proximity and Whitespace.
- `dmc.Flex`: This component provides comprehensive control over flexbox layouts, suitable for both horizontal and vertical arrangements when `dmc.Stack` or `dmc.Group` do not offer sufficient flexibility.
  - Props:
    - `direction` (string | dict): Sets the flex direction (e.g., 'row', 'column', 'row-reverse', 'column-reverse'). Supports responsive dictionary format (e.g., `direction={{"base": "column", "sm": "row"}}`).
    - `wrap` (string | dict): Controls flex wrapping (e.g., 'wrap', 'nowrap'). Supports responsive dictionary format.

        - `align` (string | dict): Controls `align-items`. Supports responsive dictionary format.
        - `justify` (string | dict): Controls `justify-content`. Supports responsive dictionary format.
        - `gap` (string | number | dict): Sets the gap between flex items, using theme spacing keys or numerical values. Supports responsive dictionary format.
        - `rowGap`, `columnGap`: For independent row and column spacing.
    - **AI Directive**: For more complex one-dimensional layouts that require nuanced flexbox control or responsive direction changes, the AI should use `dmc.Flex`. It should leverage its responsive props extensively to create adaptive layouts.

Consistent application of `theme.spacing` tokens through the `gutter` prop in `dmc.Grid`, `spacing` in `dmc.SimpleGrid`, and `gap` in `dmc.Stack`, `dmc.Group`, and `dmc.Flex` is paramount. This practice systematically enforces the principles of "Whitespace" and "Proximity". By instructing the AI to use these theme keys (e.g., `gutter="md"`, `gap="lg"`) instead of arbitrary pixel values, the resulting UIs will exhibit better visual rhythm, logical grouping of related elements, and sufficient breathing room, leading to more harmonious and easily scannable interfaces.

3.4. Spacing and Containment

Strategic use of space and effective containment of content are vital for creating balanced, readable, and visually appealing interfaces. Whitespace provides room for elements to "breathe," while containers help manage content width and focus.

DMC Implementation for Spacing and Containment:

- **`dmc.Space`**: This utility component adds explicit horizontal (`w` prop) or vertical (`h` prop) space. The values for `h` and `w` can be theme spacing keys (e.g., "md", "xl") or numerical pixel values.
    - **AI Directive**: The AI should use `dmc.Space(h="md")` or `dmc.Space(w="xl")` to insert deliberate whitespace where margin or padding props on other components are not suitable or sufficient. This aligns with the principle of starting with ample space and refining it.
- **`dmc.Container`**: This component centers its content horizontally within its parent and applies horizontal padding based on the theme. It's crucial for controlling the maximum width of content sections, especially text, to enhance readability.
    - Props:
        - `size` (string | number): Sets the maximum width of the container. It can accept theme breakpoint keys (e.g., 'sm', 'md', 'lg', 'xl') which correspond to predefined max-widths, or a numerical pixel value.
        - `fluid` (boolean): If True, the container will take up 100% of the available width, ignoring the `size` prop.
    - **AI Directive**: The AI should wrap main page content or large text-heavy sections in `dmc.Container`. This helps control the maximum width, preventing overly long lines of text and improving readability, in line with line length recommendations.
- **`dmc.Center`**: A simple utility component that centers its children both horizontally and vertically within its own bounds.
    - Prop: `inline` (boolean): If True, uses `inline-flex` for display, allowing it to be centered within a line of text or alongside other inline elements.
    - **AI Directive**: The AI should use `dmc.Center` for straightforward centering of elements like loading indicators (`dmc.Loader`), single buttons in a section, placeholder text, or icons.

- `dmc.AspectRatio`: This component maintains a consistent width-to-height ratio for its child content, which is particularly useful for media elements.
  - Prop: `ratio` (number): Defines the aspect ratio, calculated as `width / height` (e.g., 16/9 for widescreen video, 1/1 for a square).
  - **AI Directive**: The AI should use `dmc.AspectRatio` when embedding elements like videos (`html.Iframe`), images (`dmc.Image`), or any other content where maintaining a specific aspect ratio is critical for visual integrity.

3.5. Application Shell

A consistent application structure, often involving a main header, navigation area, and content display area, is key to usability and helps users orient themselves within the application.

DMC Implementation for Application Shell:

- `dmc.AppShell`: This component provides a ready-made structure for typical application layouts. It orchestrates common sections like a header, navbar (sidebar), footer, and an optional aside panel, all positioned around a main content area.
  - Children: The `dmc.AppShell` typically contains child components like `dmc.AppShellHeader`, `dmc.AppShellNavbar`, `dmc.AppShellMain` (for the primary content), `dmc.AppShellAside`, and `dmc.AppShellFooter`.
  - Key `dmc.AppShell` props:
    - `padding` (string | number): Controls the padding of the `AppShellMain` section, accepting theme spacing keys or numerical values.
    - `layout` (string): Can be 'default' or 'alt', affecting how Navbar/Aside are arranged relative to Header/Footer.
    - `zIndex` (number | string): Sets the z-index for the shell elements.
    - `withBorder` (boolean): Determines if associated components (Header, Navbar, etc.) should have a border.
    - `header`, `navbar`, `aside`, `footer` (dicts): These props are crucial. Each takes a dictionary to configure the dimensions (`height` for Header/Footer, `width` for Navbar/Aside), breakpoints for responsive behavior (e.g., when a Navbar should collapse or become hidden), and collapsed states.
      - Example: `dmc.AppShell(header={"height": 60}, navbar={"width": 300, "breakpoint": "sm", "collapsed": {"mobile": True, "desktop": False}}, children=[...])`
  - **AI Directive**: For standard application interfaces, the AI should utilize `dmc.AppShell` to establish a consistent and professional frame. It must carefully configure the `header`, `navbar`, `aside` (if used), and `footer` props with appropriate height/width values and, importantly, breakpoint settings to ensure responsive behavior. Navigation links should typically be placed within `dmc.AppShellNavbar`, and the main content of each page should reside within `dmc.AppShellMain`. The configuration of these dictionary-based props is vital for achieving a layout that adapts well to different screen sizes, for example, by collapsing a sidebar into a burger menu on mobile.

Table 3.1: DMC Layout Components and Key Props

Component Name	Key Props	Brief Description & Use Case	Relevant PDF Principle(s)
`dmc.Grid`	`children`, `gutter`, `columns`, `grow`, `justify`, `align`	Flexbox grid system for creating multi-column layouts. Use for main page structure and complex arrangements.	Alignment (p.18), Layout (p.29), Whitespace (via `gutter`)
`dmc.GridCol`	`children`, `span`, `offset`, `order`	Defines a column within a `dmc.Grid`. `span` controls width, supports responsive values.	Alignment, Layout, Visual Hierarchy (positioning)
`dmc.SimpleGrid`	`children`, `cols`, `spacing`, `verticalSpacing`	Responsive grid where each item takes equal width. Ideal for card lists, feature grids. Supports responsive `cols` and `spacing`.	Alignment, Layout, Whitespace, Simplicity
`dmc.Stack`	`children`, `gap`, `align`, `justify`	Arranges children vertically with consistent spacing. Use for forms, vertical lists, content sections.	Proximity (p.12), Alignment (p.16), Whitespace (via `gap`)
`dmc.Group`	`children`, `gap`, `align`, `justify`, `grow`, `wrap`	Arranges children horizontally with consistent spacing. Use for button groups, inline form elements, icon-text pairs.	Proximity (p.12), Alignment (p.16), Whitespace (via `gap`)
`dmc.Flex`	`children`, `direction`, `gap`, `align`, `justify`, `wrap`	Provides full flexbox control for 1D layouts (row or column). More versatile than Stack/Group. Supports responsive props.	Proximity, Alignment, Whitespace, Layout
`dmc.Container`	`children`, `size`, `fluid`	Centers content horizontally and applies theme padding. Controls max-width for readability.	Layout (p.30), Whitespace, Readability (line length)
`dmc.Space`	`h`, `w`	Adds explicit horizontal or vertical whitespace using theme spacing keys or pixel values.	Whitespace (p.24-28)
`dmc.Center`	`children`, `inline`	Centers its children both horizontally and vertically within its bounds.	Alignment, Balance & Harmony (p.35)

Component Name	Key Props	Brief Description & Use Case	Relevant PDF Principle(s)
`dmc.AspectRatio`	`children`, `ratio`	Maintains a consistent width/height ratio for its content (e.g., images, videos).	Layout, Visual Consistency
`dmc.AppShell`	`children`, `header`, `navbar`, `aside`, `footer`, `padding`, `layout`	Creates a standard application shell with header, navbar, main content area, etc. Requires configuration for responsive behavior.	Consistency (p.37), Layout, Usability

By leveraging these layout components and adhering to the associated design principles, the AI can construct Dash applications that are not only visually organized and appealing but also highly functional and adaptable across various devices.

---

4. Effective Data Display with Dash Mantine Components

Presenting data effectively is crucial for user comprehension and decision-making. This involves ensuring clarity, minimizing cognitive load, using visual cues appropriately, and optimizing interaction cost. Dash Mantine Components (DMC) offers a range of components designed for clear and engaging data presentation. Effective data display often arises not just from a single component, but from a thoughtful composition of multiple DMC elements, guided by established UI/UX principles.

4.1. Tabular Data: dmc.Table

Tables are a standard way to present structured datasets. Clarity and proper alignment are key to their effectiveness.

DMC Implementation for Tables:

- **Structure**: A `dmc.Table` component typically wraps a `dmc.TableThead` containing `dmc.TableTr` (table row) and `dmc.TableTh` (table header cell) elements, and a `dmc.TableTbody` containing `dmc.TableTr` and `dmc.TableTd` (table data cell) elements. A `dmc.TableCaption` can also be included.
- Key `dmc.Table` props:
  - `striped` (boolean): Adds alternating row colors (zebra-striping), which significantly improves readability for wide tables.
  - `highlightOnHover` (boolean): Highlights the row under the mouse cursor, providing visual feedback for interaction.
  - `withBorder` (boolean): Adds an outer border to the table.
  - `withColumnBorders` (boolean): Adds vertical borders between columns.
  - `withRowBorders` (boolean): Adds horizontal borders between rows.
  - `captionSide` (string: 'top' or 'bottom'): Specifies the position of the `dmc.TableCaption`.
  - `verticalSpacing`, `horizontalSpacing` (string | number): Control cell padding. These should use theme spacing keys (e.g., "sm", "md") for consistency.
  - `stickyHeader` (boolean): If True, the table header remains visible when scrolling vertically. `stickyHeaderOffset` (number) can define an offset from the top.

- **`dmc.TableScrollContainer`**: For tables that are wider than their container, wrapping the `dmc.Table` with `dmc.TableScrollContainer` enables horizontal scrolling. Key props include `minWidth` (minimum width before scrolling activates) and `maxHeight` (to limit vertical height and enable vertical scrolling within the container).
- **AI Directive**: When displaying datasets, the AI should use `dmc.Table`. To enhance scannability and user interaction, it should set `striped=True` and `highlightOnHover=True`. Consistent cell padding should be maintained using `horizontalSpacing` and `verticalSpacing` with theme spacing keys (e.g., `"sm"`). For tables with many columns that might overflow horizontally, the AI must wrap the `dmc.Table` in a `dmc.TableScrollContainer` and set an appropriate `minWidth`.

4.2. Cards for Segmented Information: dmc.Card

Cards are versatile containers for grouping related information into digestible, visually distinct units. They naturally support principles of Proximity (grouping related items), Simplicity (clean, focused presentation), and Visual Hierarchy (highlighting important segments).

DMC Implementation for Cards:

- **`dmc.Card`**: The base component for creating card layouts.
  - Props:
    - `shadow` (string): Applies a box shadow using theme shadow keys (e.g., 'xs', 'sm', 'md').
    - `padding` (string | number): Controls internal padding, accepting theme spacing keys or numerical values.
    - `radius` (string | number): Sets the border radius, accepting theme radius keys or numerical values.
    - `withBorder` (boolean): Adds a border to the card.
- **`dmc.CardSection`**: A special child component for `dmc.Card` that allows content (like images or footers) to extend to the edges of the card, effectively removing the parent `dmc.Card`'s padding for that section.
  - Props:
    - `withBorder` (boolean): Adds a border to the section, often used to separate it from other card content.
    - `inheritPadding` (boolean): If True, the section will inherit padding from the parent Card.
- **AI Directive**: The AI should use `dmc.Card` to encapsulate related pieces of information, such as user profiles, product summaries, metric displays, or article previews. It should apply a `shadow` (e.g., `shadow="sm"`) to create depth and visual separation. Padding should be controlled via the `padding` prop using theme spacing keys (e.g., `padding="lg"`). `dmc.CardSection` is particularly useful for placing `dmc.Image` components at the top of a card without internal padding, or for creating full-width footer sections with action buttons. To maintain visual consistency for card layouts, global defaults for `dmc.Card` can be set via `theme.components.Card.defaultProps`.

4.3. Images for Visual Context: dmc.Image

Images are powerful tools for conveying information, adding aesthetic appeal, and providing visual context. Effective use of images considers relevance, quality, and performance.

DMC Implementation for Images:

- `dmc.Image`: Component for displaying images with various fitting options and placeholders.
  - Key Props:
    - `src` (string): The path or URL to the image. This is the primary prop.
    - `alt` (string): Alternative text for the image, crucial for accessibility (screen readers) and if the image fails to load.
    - `h` (number | string): Sets the height of the image.
    - `w` (number | string): Sets the width of the image.
    - `fit` (string: 'cover' | 'contain'): Defines how the image should be resized to fit its container. 'cover' scales the image to maintain its aspect ratio while filling the container, potentially cropping some parts. 'contain' scales the image to maintain its aspect ratio while fitting within the container, potentially leaving empty space.
    - `radius` (string | number): Applies a border radius to the image, using theme radius keys or numerical values.
    - `withPlaceholder` (boolean): If True, displays a placeholder (icon and/or text) if the image `src` is not provided or fails to load.
    - `placeholder` (Dash component): Custom content to display as a placeholder.
    - `caption` (string | Dash component): A caption to display below the image.
  - **AI Directive**: The AI must always provide a descriptive `alt` prop for `dmc.Image` to ensure accessibility. When placing images within constrained containers (like `dmc.CardSection` or `dmc.GridCol`), the AI should use the `fit` prop ('cover' or 'contain') appropriately to ensure the image displays well without distortion. If specific dimensions are required, `h` and `w` props should be used. Consider using `radius` to match the `defaultRadius` of other elements for consistency. For images that might not load or are optional, `withPlaceholder=True` should be set to provide a graceful fallback. The `caption` prop can be used for providing context or attribution for the image.

4.4. Lists for Scannable Information: dmc.List

Lists are ideal for presenting a series of related items in a scannable format. Visual cues, such as icons, can significantly enhance their clarity and user-friendliness.

DMC Implementation for Lists:

- **Structure**: A `dmc.List` component contains one or more `dmc.ListItem` components as its children.
- `dmc.List` props:
  - `type` (string: 'ordered' or 'unordered'): Determines if the list is numbered or bulleted. Default is 'unordered'.
  - `withPadding` (boolean): If True, adds left padding to the list items, indenting them.
  - `spacing` (string | number): Controls the vertical spacing between list items, using theme spacing keys or numerical values.
  - `center` (boolean): If True, aligns list items (including their icons) vertically to the center.
  - `icon` (Dash component): A global icon to be used for all list items, replacing the default bullet or number. This is often a `dmc.ThemeIcon` containing a DashIconify icon.
- `dmc.ListItem` props:
  - `icon` (Dash component): An icon specific to this list item, which will override any global icon set on the parent `dmc.List`.

- **AI Directive**: The AI should use `dmc.List` for presenting series of items like features, steps, or navigation links. For unordered lists requiring strong visual cues (e.g., a list of benefits), it should provide an `icon` to the `dmc.List` prop (e.g., `icon=dmc.ThemeIcon(children=DashIconify(icon="tabler:check"), size=24, radius="xl")`). Consistent spacing between items should be maintained using theme.spacing keys.

4.5. Progressive Disclosure: dmc.Accordion and dmc.Spoiler

Progressive disclosure is a technique used to manage complexity by revealing information gradually, thus minimizing initial cognitive load and interaction cost.

DMC Implementation for Progressive Disclosure:

- **`dmc.Accordion`**: Allows content to be divided into collapsible sections.
  - **Structure**: `dmc.Accordion` wraps one or more `dmc.AccordionItem` components. Each `dmc.AccordionItem` contains a `dmc.AccordionControl` (the clickable header) and a `dmc.AccordionPanel` (the collapsible content area).
  - Key `dmc.Accordion` props:
    - `multiple` (boolean): If True, allows multiple accordion items to be open simultaneously.
    - `variant` (string): Controls the visual style (e.g., 'default', 'contained', 'filled', 'separated').
    - `chevronPosition` (string: 'left' or 'right'): Position of the expand/collapse chevron icon in the control.
    - `disableChevronRotation` (boolean): If True, the chevron icon will not rotate upon expanding/collapsing.
  - **AI Directive**: The AI should use `dmc.Accordion` for content like FAQs, detailed settings panels, or lengthy instructional sections where users typically only need to view one piece of information at a time. The choice of `variant` should align with the overall application theme. The `variant` prop is a good example of achieving subtle visual distinction without adding many new colors, supporting simplicity.
- **`dmc.Spoiler`**: Hides long sections of content behind a "show more" / "hide less" toggle.
  - Props:
    - `maxHeight` (number): The maximum height (in pixels) of the content visible before the spoiler control appears.
    - `showLabel` (string | Dash component): Label for the control to expand the content.
    - `hideLabel` (string | Dash component): Label for the control to collapse the content.
    - `transitionDuration` (number): Duration of the reveal animation in milliseconds.
  - **AI Directive**: The AI should employ `dmc.Spoiler` for long text descriptions, comments, or any content that might initially overwhelm users or make a page excessively long. A reasonable `maxHeight` should be set to trigger the spoiler.

4.6. Feedback and Status Indicators

Clear communication of system states, actions, and statuses is vital for a good user experience. This involves using visual cues and adhering to conventional color codes for feedback messages.

DMC Implementation for Feedback and Status:

- **`dmc.Alert`**: Used for displaying static messages to the user, often for feedback on operations.

    - Props: `title` (string), `children` (message content), `color` (string: theme colors like "red", "green", "blue", or CSS color value), `icon` (Dash component, e.g., DashIconify), `variant` (string: 'light', 'filled', 'outline', 'transparent', 'white'), `withCloseButton` (boolean), `duration` (number, for auto-dismissal in milliseconds).
    - **AI Directive**: The AI must use `dmc.Alert` for form validation errors, success messages after an operation, or important informational warnings. It should strictly adhere to color conventions: `color="red"` for errors/danger, `color="green"` for success, `color="yellow"` for warnings, and `color="blue"` or another neutral/brand color for informational alerts. A clear `title` and descriptive `children` message are essential. Including an `icon` (e.g., `icon=DashIconify(icon="tabler:alert-circle")` for errors) enhances the visual cue.
- **`dmc.Badge`**: Small, inline descriptors used to indicate status, categories, or tags.
    - Props: `children` (badge content), `color` (theme color), `variant` (string: 'light', 'filled', 'outline', 'dot', 'gradient'), `size` (string: theme sizes 'xs'-'xl'), `radius` (theme radius keys), `fullWidth` (boolean), `leftSection`, `rightSection` (for icons or other content within the badge).
    - **AI Directive**: The AI should use `dmc.Badge` to denote item statuses (e.g., "New", "Active", "Completed", "Pending") or for categorization tags. The `variant` prop can create visual differentiation.
- **`dmc.Progress`**: A linear progress bar.
    - Props: `value` (number: 0-100), `color` (theme color), `size` (theme size key or number for height), `radius` (theme radius key), `striped` (boolean), `animated` (boolean, for striped variant). Can accept `dmc.ProgressSection` components as children to display multiple segments within a single progress bar.
- **`dmc.RingProgress`**: A circular progress indicator.
    - Props: `sections` (list of dicts, each with `value`, `color`, and optional `tooltip`), `label` (content to display in the center of the ring), `size` (number: diameter), `thickness` (number: ring thickness), `roundCaps` (boolean), `rootColor` (color for the track).
    - **AI Directive**: The AI should use `dmc.Progress` or `dmc.RingProgress` to provide visual feedback on the status of ongoing tasks, display completion percentages, or represent statistical distributions.
- **`dmc.ThemeIcon`**: Displays an icon within a themed circular or rounded-square background.
    - Props: `children` (the icon component, typically DashIconify), `color` (theme color), `variant` (string: 'filled', 'light', 'outline', 'default', 'transparent', 'gradient'), `size` (theme size key or number), `radius` (theme radius key), `gradient` (dict, for gradient variant).
    - **AI Directive**: The AI should use `dmc.ThemeIcon` to create visually distinct and consistently styled icons for lists, alerts, section headers, or navigation items, thereby reinforcing visual cues as advocated. The `icon` prop on components like `dmc.List` or `dmc.Alert` often expects a component like `dmc.ThemeIcon`.

The `icon` prop, present in many components (or achieved via `leftSection`/`rightSection` in components like `dmc.Badge`), is critical for implementing the "Visual Cues" principle. The AI should be guided to use DashIconify (for a wide selection of icons) nested within `dmc.ThemeIcon` or directly in these props to add contextually relevant icons, making the UI more intuitive and scannable.

4.7. Displaying Statistics (KPI Cards)

Key Performance Indicators (KPIs) or other statistics are often displayed in prominent "cards." This requires a combination of visual hierarchy, clarity, and simplicity to make the key figures stand out and be easily understood.

DMC Implementation for KPI Cards: This is a compositional pattern, not a single component.

- **Container**: Use `dmc.Card` as the primary container, typically with `padding="lg"` and a subtle shadow (e.g., `shadow="sm"`).
- **Layout**:
  - Inside the card, `dmc.Stack(gap="xs")` can arrange elements vertically.
  - If an icon is paired with text, or multiple stats are in one card, `dmc.Grid` or `dmc.SimpleGrid` can be used for internal layout.
- **Elements**:
  - **Label/Title**: `dmc.Text("METRIC LABEL", size="sm", c="dimmed")` for the descriptive label of the statistic.
  - **Value**: `dmc.Title("1,234.56", order=2, size="h1")` or a `dmc.Text` component with a large size (e.g., `size="2.5rem"`) and `weight="bold"` for the main statistic value. This directly applies the principle of prioritizing important figures.
  - **Icon (Optional)**: `dmc.ThemeIcon` with a DashIconify icon relevant to the metric can be placed alongside the text using `dmc.Group` or `dmc.Grid`.
  - **Change Indicator (Optional)**: A `dmc.Group` containing a `dmc.ThemeIcon` (e.g., with an up/down arrow icon from DashIconify) and a `dmc.Text` (e.g., `"+5.2%"`, `c="green"` or `cr="red"`) can show trends or changes. `dmc.Badge` can also be used for this.
- **AI Directive**: For generating KPI cards, the AI should follow this compositional pattern:
  1. Start with `dmc.Card(padding="lg", shadow="sm", radius="md")`.
  2. Inside, use `dmc.Stack(gap="xs", align="flex-start")` for vertical arrangement.
  3. Place the metric label: `dmc.Text("Statistic Title", size="sm", c="dimmed", mb=4)`.
  4. Display the main value prominently: `dmc.Title("Value", order=3, fz=30, fw=700)` (using style props for precise control).
  5. Optionally, include a change indicator below the main value: `dmc.Group(gap="xs", mt=4)`.
  6. Ensure sufficient whitespace around elements within the card.

Table 4.1: DMC Data Display Components and Key Props

Component Name	Key Props	Primary UI/UX Goal	Relevant PDF Principle(s)
`dmc.Table`	`children`, `striped`, `highlightOnHover`, `withBorder`, `horizontalSpacing`, `verticalSpacing`, `captionSide`	Clear tabular data presentation, enhanced readability for structured data.	Clarity (p.13), Alignment (p.16), Visual Hierarchy (structure)

Component Name	Key Props	Primary UI/UX Goal	Relevant PDF Principle(s)
`dmc.Card`	`children`, `shadow`, `padding`, `radius`, `withBorder`	Segmented content display, grouping related information into digestible units.	Proximity (p.12), Simplicity (p.22), Visual Hierarchy (p.6), Depth & Texture (p.49)
`dmc.CardSection`	`children`, `withBorder`, `inheritPadding`	Allows content (e.g., images, footers) within a Card to span full-width without parent padding.	Layout (visual structure within card), Consistency
`dmc.Image`	`src`, `alt`, `h`, `w`, `fit`, `radius`, `withPlaceholder`, `placeholder`, `caption`	Displays images with fitting options, placeholders, and captions for visual context and appeal.	Clarity (Visual Information), Accessibility (via `alt`), Visual Consistency (via `radius`)
`dmc.List`	`children`, `type`, `icon`, `spacing`, `center`, `withPadding`	Scannable lists of items, enhanced with optional icons for quick comprehension.	Clarity (p.13), Visual Cues (p.46, via `icon`), Readability
`dmc.ListItem`	`children`, `icon`	Individual item within a `dmc.List`, can have its own specific icon.	Clarity, Visual Cues
`dmc.Accordion`	`children`, `multiple`, `variant`, `chevronPosition`	Progressive disclosure of information, reducing cognitive load by hiding complex details until needed.	Interaction Cost (p.97, p.101), Clarity
`dmc.Spoiler`	`children`, `maxHeight`, `showLabel`, `hideLabel`	Hides long sections of content, revealing it on demand to prevent overwhelming the user.	Interaction Cost, Clarity, Simplicity

Component Name	Key Props	Primary UI/UX Goal	Relevant PDF Principle(s)
`dmc.Alert`	`title`, `children`, `color`, `icon`, `variant`, `withCloseButton`, `duration`	Provides contextual feedback messages (errors, warnings, success, info).	Clarity (p.13), Visual Cues (p.46, via `icon` and `color`), Color Conventions (p.70), Interaction Cost (feedback reduces errors)
`dmc.Badge`	`children`, `color`, `variant`, `size`, `radius`, `leftSection`, `rightSection`	Small, inline indicators for status, tags, or categories.	Visual Cues (p.46), Clarity (status indication)
`dmc.Progress`	`value`, `color`, `size`, `radius`, `striped`, `animated`, `sections`	Linear visual feedback on task completion or quantitative data.	Visual Cues, Clarity (progress indication)
`dmc.RingProgress`	`sections`, `label`, `size`, `thickness`, `rootColor`	Circular visual feedback on task completion or quantitative data, often used for dashboards.	Visual Cues, Clarity (progress indication)
`dmc.ThemeIcon`	`children` (icon), `color`, `variant`, `size`, `radius`, `gradient`	Displays an icon within a themed background, enhancing visual cues and consistency.	Visual Cues (p.46), Consistency (icon styling)

By strategically employing these data display components and adhering to the outlined principles, the AI can create Dash applications that present information clearly, efficiently, and engagingly.

4.x. Visualizing Data: Charts and Graphs with Dash Mantine Components

Dash Mantine Components provides a dedicated suite for creating various types of charts, built on top of Recharts. These components are essential for visualizing data trends, comparisons, and distributions, making complex datasets more understandable and actionable. [53] Effective data visualization adheres to principles of clarity, accuracy, and relevance, ensuring that charts support the user's task without misleading them.

General Principles for Chart Usage:

- **Choose the Right Chart Type**: Select a chart appropriate for the data and the insight you want to convey (e.g., line charts for trends, bar charts for comparisons, pie charts for proportions).

- **Clarity and Simplicity**: Keep charts clean and uncluttered. Avoid unnecessary decorations ("chart junk"). Labels, axes, and legends should be clear and legible.
- **Color Usage**: Use colors purposefully to distinguish data series or highlight key information. Ensure colors are accessible and contrast well. Leverage `theme.colors` for consistency.
- **Interactivity**: Utilize tooltips to provide more details on demand. Consider features like zoom and pan if dealing with dense datasets.
- **Responsiveness**: Ensure charts adapt well to different screen sizes.

DMC Chart Components Overview: All DMC chart components share some common props related to data, series, axes, and styling. They are designed to work seamlessly within the Dash ecosystem.

- `data` **(list of dicts)**: The dataset to be plotted. Each dictionary represents a data point.
- `dataKey` **(string)**: Specifies the key in the `data` dictionaries that corresponds to the independent variable (often the x-axis or categories).
- `series` **(list of dicts)**: Defines the data series to be plotted. Each dictionary in the list typically specifies:
  - `name` (string): The key in the `data` dictionaries for this series' values.
  - `label` (string): A human-readable name for the series (used in tooltips, legends).
  - `color` (string): A theme color key (e.g., "blue.6", "myBrandColor.5") or a CSS color for this series.
- `xAxisProps`, `yAxisProps` **(dict)**: Props to customize the X and Y axes, respectively (e.g., labels, tick formatting).
- `tooltipProps` **(dict)**: Props for customizing the tooltip.
- `legendProps` **(dict)**: Props for customizing the legend.
- `withTooltip` **(boolean)**: Enables/disables the tooltip (default True).
- `withLegend` **(boolean)**: Enables/disables the legend (default True).

4.x.1. Area Chart: `dmc.AreaChart`

Area charts are ideal for showing trends over time or categories, emphasizing the volume or magnitude of change. [54]

- Key `dmc.AreaChart` specific props:
  - `type` (string: 'default', 'stacked', 'percent', 'split'): Determines how areas are drawn. 'stacked' shows cumulative values, 'percent' normalizes to 100%, 'split' handles positive/negative values. [54]
  - `curveType` (string: e.g., 'linear', 'natural', 'monotone'): Controls the line curvature. [54]
  - `strokeWidth` (number): Width of the line at the top of the area. [54]
  - `fillOpacity` (number): Opacity of the area fill. [54]
- **AI Directive**: Use `dmc.AreaChart` for quantitative data that changes continuously. If comparing multiple series, consider `type='stacked'` or `type='percent'` for part-to-whole relationships. Ensure `fillOpacity` is set to a value that allows underlying grid lines or other series to be visible if overlapping. Use `theme.colors` for series colors.

4.x.2. Bar Chart: `dmc.BarChart`

Bar charts are effective for comparing discrete categories or showing variations over time when the number of time points is limited. [55]

- Key `dmc.BarChart` specific props:
  - `layout` (string: 'horizontal' or 'vertical'): Orientation of the bars. Default is 'vertical'. [55]
  - `type` (string: 'default', 'stacked', 'percent'): Determines how bars for multiple series are drawn. [55]
  - `barProps` (dict): Props for customizing the bars (e.g., `radius` for rounded corners). [55]
- **AI Directive**: Use `dmc.BarChart` for comparing values across different groups. `layout='horizontal'` is often better when category labels are long. Use `type='stacked'` or `type='percent'` for part-to-whole comparisons within categories. Apply `theme.spacing` principles if adding custom spacing or padding.

4.x.3. Bubble Chart: `dmc.BubbleChart`

Bubble charts display three dimensions of data: two for position (x, y axes) and one for bubble size. [56]

- Key `dmc.BubbleChart` specific props:
  - Requires a third dimension in the `data` for bubble size, specified via `zAxisProps={{ dataKey: 'size_dimension' }}`. [56]
- **AI Directive**: Use `dmc.BubbleChart` when you need to visualize relationships between three variables, where one variable influences the size of the plotted points. Ensure clear legends for all three dimensions.

4.x.4. Donut Chart: `dmc.DonutChart`

Donut charts (and Pie Charts) display proportions of a whole. The central hole in a donut chart can be used to display a total value or a label. [58]

- Key `dmc.DonutChart` specific props:
  - `data` (list of dicts): Each dict needs a `name` (category label) and `value` (its proportion), and `color`. [58]
  - `chartLabel` (string | Dash component): Content for the center of the donut. [58]
  - `startAngle`, `endAngle` (number): To create semi-donut charts. [58]
  - `paddingAngle` (number): Space between donut segments. [58]
- **AI Directive**: Use `dmc.DonutChart` for showing parts of a whole, ideally with a small number of categories (e.g., 2-6). Ensure segment colors are distinct and use `chartLabel` for summary information. Adhere to principles of using color meaningfully.

4.x.5. Line Chart: `dmc.LineChart`

Line charts are best for showing trends and changes in data over continuous intervals or time. [59]

- Key `dmc.LineChart` specific props:
  - `curveType` (string: e.g., 'linear', 'natural', 'step'): Controls line curvature. [59]
  - `connectNulls` (boolean): If True, lines will connect across null data points. [59]
  - `strokeWidth` (number): Common prop from series for line thickness.
- **AI Directive**: Use `dmc.LineChart` for time-series data or continuous data. Use distinct `theme.colors` for multiple lines and ensure the legend is clear. `curveType='natural'` can provide a smoother look, but 'linear' is often more accurate for direct data representation.

4.x.6. Pie Chart: `dmc.PieChart`

Similar to Donut Charts, Pie Charts show proportions. They are classic but should be used cautiously, as comparing angles can be harder than comparing lengths (as in bar charts). [60]

- Key `dmc.PieChart` specific props:
  - `data` (list of dicts): Each dict needs `name`, `value`, and `color`. [60]
  - `withLabels` (boolean): Display labels directly on pie segments. [60]
  - `labelsPosition`, `labelsType` (string): Control label appearance. [60]
- **AI Directive**: Prefer `dmc.BarChart` or `dmc.DonutChart` if possible, especially if precise comparison is needed or if there are many categories. If using `dmc.PieChart`, limit the number of slices (ideally <7) and use `withLabels=True` for clarity.

4.x.7. Radar Chart: `dmc.RadarChart`

Radar charts (or spider charts) are used to display multivariate data in the form of a two-dimensional chart of three or more quantitative variables represented on axes starting from the same point. [61]

- Key `dmc.RadarChart` specific props:
  - `dataKey` (string): Key for the categorical variable forming the axes.
  - `series` (list of dicts): Each series represents a different entity being compared across these categories.
- **AI Directive**: Use `dmc.RadarChart` to compare multiple entities across several common variables. Ensure axes are clearly labeled. Too many variables or series can make it hard to read.

4.x.8. Scatter Chart: `dmc.ScatterChart`

Scatter charts are used to plot data points on a horizontal and vertical axis to show how much one variable is affected by another. [62]

- Key `dmc.ScatterChart` specific props:
  - `series` (list of dicts): Each dict needs `name` (data key for y-axis values) and `label`. The x-axis data key is usually set in `xAxisProps`. [62]
- **AI Directive**: Use `dmc.ScatterChart` to identify correlations or patterns between two continuous variables. Tooltips are crucial here to identify individual data points.

4.x.9. Sparkline Chart: `dmc.Sparkline`

Sparklines are small, simple, data-dense charts typically embedded in text, tables, or alongside KPIs to provide a quick visual representation of trends without axes or labels. [63]

- Key `dmc.Sparkline` props:
  - `data` (list of numbers): The data points for the sparkline. [63]
  - `width`, `height` (number): Dimensions of the sparkline. [63]
  - `curveType` (string): Line curvature. [63]
  - `color` (string): Color of the sparkline. [63]
  - `fill` (boolean): Whether to fill the area below the line. [63]
- **AI Directive**: Use `dmc.Sparkline` within `dmc.Card` (e.g., for KPI trends), `dmc.Table` cells, or alongside `dmc.Text` to give a quick, compact visual summary of a trend. Keep them minimal and ensure they are large enough to be discernible.

4.x.10. Composite Chart: `dmc.CompositeChart`

Composite charts (referred to as "MixedChart" or similar in other libraries, though DMC names it `CompositeChart`) allow combining different chart types (e.g., bar and line) within the same plot area, sharing axes. This is useful for showing data with different scales or types of relationships simultaneously. [57]

- The `dmc.CompositeChart` typically involves defining different types of series (e.g., some as bars, some as lines) within its structure, using components like `<Bar />`, `<Line />` from Recharts contextually if such detailed composition is exposed, or more likely, by defining series types within the `series` prop structure if the DMC abstraction supports it. The documentation for `dmc.CompositeChart` would specify how to mix series types.
- **AI Directive**: Use `dmc.CompositeChart` when you need to overlay different types of data representations, for example, showing sales volume as bars and profit margin as a line on the same chart. Ensure clarity by using distinct colors and clear legends for each series and chart type. Be mindful of not overcrowding the chart.

AI Directive for All Charts: The AI should always ensure that chart data (`data`, `dataKey`, `series`) is correctly structured. For styling, it must prioritize using `theme.colors` for series colors, `theme.fontFamily` for labels and text through chart props, and ensure sufficient contrast and readability for all textual elements in charts (axis labels, legend, tooltips), aligning with general typography and accessibility principles. Whenever a chart is presented, it should have a descriptive `dmc.Title` or `dmc.Text` nearby explaining what the chart shows, unless it's a sparkline intended for very compact representation.

---

5. Typography in Dash Mantine Components

Typography is a foundational element of user interface design, playing a crucial role in communication, readability, visual hierarchy, and overall user experience. Dash Mantine Components (DMC) provides a suite of typography components and extensive theme customization options to implement effective typographic systems. This section details how to use specific DMC typography components and their props, guided by the principles outlined in "The UI_UX Playbook".

Core Typography Principles (Recap):

- **Font Choice & Readability**: Select fonts appropriate for the context, prioritizing readability and aesthetic appeal. Limit to 1-2 font families.
- **Hierarchy**: Use variations in size, weight, and color to establish a clear visual hierarchy, guiding the user's attention to important information.
- **Line Length & Height**: Optimize line length (45-75 characters for desktop) and line height (150-160% of font size for body text) for comfortable reading.
- **Contrast**: Ensure sufficient contrast between text and its background (WCAG AA 4.5:1). Avoid pure black/white text on contrasting backgrounds; opt for softer grays.
- **Alignment**: Left-align text for long passages to enhance readability in left-to-right languages.

DMC Typography Components:

- **`dmc.Title`**: Used for rendering semantic HTML headings (`<h1>`-`<h6>`) with styles derived from the Mantine theme.
    - Key Props:

        - `children`: The text content of the title.
        - `order` (number: 1-6): Determines the HTML heading tag (e.g., `order=1` renders `<h1>`). This also influences the default font size if the `size` prop is not set.
        - `size` (string | number): Overrides the default font size. Can be a theme heading key (e.g., "h1", "h3" to match `theme.headings.sizes`), a theme font size key (e.g., "xl" from `theme.fontSizes`), or a numerical pixel/rem value.
        - `fw` (string | number): Sets the font weight (e.g., 100, 400, 700, 'bold', 'normal').
        - `c` (string): Specifies the text color. Accepts theme color keys (e.g., "blue", "myCustomColor.7", "dimmed") or any valid CSS color string.
        - `ta` (string: 'left' | 'center' | 'right' | 'justify'): Controls text alignment.
        - `lineClamp` (number): Truncates the text after a specified number of lines, adding an ellipsis.
        - `transform` (string): CSS text-transform property (e.g., 'capitalize', 'uppercase', 'lowercase').
        - `variant` (string): Can be used for special styling, e.g., 'gradient' if a gradient is defined in `theme.defaultGradient` or via style props.
        - Style props (shortcuts for `style` prop): `fz` (fontSize), `fw` (fontWeight), `c` (color), `ff` (fontFamily), `lh` (lineHeight), `ta` (textAlign).
    - **AI Directive**: The AI must use `dmc.Title` for all semantic page and section headings. The `order` prop should be set appropriately to maintain a logical document structure. `size` and `fw` props are crucial for establishing visual hierarchy. For text color, consider using shades from `theme.colors` (e.g., `c="dark.6"` or a specific brand color) rather than pure black to adhere to softer contrast principles. `align` should be used for specific layout needs, defaulting to left for readability.
- **`dmc.Text`**: The primary component for rendering paragraphs and general text content.
    - Key Props:
        - `children`: The text content.
        - `size` (string | number): Sets the font size. Accepts theme font size keys (e.g., 'xs', 'sm', 'md', 'lg', 'xl') or a numerical pixel/rem value. Defaults to 'md'.
        - `fw` (string | number): Sets the font weight.
        - `c` (string): Specifies the text color. Accepts theme color keys (including the special value "dimmed" which maps to a subdued gray) or any valid CSS color string.
        - `ta` (string: 'left' | 'center' | 'right' | 'justify'): Controls text alignment.
        - `span` (boolean): If True, renders the text as an inline `<span>` element instead of the default block-level `<p>` element.
        - `inherit` (boolean): If True, the component will inherit font properties (like font-family, size, color) from its parent element, rather than applying its own theme-based styles.
        - `gradient` (dict): Defines a gradient for the text color. Requires `variant="gradient"` to be set. The dictionary should specify `from`, `to`, and `deg` (e.g., `{"from": "indigo", "to": "cyan", "deg": 45}`).
        - `variant` (string): Can be 'text' (default) or 'gradient'.
        - `lineClamp` (number): Truncates the text after a specified number of lines with an ellipsis.
        - `truncate` (boolean | string: 'start' | 'end'): If True or 'end', truncates overflowing text with an ellipsis at the end. If 'start', truncates at the beginning (less common).

        - `inline` (boolean): Sets line-height to 1 for better centering when used as an inline element.
        - Style props: `fz`, `fw`, `c`, `ff`, `lh`, `ta`.
    - **AI Directive**: The AI should use `dmc.Text` for all body copy, labels (not associated with inputs), and other non-heading text. The default `size="md"` should correspond to the 16px base body size defined in the theme. For de-emphasized text or secondary information, `c="dimmed"` or a lighter gray (e.g., `c="gray.7"`) should be used, always ensuring sufficient contrast with the background. The `lineClamp` or `truncate` props are valuable for managing text overflow in constrained spaces like cards or table cells. The AI must ensure that the effective line height (either through the `lh` style prop or global theme settings) is approximately 1.5 to 1.6 for body text to maximize readability. The `inherit` prop is useful when `dmc.Text` needs to blend seamlessly with parent elements that have specific typographic styling.
- **`dmc.Blockquote`**: Used for displaying quoted text, often with attribution and an icon.
    - Key Props:
        - `children`: The main content of the quote.
        - `color` (string): Theme color key used for the left border and the icon's color (if an icon is provided and not colored itself).
        - `icon` (Dash component): An optional icon to display next to the blockquote, typically a DashIconify component.
        - `cite` (string | Dash component): Attribution text for the quote, usually displayed below it.
    - **AI Directive**: The AI should use `dmc.Blockquote` for visually distinguishing quoted material, such as testimonials or excerpts. It should provide an `icon` (e.g., a quotation mark icon) for better visual cueing and use the `cite` prop for any attributions. The `color` prop can be set to the theme's `primaryColor` or another accent color.
- **`dmc.Code`**: For displaying inline or block-level code snippets.
    - Key Props:
        - `children` (string): The code string to be displayed.
        - `color` (string): Theme color key for the background (if `block=True`) or text color.
        - `block` (boolean): If True, renders as a block-level element suitable for multi-line code snippets, often with a distinct background. If False (default), renders as an inline element.
    - This component uses `theme.fontFamilyMonospace` by default.
    - **AI Directive**: The AI should use `dmc.Code` for any representation of computer code. For inline code mentions, `block=False` is appropriate. For multi-line code examples, `block=True` should be used, which typically applies a background color and padding for better visual separation.
- **`dmc.Highlight`**: Used to highlight occurrences of specific substrings within a larger text string.
    - Key Props:
        - `children` (string): The full string in which to search for highlights.
        - `highlight` (string | list of strings): The substring(s) to be highlighted.
        - `highlightColor` (string): Theme color key or CSS color for the highlighted text segments.
        - `color` (string): Optional theme color key or CSS color for the non-highlighted parts of the text.

    - **AI Directive**: The AI should employ `dmc.Highlight` in contexts such as search results (highlighting the search term within results) or when drawing attention to specific keywords in a passage of text.
- **`dmc.RichTextEditor`**: A component for displaying (and potentially editing) rich text content formatted with HTML-like features (WYSIWYG - What You See Is What You Get).
    - Key Props for display:
        - `value` (string): The HTML string representation of the content to be displayed.
        - `readOnly` (boolean): When True, the editor is in display-only mode.
        - `withTypographyStyles` (boolean): If True (default), applies Mantine's typography styles (font family, sizes for headings, lists, etc.) to the rendered content, ensuring it aligns with the application's theme.
    - The editor supports rendering various elements like headings (h1-h6), lists (ordered/unordered), bold, italic, underline, blockquotes, code blocks, etc.
    - **AI Directive**: If the application needs to display complex formatted text that might originate from a database, a CMS, or user input (e.g., blog posts, detailed product descriptions with formatting), the AI should use `dmc.RichTextEditor(value=html_content, readOnly=True, withTypographyStyles=True)`. The `withTypographyStyles=True` prop is crucial for ensuring the displayed rich text adheres to the overall typographic theme of the Dash application.

The combination of global theme settings for typography (e.g., `theme.fontFamily`, `theme.headings`, `theme.fontSizes`, `theme.lineHeights`) and the per-instance styling capabilities offered by props on individual typography components (especially the style props like `fz`, `fw`, `c`, `lh`, `ta`) provides a powerful and flexible system. The theme establishes the baseline consistency, ensuring a cohesive look and feel across the application. Style props then allow the AI to make specific adjustments for emphasis, hierarchy, or unique layout requirements, directly translating the nuanced typographic principles from "The UI_UX Playbook" into the generated Dash interface. Furthermore, utility props like `lineClamp` and `truncate` on `dmc.Text` and `dmc.Title` are practical tools for managing text overflow, which contributes to "Clarity" and "Simplicity" by preventing UIs from becoming cluttered with excessively long text blocks. The `inherit` prop on `dmc.Text` is also noteworthy for scenarios where text needs to seamlessly blend with non-standard parent elements, preserving typographic consistency.

Table 5.1: DMC Typography Components and Key Styling Props

Component Name	Key Props for Styling	Primary Purpose	Relevant PDF Principle(s)
`dmc.Title`	`order`, `size`, `fw`, `c`, `ta`, `lineClamp`, `transform`, `variant`, style props (`fz`, `fw`, `c`, etc.)	Semantic headings (h1-h6). Establishes content structure and visual hierarchy.	Typography (Hierarchy, Size, Weight, Color, Alignment, Readability)

Component Name	Key Props for Styling	Primary Purpose	Relevant PDF Principle(s)
`dmc.Text`	`size`, `fw`, `c`, `ta`, `transform`, `variant`, `gradient`, `span`, `inherit`, `lineClamp`, `truncate`	Paragraphs, labels, and general text content.	Typography (Readability, Size, Weight, Color, Alignment, Line Length, Line Height)
`dmc.Blockquote`	`children`, `color`, `icon`, `cite`	Displaying quoted text with optional visual emphasis (border, icon) and attribution.	Typography (Visual Distinction for Quotes), Visual Cues
`dmc.Code`	`children`, `color`, `block`	Displaying inline or block-level code snippets with monospace font.	Typography (Readability for Code), Clarity
`dmc.Highlight`	`children`, `highlight`, `highlightColor`, `color`	Highlighting specific substrings within a text for emphasis or search results.	Visual Cues, Clarity (Drawing Attention)
`dmc.RichTextEditor`	`value` (HTML string), `readOnly`, `withTypographyStyles`	Displaying complex, pre-formatted rich text content (headings, lists, bold, etc.) while adhering to theme typography.	Typography (Consistent display of rich content), Clarity

By correctly utilizing these components and their props, guided by the theme and typographic principles, the AI can generate Dash applications with text that is not only legible and readable but also effectively structured and visually appealing.

---

6. Advanced UI/UX Considerations for AI Implementation

Beyond the foundational aspects of theming, layout, data display, and typography, several advanced UI/UX considerations, drawn from "The UI_UX Playbook", can further elevate the quality of AI-generated Dash applications using DMC. Many of these involve not necessarily new components, but specific patterns of composing existing DMC elements and applying principles in nuanced contexts.

6.1. Consistency

Consistency in design is paramount for creating a predictable, trustworthy, and easy-to-use interface. It reduces cognitive load by allowing users to learn patterns and apply them across the application.

- **Recap of DMC Mechanisms**: Achieved through global theme settings (`theme.primaryColor`, `theme.fontFamily`, `theme.defaultRadius`, `theme.spacing`, `theme.shadows`) and especially

`theme.components` for default props and styles. Consistent use of component props (e.g., always using `size="sm"` for secondary buttons) also contributes.

- **Specific PDF Principles and DMC Application for AI**:
  - **Shape and Corner Radius Consistency**:
    - **AI Directive**: The AI must use `theme.defaultRadius` as the base for most components. When specific radius values are needed (e.g., for `dmc.Button`, `dmc.Card`, `dmc.Input`, `dmc.Paper`), it should use predefined tokens from `theme.radius` (e.g., `radius="sm"`, `radius="lg"`) rather than arbitrary values to maintain uniformity.
  - **Button Consistency**:
    - **AI Directive**: The AI should define default button styles in `theme.components.Button.defaultProps` (e.g., a common size or variant). For different semantic actions (primary, secondary, tertiary), it must consistently use specific `variant` and `color` combinations. For example:
      - Primary actions: `dmc.Button(children="Submit", variant="filled", color=theme.primaryColor)`
      - Secondary actions: `dmc.Button(children="Cancel", variant="light", color="gray")` or `dmc.Button(children="View Details", variant="outline")`
  - **Card Length Consistency**:
    - **AI Directive**: When generating lists of `dmc.Card` components within a `dmc.Grid` or `dmc.SimpleGrid`, the AI should strive to make the content within each card (especially titles and descriptions) of similar length to maintain visual balance and alignment. If content length varies significantly, ensure that key elements like call-to-action buttons within the cards are aligned (e.g., all at the bottom of the card, potentially by using `dmc.Stack` with `justify="space-between"` inside the card and ensuring cards have a fixed height or use flex properties in their container).
  - **Icon Consistency**:
    - **AI Directive**: The AI should primarily use DashIconify for icons, often wrapped in `dmc.ThemeIcon` for consistent styling (background, size, radius). It must maintain a consistent icon style throughout the application (e.g., all outlined icons or all filled icons from the same icon set like Tabler Icons). An exception is using filled vs. outlined icons to indicate state (e.g., selected vs. unselected tab icon), as noted in the principles. A default size for icons in common contexts (e.g., within buttons or list items) should be considered.

6.2. Interaction Cost

Minimizing interaction cost—the mental, physical, and time effort users must expend to achieve their goals—is crucial for a positive user experience.

- **Specific PDF Principles and DMC Application for AI**:
  - **Keep Related Actions Close (Fitts's Law)**:
    - **AI Directive**: The AI should place interactive elements (e.g., `dmc.Button`, `dmc.ActionIcon`) that perform actions on a specific item (like "Edit" or "Delete" for a table row, or "Add to Cart" for a product) physically close to that item. In forms, the primary submission `dmc.Button` should be located directly after the last input field or in a consistently accessible area (e.g., bottom right of a form card).

- **Minimize Choice (Hick's Law)**:
  - **AI Directive**: When presenting users with a set of options:
    - If there are few options (typically 2-5) and they are mutually exclusive, the AI should prefer `dmc.SegmentedControl` or `dmc.RadioGroup` over `dmc.Select`. These components make all choices immediately visible, reducing clicks and cognitive load.
    - For color selection, if space permits and options are limited, displaying `dmc.ColorSwatch` components directly is preferable to hiding them in a dropdown.
- **Reduce Distractions**:
  - **AI Directive**: During critical user tasks like form submissions (e.g., sign-up, checkout) or complex data entry, the AI must avoid placing distracting elements nearby. This includes non-essential `dmc.Alerts`, overly complex or animated `dmc.Cards`, or irrelevant imagery. The layout should be focused, using components like `dmc.Stack` or `dmc.Container` with ample whitespace to draw attention to the task at hand.
- **Streamline Tasks**:
  - **AI Directive**: For multi-step processes, the AI should use `dmc.Stepper` to guide the user. For data input, it must select the most appropriate input component for the data type (e.g., `dmc.TextInput` for text, `dmc.NumberInput` for numbers, `dmc.DatePicker` for dates, `dmc.PasswordInput` for passwords). Input fields should be designed for the type of input expected, for example, for verification codes, using multiple small, connected `dmc.TextInput` fields or a dedicated `dmc.PinInput` component can reduce errors and improve input speed.

The choice of interactive DMC component directly impacts interaction cost. For example, using `dmc.RadioGroup` for 3-4 mutually exclusive options requires only one click, whereas a `dmc.Select` component would require at least two (click to open, click to select). Guiding the AI to select the most efficient component based on the number of options, data type, and available screen real estate is key to minimizing user effort.

6.3. Accessibility

Designing for accessibility ensures that applications are usable by people with a wide range of abilities, including those with visual, motor, or cognitive impairments. Key aspects include contrast, keyboard navigation, and semantic HTML.

- **DMC Application for AI**:
  - **Contrast**:
    - **AI Directive**: When the AI selects a color for `dmc.Text`, `dmc.Title`, or background colors for components like `dmc.Paper` or `dmc.Card`, it must ensure that the contrast ratio between the text and its background is at least 4.5:1 (WCAG AA). This is critical for readability. The theme's `autoContrast` feature can help, or external tools can be used for verification.
  - **Interactive Elements**:
    - **AI Directive**: The AI must ensure that all interactive elements (`dmc.Button`, `dmc.ActionIcon`, `dmc.TextInput`, links, etc.) have clear and visible focus states. Mantine components generally handle this well through the theme's `focusRing` settings. For components like `dmc.ActionIcon` that may not have visible text labels, an

`aria-label` prop (if available, or by wrapping with an element that supports it) should be included to provide context for screen reader users.

- **Semantic HTML**:
  - **AI Directive**: The AI should use `dmc.Title` with the correct `order` prop (1-6) to create a proper semantic heading structure for the page. It should use `dmc.List` for actual lists of items rather than simulating lists with other components. Standard HTML tags, when used via `html.Div`, `html.P`, etc., should also be semantically appropriate.

6.4. Visual Cues and Feedback

Visual cues help users understand information more quickly and make interfaces more intuitive. Feedback mechanisms inform users about the results of their actions or the system's status.

- **DMC Implementation for AI**:
  - **`dmc.Tooltip`**:
    - Props: `label` (string | Dash component for the tooltip content), `position` (string, e.g., 'top', 'bottom', 'left', 'right'), `withArrow` (boolean).
    - **AI Directive**: The AI should wrap components like `dmc.ActionIcon` (especially if they only contain an icon without text) or truncated `dmc.Text` elements with `dmc.Tooltip` to provide additional information or context on hover.
  - **`dmc.Notification`**: For displaying non-modal, often timed, feedback messages (toasts).
    - Requires `dmc.NotificationProvider()` to be included in the application layout (typically once, within `dmc.MantineProvider`). Notifications are then triggered and updated via callbacks by returning `dmc.Notification` components as children of a designated output container.
    - `dmc.Notification` props: `id` (string, for updating/hiding), `action` (string: 'show', 'update', 'hide'), `title` (string), `message` (string | Dash component), `icon` (DashIconify), `color` (theme color), `loading` (boolean), `disallowClose` (boolean), `autoClose` (number in ms, or False).
    - **AI Directive**: The AI should use `dmc.Notification` for providing feedback on asynchronous operations (e.g., "Data saved successfully," "Error submitting form"). It should use appropriate `color` and `icon` props to convey the message type (success, error, info, warning).
  - **`dmc.Loader`**: For indicating that a process is ongoing and the user should wait.
    - Props: `color` (theme color), `size` (string | number), `variant` (string: 'oval', 'bars', 'dots').
    - Can be used standalone or integrated into `dmc.Button` via its `loading=True` prop, which displays a loader within the button.
    - **AI Directive**: The AI should display `dmc.Loader` (or set `loading=True` on relevant buttons) during callback execution or any period where the application is fetching or processing data, to provide clear feedback that the system is working.

6.5. Empty States and Error Handling

Well-designed empty states and error pages are crucial for guiding users and preventing frustration when content is unavailable or something goes wrong.

- **DMC Application for AI (Compositional Patterns)**:

- **Empty States**: When a list, table, or data view has no content to display.
  - **AI Directive**: The AI should construct an empty state message typically using `dmc.Stack(align="center", gap="md", p="xl")`. This stack might contain:
    1. An optional `dmc.ThemeIcon` with a relevant icon (e.g., `DashIconify(icon="tabler:mood-empty")`, `size=50`).
    2. A `dmc.Title("No items to display", order=3, align="center")`.
    3. A `dmc.Text("Get started by creating a new item or adjusting your filters.", color="dimmed", align="center", maw=400)`.
    4. Optionally, a `dmc.Button("Create New Item", ...)` to guide the user to the next logical action.
- **Error Handling (e.g., 404 Page Not Found)**:
  - **AI Directive**: For a 404 page, the AI should use a clear and helpful layout, for instance, `dmc.Stack(align="center", justify="center", style={"height": "70vh"}, gap="lg")`:
    1. `dmc.Title("404 - Page Not Found", order=1, align="center")`.
    2. `dmc.Text("Oops! The page you are looking for might have been removed, had its name changed, or is temporarily unavailable.", color="dimmed", align="center", maw=500)`.
    3. `dmc.Button("Go to Homepage", component="a", href="/", variant="outline")`.

Table 6.1: Mapping Advanced PDF Principles to DMC Patterns

PDF Principle	Key DMC Components/Props Involved	AI Directive/Pattern Example
Consistent Buttons	`dmc.Button`, `theme.components.Button.defaultProps`, `variant`, `color`, `size`, `radius`	Define default button style in theme. Use consistent variant/color for primary (`variant="filled"`, `color=primaryColor`), secondary (`variant="light"`), and tertiary (`variant="subtle"`) actions.
Minimize Choice	`dmc.RadioGroup`, `dmc.SegmentedControl`, `dmc.Select`, `dmc.ColorSwatch`	For <5 mutually exclusive options, prefer `dmc.RadioGroup` or `dmc.SegmentedControl` over `dmc.Select`. Display `dmc.ColorSwatch` directly for color choices if space allows.
Empty State Design	`dmc.Stack`, `dmc.ThemeIcon`, DashIconify, `dmc.Title`, `dmc.Text`, `dmc.Button`	Use a centered `dmc.Stack` with an icon, a clear title (e.g., "No Data"), informative text, and a CTA button (e.g., "Add New Record").

PDF Principle	Key DMC Components/Props Involved	AI Directive/Pattern Example
Clear Error States	`dmc.Stack`, `dmc.Title`, `dmc.Text`, `dmc.Button` (for 404); `dmc.Alert` (for form errors)	For 404: Centered message with title, explanation, and link to homepage. For form errors: `dmc.Alert` with `color="red"`, descriptive message, and icon.
Visual Cues (Icons)	`dmc.ThemeIcon`, DashIconify, `dmc.Alert(icon=...)`, `dmc.List(icon=...)`, `dmc.Button(leftSection=...)`	Consistently use icons (via DashIconify within `dmc.ThemeIcon` or directly in props like `icon` or `leftSection`) to enhance understanding and scannability in lists, alerts, buttons, and navigation. Maintain a consistent icon style.
Reduce Distractions	`dmc.Container`, `dmc.Stack`, `dmc.Paper`, careful use of `dmc.Alert`, `dmc.Card`	During focused tasks (e.g., checkout, sign-up), use simple layouts with ample whitespace. Avoid non-essential visual elements or animations. `dmc.Container` can limit width to improve focus.

By applying these advanced considerations, the AI can generate Dash applications that are not only functional and aesthetically pleasing at a surface level but also deeply user-centric, offering intuitive interactions, clear feedback, and a consistently high-quality experience.

---

7. Conclusion

This guide has systematically translated UI/UX best practices, primarily from "The UI_UX Playbook", into actionable directives for an AI tasked with creating clean, useful, and aesthetically pleasing Dash applications using dash-mantine-components (DMC) for dash==3.0.4. The core focus has been on leveraging DMC's theming capabilities, layout components, data display elements, typography system, and charting components to achieve these goals.

Key Takeaways for AI-Driven Development with DMC:

1. **Thematic Foundation is Crucial**: The `dmc.MantineProvider` and its `theme` object are paramount. A well-structured theme, defining `primaryColor`, a comprehensive `colors` palette (10 shades per color), `fontFamily`, `headings` styles, `spacing` scales, `defaultRadius`, and `shadows`, forms the bedrock of a consistent and visually appealing application. The AI should prioritize establishing a robust theme as its first step. Global component styling via `theme.components` further enhances consistency and reduces code redundancy.
2. **Layout for Clarity and Adaptability**: DMC's layout components (`Grid`, `SimpleGrid`, `Stack`, `Group`, `Flex`, `Container`, `AppShell`) provide the tools for creating well-organized and responsive interfaces. The AI must utilize responsive props (e.g., `span` on `dmc.GridCol`, `cols` on

`dmc.SimpleGrid`) and theme spacing tokens (`gap`, `gutter`) consistently to ensure layouts adapt to different screen sizes and maintain visual rhythm.

3. **Purposeful Data Display and Visualization**: Selecting the appropriate DMC component for data presentation (`Table`, `Card`, `List`, `Image`, `Accordion`, `Alert`, `Badge`, `Progress`) and visualization (`AreaChart`, `BarChart`, `LineChart`, etc.) and composing them effectively is key. Visual hierarchy within data displays (e.g., KPI cards) should highlight important information, and feedback elements must use clear visual cues and conventional coloring. Charts should be chosen appropriately for the data type and insight, maintaining clarity and leveraging theme colors.
4. **Typography for Readability and Hierarchy**: The AI must leverage both global theme settings (`theme.fontFamily`, `theme.headings`, `theme.fontSizes`, `theme.lineHeights`) and specific props on typography components (`dmc.Title`, `dmc.Text`) to ensure text is legible, readable, and hierarchically structured. Adherence to principles of font size, weight, color contrast, and line height is critical.
5. **User-Centric Interactions**: Principles like minimizing interaction cost, providing clear feedback, ensuring accessibility, and designing helpful empty/error states should guide the AI's component choices and compositional patterns. This often involves combining simpler DMC elements into meaningful UX patterns.

By internalizing these guidelines and applying them diligently, the AI can significantly enhance the quality of the Dash applications it generates. The emphasis on specific DMC components and props, tied back to established UI/UX principles, provides a robust framework for creating interfaces that are not only functional but also genuinely user-friendly and visually engaging. This structured approach will lead to Dash applications that are easier to use, more intuitive, and ultimately more successful in meeting user needs.

Dash Mantine Components v1.3.0: Enhanced Guide

This document provides comprehensive guidance and documentation for dash-mantine-components version 1.3.0. It aims to be a practical resource for developers, covering component usage, properties, and best practices. This guide expands upon previous versions by incorporating new components, updating information on existing ones, and clarifying common points of confusion.

The primary objectives of this updated guide are:

- To ensure all information reflects the features and components available in dash-mantine-components==1.3.0.
- To clarify specific component functionalities, such as the absence of an href prop in dmc.Button and the introduction of dmc.Anchor for hyperlink functionality.
- To systematically add documentation for components new or significantly updated in version 1.3.0, as well as other essential components, following a consistent and developer-friendly structure.

Accurate and thorough documentation is paramount for efficient development. This guide endeavors to meet that need by providing clear explanations, practical examples, and detailed API references.

dmc.Button Component Clarifications

href Property in dmc.Button

A common query among developers pertains to adding hyperlink functionality directly to a dmc.Button. It is important to note that **the dmc.Button component in dash-mantine-components does not possess an href property for direct URL navigation.** Buttons are primarily designed for actions within the application, such as triggering callbacks, submitting forms, or controlling UI states.

While dmc.Button offers extensive styling options, including variants like filled, outline, light, subtle, and gradient , and can include icons or be grouped , it is not intended to function as a navigational link.

For claude.md maintainers: It is recommended to add the following note directly within the dmc.Button component's documentation section:

Note on Navigation: dmc.Button is designed for actions. To create a hyperlink that looks like a button or to navigate to a new URL, please use the dmc.Anchor component, which can be styled accordingly. For navigation within a dmc.Menu, the dmc.MenuItem component accepts an href prop.2

Alternatives for Link-like Behavior

When hyperlink functionality is required, developers should use the appropriate components:

1. **dmc.Anchor**: This is the primary component for creating hyperlinks. It can be styled to resemble a button if necessary, while semantically representing a link. Detailed documentation for dmc.Anchor is provided in the subsequent section.
2. **dmc.MenuItem (within dmc.Menu)**: If the link is part of a dropdown menu, the dmc.MenuItem component accepts an href property, allowing it to function as a navigational link.2

Python

```
import dash_mantine_components as dmc
from dash_iconify import DashIconify

dmc.Menu()
])
```

Using the correct component for the intended purpose ensures semantic correctness and leverages the designed functionalities of dash-mantine-components.

dmc.Anchor

The dmc.Anchor component is used to create hyperlinks with Mantine's theme styles.3 It functions similarly to dcc.Link for multipage applications and is essentially a wrapper around the dmc.Text component, inheriting its text styling capabilities.3

Simple Usage Example

To create a basic link:

Python

```
import dash_mantine_components as dmc

dmc.Anchor(
"Visit Dash Mantine Components",
```

href="https://www.dash-mantine-components.com/",
)

To open a link in a new tab, use the target prop:

Python

import dash_mantine_components as dmc

dmc.Anchor(
"Dash Mantine Components (New Tab)",
href="https://www.dash-mantine-components.com/",
target="_blank" # Opens in a new tab
)

Key Features and Props

- **href (Required):** This string property specifies the URL the hyperlink points to.3
- **target:** Defines where to open the linked document. Common values include _self (default, opens in the same frame) and _blank (opens in a new tab or window).3
- **underline Prop:** Controls the text decoration for the link. It accepts the following string values 3:
    - "hover" (default): Underline appears only when the mouse hovers over the link.
    - "always": The link is always underlined.
    - "never": The link is never underlined.

Python
import dash_mantine_components as dmc

dmc.Group([
dmc.Anchor("Underline always", href="#", underline="always"),
dmc.Anchor("Underline on hover (default)", href="#", underline="hover"),
dmc.Anchor("Underline never", href="#", underline="never"),
])

- **Text Props Inheritance:** As dmc.Anchor is a wrapper around dmc.Text, it supports all dmc.Text component props. This allows for rich text styling, including variant, gradient, size, fw (font weight), fz (font size), c (color), and more.3
  For example, to create an anchor with a gradient style:
  Python
  import dash_mantine_components as dmc

  dmc.Anchor(
  "Gradient Link to Text Props",
  href="#text-props",
  variant="gradient",
  gradient={"from": "pink", "to": "yellow"},
  fw=500,
  fz="lg"
)

Keyword Arguments for dmc.Anchor

The following table details the keyword arguments (props) available for the dmc.Anchor component. This structured reference is essential for developers to quickly understand the available customization options and their expected data types, ensuring efficient and accurate component implementation.3

Name	Description	Type
children	Content of the anchor.	any
id	Unique ID to identify this component in Dash callbacks.	string
href	The URL the hyperlink points to.	string; required
target	Specifies where to open the linked document (e.g., _blank, _self).	'_blank', '_self'
underline	Determines in which cases link should have text-decoration: underline styles, hover by default.	'always', 'hover', 'never'
variant	Text variant (e.g., "gradient", "text"). Inherited from dmc.Text.	string
gradient	Gradient configuration, used when variant is gradient.	dict
size	Controls font-size and line-height. Inherited from dmc.Text.	optional ('xs', 'sm', 'md', 'lg', 'xl', number)
inherit	Determines whether font properties should be inherited from the parent, False by default.	boolean
inline	Sets line-height to 1 for centering, False by default.	boolean
lineClamp	Number of lines after which Text will be truncated.	number
truncate	Side on which Text must be truncated. If True or 'start', text is truncated from the start.	optional ('start', boolean)
refresh	Whether to refresh the page when the link is clicked. False by default.	boolean
className	Class added to the root element, if applicable.	string
classNames	Adds class names to Mantine components' inner elements.	dict
styles	Mantine styles API to apply inline styles to inner elements.	dict
darkHidden	Determines whether component should be hidden in dark color scheme with display: none.	boolean
lightHidden	Determines whether component should be hidden in light color scheme with display: none.	boolean
hiddenFrom	Breakpoint above which the component is hidden with display: none.	optional (e.g., 'sm', 'md')

Name	Description	Type
visibleFrom	Breakpoint below which the component is hidden with display: none.	optional (e.g., 'sm', 'md')
mod	Element modifiers transformed into data- attributes.	string or dict
tabIndex	tab-index.	number
loading_state	Object that holds the loading state object coming from dash-renderer. For use with dash<3.	dict

Styles API

dmc.Anchor supports the Mantine Styles API, allowing for fine-grained customization of its inner elements through the styles and classNames props.3 For comprehensive control over component styling beyond standard props, developers can refer to the general Styles API documentation and the theming section of the Dash Mantine Components documentation.4 While specific selectors for dmc.Anchor are not detailed here, its inheritance from dmc.Text means that text-related styling capabilities are extensive.

Newly Documented and Updated Components in v1.3.0

This section details components that are either new, have received significant updates in version 1.3.0, or are essential components whose documentation is now being added for completeness. A consistent documentation structure is applied to facilitate ease of learning and use. This typically includes an introduction, usage examples, key feature explanations, a keyword arguments table, and Styles API information.

dmc.Table

The dmc.Table component is utilized for displaying tabular data, styled according to the Mantine theme. It serves as an enhanced alternative to the standard html.Table.6 dmc.Table is particularly well-suited for smaller datasets where seamless integration with the Mantine theme and the ability to embed other Dash components directly within cells are desired.7 For large datasets, performance-intensive UIs, or advanced features like filtering, sorting, and virtualization, dash-ag-grid remains the recommended solution.7

New Features in v1.3.0 7

Version 1.3.0 introduces several enhancements to dmc.Table:

- **Scrollable Tables:** The dmc.TableScrollContainer component can now be used to make tables scrollable.
    - Set minWidth on dmc.TableScrollContainer to enable horizontal scrolling.
    - Set maxHeight on dmc.TableScrollContainer to limit vertical height and enable vertical scrolling.
    - The type prop of dmc.TableScrollContainer can be set to "scrollarea" for Mantine-styled scrollbars or "native" for native browser scrollbars.
- **Vertical Variant:** A new variant="vertical" option renders the table in a pivot-table-like style. This is ideal for detail views, displaying metadata, or summarizing records.

- **tableProps Support:** This powerful feature allows passing native HTML attributes (e.g., rowSpan, colSpan) directly to <td>, <tr>, and <th> elements within the table. This is achieved by passing a dictionary to the tableProps argument of dmc.Table, enabling more complex table layouts, including merged cells.

Simple Usage Example (Basic Table)

Python

```
import dash_mantine_components as dmc
from dash import html

dmc.Table(
)
),
html.Tbody(
),
html.Tr(),
]
),
],
captionSide="bottom",
caption="Employee Data"
)
```

Example for v1.3.0 Feature (tableProps for rowSpan and colSpan)

This example demonstrates using tableProps to achieve rowSpan and colSpan for merged cells, a new capability in version 1.3.0.7

Python

```
import dash_mantine_components as dmc
from dash import html

dmc.Table(
)
),
html.Tbody(),
html.Tr(),
html.Tr()
])
],
withBorder=True,
withColumnBorders=True
)
```

Keyword Arguments for dmc.Table

Name	Description	Type
children	Typically html.Thead, html.Tbody, html.Tfoot containing rows (html.Tr) and cells (html.Th, html.Td).	any
id	Unique ID for Dash callbacks.	string
captionSide	Position of the <caption> element: "top" or "bottom".	'top', 'bottom'
caption	Content for the table's <caption>.	string
fz	Font size for the table content. Can be a theme key ('xs', 'sm') or a number (pixels).	string or number
highlightOnHover	If True, table rows will be highlighted on mouse hover.	boolean
horizontalSpacing	Horizontal spacing between table cells. Can be a theme key ('xs', 'sm') or a number (pixels).	string or number
layout	Table layout algorithm: 'auto' or 'fixed'.	'auto', 'fixed'
striped	If True, rows will have alternating background colors.	boolean
stripedColor	Background color for striped rows when striped is True. Uses theme color if not specified.	string
verticalSpacing	Vertical spacing between table cells. Can be a theme key ('xs', 'sm') or a number (pixels).	string or number
withBorder	If True, the table will have an outer border.	boolean
withColumnBorders	If True, the table will have vertical borders between columns.	boolean
borderColor	Color of the borders, if withBorder or withColumnBorders is True.	string
variant	Table style variant. Includes "vertical" (new in 1.3.0) for a pivot-table like display.	string
tableProps	Pass native HTML attributes to <td>, <tr>, and <th> elements. Example: {"td": {"rowSpan": 2}, "th": {"colSpan": 3}} (New in 1.3.0)	dict
className	Adds a class name to the root element.	string
styles	Mantine styles API to apply inline styles to inner elements.	dict
classNames	Adds class names to Mantine components' inner elements.	dict
loading_state	Object that holds the loading state object coming from dash-renderer. For use with dash<3.	dict

Keyword Arguments for dmc.TableScrollContainer 7

Name	Description	Type

Name	Description	Type
children	The dmc.Table component to be made scrollable.	any
id	Unique ID for Dash callbacks.	string
minWidth	Minimum width in pixels to enable horizontal scrolling.	number
maxHeight	Maximum height in pixels to limit vertical scrolling.	number
type	Scrollbar type: "scrollarea" (Mantine) or "native" (browser default).	string

Styles API for dmc.Table

dmc.Table supports the Styles API, allowing customization of its internal elements via the styles and classNames props. Refer to the general Styles API documentation for more details on how to apply targeted styles.

dmc.Carousel (and dmc.CarouselSlide)

The dmc.Carousel component is used to display a slideshow or a series of content items that users can navigate through. It consists of the main dmc.Carousel container and individual dmc.CarouselSlide components for each item in the carousel.8

Simple Usage Example

Python

```
import dash_mantine_components as dmc
from dash import html

dmc.Carousel(
})
),
dmc.CarouselSlide(
html.Div("Slide 2", style={"padding": "20px", "backgroundColor": dmc.theme.DEFAULT_COLORS["red"]})
),
dmc.CarouselSlide(
html.Div("Slide 3", style={"padding": "20px", "backgroundColor": dmc.theme.DEFAULT_COLORS["green"]})
),
],
id="my-carousel",
height=200,
loop=True,
withIndicators=True, # Assuming this common prop exists or use withControls
controlsOffset="sm",
controlSize=30
)
```

Key Features and Props 8

- **children**: Expects a list of dmc.CarouselSlide components.
- **orientation**: Can be 'horizontal' (default) or 'vertical'.
- **slideSize**: Defines the width (for horizontal) or height (for vertical) of each slide, often as a percentage (e.g., '100%', '50%').
- **slideGap**: Specifies the space between slides (e.g., 'md', 10).
- **align**: Alignment of slides within the container: 'start', 'center', or 'end'.
- **slidesToScroll**: Number of slides to move with each navigation action.
- **loop**: Boolean, enables infinite looping of slides.
- **withControls**: Boolean, shows or hides the next/previous navigation buttons.
- **withIndicators**: (Often available, check specific version) Boolean, shows or hides dot indicators for slides.
- **controlSize, controlsOffset**: Customize the appearance and positioning of navigation controls.
- **initialSlide**: Index of the slide to display initially.
- **autoplay**: Dictionary to configure automatic slide transitions (e.g., {"delay": 4000}).
- **dragFree**: Boolean, enables momentum-based scrolling after drag.

Keyword Arguments for dmc.Carousel

The dmc.Carousel component offers a rich set of properties for fine-tuning its behavior and appearance. The following table, derived from its documentation 8, provides a comprehensive list.

Name	Description	Type
children	<Carousel.Slide /> components.	any
id	Unique ID to identify this component in Dash callbacks.	string
active	The index of the current slide. Read only. Use initialSlide to set.	number; default 0
align	Determines how slides will be aligned relative to the container. 'center' by default.	'start', 'center', 'end'
autoScroll	Enables autoScroll with optional configuration.	dict
autoplay	Enables autoplay with optional configuration (e.g. {"delay": 2000}).	dict
className	Class added to the root element.	string
classNames	Adds class names to inner Mantine components.	dict
containScroll	Clear leading/trailing empty space. Use trimSnaps or keepSnaps.	'', 'trimSnaps', 'keepSnaps'
controlSize	Size of next/previous controls (e.g., 26).	string or number
controlsOffset	Position of next/previous controls (theme spacing key or CSS value, 'sm' by default).	string or number
darkHidden	Hide in dark color scheme.	boolean
dragFree	Enable momentum scrolling (False by default).	boolean

Name	Description	Type
height	Slides container height, required for vertical orientation.	string or number
hiddenFrom	Breakpoint above which component is hidden.	optional
inViewThreshold	Percentage of slide visible to be considered in view (e.g., 0.5 for 50%).	number
includeGapInSize	Treat gap as part of slide size (True by default).	boolean
initialSlide	Index of initial slide (0 by default).	number
lightHidden	Hide in light color scheme.	boolean
loading_state	Dash loading state object.	dict
loop	Enable infinite looping (True by default).	boolean
mod	Element modifiers as data- attributes.	string or dict
nextControlIcon	Icon for the next control.	any
orientation	Carousel orientation ('horizontal' by default).	'horizontal', 'vertical'
previousControlIcon	Icon for the previous control.	any
skipSnaps	Allow skipping scroll snaps on vigorous drag (False by default).	boolean
slideGap	Gap between slides (theme spacing key or number).	string or number
slideSize	Slide width based on viewport ('100%' by default).	string or number
slidesToScroll	Number of slides scrolled with buttons (1 by default).	number
styles	Mantine styles API.	dict
tabIndex	tab-index.	number
withControls	Determines whether next/previous controls are rendered (True by default).	boolean
withIndicators	Determines whether dot indicators are rendered (False by default).	boolean

Keyword Arguments for dmc.CarouselSlide

Each slide within the carousel is defined by a dmc.CarouselSlide component.8

Name	Description	Type
children	Content of the slide.	any
id	Unique ID for Dash callbacks.	string
className	Class added to the root element.	string

Name	Description	Type
classNames	Adds class names to inner Mantine components.	dict
darkHidden	Hide in dark color scheme.	boolean
hiddenFrom	Breakpoint above which component is hidden.	optional
lightHidden	Hide in light color scheme.	boolean
loading_state	Dash loading state object.	dict
mod	Element modifiers as data- attributes.	string or dict
size	Relative size of the slide.	string or number
styles	Mantine styles API.	dict
tabIndex	tab-index.	number
variant	Variant.	string
visibleFrom	Breakpoint below which component is hidden.	optional

Styles API for dmc.Carousel

dmc.Carousel and dmc.CarouselSlide support the Styles API for advanced customization of their constituent parts. Developers can use the styles and classNames props to target specific inner elements.

Other Components for Documentation Expansion

Consistent and detailed documentation, following the template of an introduction, usage examples, key features, a keyword arguments table, and Styles API notes, should also be developed for the following components, which are either updated in v1.3.0 or are fundamental to building applications with dash-mantine-components:

- **dmc.Timeline and dmc.TimelineItem**: Version 1.3.0 enables Dash callbacks on components passed as props within dmc.Timeline.7 Documentation should cover its usage for displaying chronological events, props like active, color, bulletSize, and the structure using dmc.TimelineItem.9 The Styles API and data attributes specific to Timeline 9 are important for customization.
- **dmc.Stepper and dmc.StepperStep**: Similar to Timeline, callbacks on components as props are now supported.7 Documentation should explain its use for multi-step processes, props like active, color, iconPosition, and the role of dmc.StepperStep.10
- **dmc.CodeHighlight, dmc.CodeHighlightTabs, dmc.InlineCodeHighlight**: Callbacks on components as props are enabled.7 A key update is that CSS for CodeHighlight is now bundled automatically as of DMC 1.2.0, simplifying setup.11 Documentation should cover these three components for different code display needs, their props (code, language, withCopyButton, withExpandButton), and how CodeHighlightTabs handles multiple code blocks.11
- **dmc.SegmentedControl**: This component saw a fix for transitionDuration and now supports callbacks on components passed as props.7 Its documentation should explain its use for selecting one option from a set, typically displayed as connected buttons.

- **dmc.List and dmc.ListItem**: For creating ordered and unordered lists with icon support and nesting capabilities.12
- **dmc.Center**: A layout component to center content vertically and horizontally, with an inline prop option.13
- **dmc.Menu, dmc.MenuTarget, dmc.MenuDropdown, dmc.MenuItem**: For creating dropdown menus. dmc.MenuItem can act as a button or a link (using href).2
- **dmc.ActionIcon**: An alternative to dmc.Button for icon-only buttons, with various styling options and a loading state.14
- **dmc.Highlight**: To highlight substrings within a text, with customizable highlight styles.15
- **dmc.Pagination**: For navigating paged content, with props to control siblings, boundaries, and appearance.16
- **dmc.Tree**: To display hierarchical data structures, with features like checkboxes, node expansion control, and icon customization.17
- **dmc.Grid and dmc.GridCol**: For creating responsive layouts using a flexbox grid system, with control over column spans, gutters, and offsets.18

The systematic addition of documentation for these components will greatly enhance the utility of claude.md as a comprehensive resource for dash-mantine-components version 1.3.0.

Maintenance and Best Practices for this Guide

To ensure this guide remains accurate, relevant, and user-friendly, the following practices are recommended for its ongoing maintenance:

- **Version Pinning:** This document is specifically tailored for dash-mantine-components==1.3.0. This versioning should be clearly stated at the beginning of the guide. Future updates to the dash-mantine-components library should ideally result in new, version-specific guides or clearly demarcated sections within this document to prevent confusion and ensure developers are referencing information pertinent to their library version.
- **Consistency with Official Documentation:** Maintainers should regularly cross-reference this guide with the official dash-mantine-components website (dash-mantine-components.com). This practice will help in identifying new components, changes to existing component props, deprecations, and updates to best practices, ensuring this guide remains a reliable and current resource.
- **Community Contributions:** If feasible, establishing a straightforward process for community members to report discrepancies, suggest improvements, or contribute updates can significantly enhance the quality and completeness of this guide. This could involve a dedicated feedback channel or a contribution workflow.
- **Focus on Practical Examples:** The inclusion of practical, easy-to-understand, and copy-paste-friendly code examples is crucial. Developers often learn most effectively from working examples that demonstrate common use cases for each component.1 Examples should be concise and illustrative of key functionalities.
- **Styles API Documentation:** For components that support the Mantine Styles API, documentation should not only mention this support but also, where possible, list key selectors (e.g., root, label, input for form components; item for list-like components).4 Referring to a general Styles API explanation within the guide is also beneficial. Providing these details empowers developers to leverage the full customization potential of dash-mantine-components. The Styles API, which allows

targeting inner elements of components using styles and classNames props, is a powerful feature for achieving bespoke designs.5

Works cited

1. The+UI_UX+Playbook+-+Learn+tips+and+tricks+to+design+beautiful+UI+designs-compressed.pdf
2. Mantine API Overview, accessed May 31, 2025, https://www.dash-mantine-components.com/mantine-api
3. Theme Object - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/theme-object
4. MantineProvider - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/mantineprovider
5. Dash Mantine Components Theme Builder - Plotly Community Forum, accessed May 31, 2025, https://community.plotly.com/t/dash-mantine-components-theme-builder/89883
6. www.dash-mantine-components.com, accessed May 31, 2025, https://www.dash-mantine-components.com/mantine-provider
7. Colors - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/colors
8. CSS Variables - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/css-variables
9. Typography - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/typography
10. Styles API - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/styles-api
11. www.dash-mantine-components.com, accessed May 31, 2025, https://www.dash-mantine-components.com/components/colorswatch
12. Dash Mantine Components | Title, accessed May 31, 2025, https://www.dash-mantine-components.com/components/title
13. Dash Mantine Components | Text, accessed May 31, 2025, https://www.dash-mantine-components.com/components/text
14. Grid - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/grid
15. SimpleGrid - Mantine, accessed May 31, 2025, https://mantine.dev/core/simple-grid/
16. Dash Mantine Components 0.15.2 Release - Plotly Community Forum, accessed May 31, 2025, https://community.plotly.com/t/dash-mantine-components-0-15-2-release/90163
17. SimpleGrid - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/simplegrid
18. Stack - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/stack
19. Group - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/group
20. AspectRatio - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/aspectratio
21. Flex - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/flex
22. Space - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/space

23. Container - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/container
24. Dash Mantine Components | Table, accessed May 31, 2025, https://www.dash-mantine-components.com/components/table
25. Center - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/center
26. Responsive Styles - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/responsive-styles
27. AppShell - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/appshell
28. AppShell | Mantine, accessed May 31, 2025, https://mantine.dev/core/app-shell/
29. Dash Mantine Components 1.3.0 Release - Dash Python - Plotly Community Forum, accessed May 31, 2025, https://community.plotly.com/t/dash-mantine-components-1-3-0-release/92379
30. Card - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/card
31. HoverCard - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/hovercard
32. List - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/list
33. Tree - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/tree
34. Accordion - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/accordion
35. Accordion - dbc docs - Dash Bootstrap Components, accessed May 31, 2025, https://dash-bootstrap-components.opensource.faculty.ai/docs/components/accordion/
36. InputWrapper - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/inputwrapper
37. Spoiler - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/spoiler
38. Alert - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/alert
39. Notification - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/notification
40. Badge - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/badge
41. SemiCircleProgress - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/semicircleprogress
42. Progress - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/progress
43. RingProgress - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/ringprogress
44. Help Center - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/help-center
45. www.dash-mantine-components.com, accessed May 31, 2025, https://www.dash-mantine-components.com/components/ring-progress

46. ThemeIcon - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/themeicon
47. Style Props - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/style-props
48. Dash Mantine Components | Blockquote, accessed May 31, 2025, https://www.dash-mantine-components.com/components/blockquote
49. Code - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/code
50. Highlight - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/highlight
51. RichTextEditor - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/richtexteditor
52. ColorInput - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/colorinput
53. Charts Introduction - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/chartsintro
54. AreaChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/areachart
55. BarChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/barchart
56. BubbleChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/bubblechart
57. CompositeChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/compositechart
58. DonutChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/donutchart
59. LineChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/linechart
60. PieChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/piechart
61. RadarChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/radarchart
62. ScatterChart - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/scatterchart
63. Sparkline - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/sparkline
64. Image - Dash Mantine Components, accessed May 31, 2025, https://www.dash-mantine-components.com/components/image
65. Button - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/button
66. Menu - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/menu
67. Anchor - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/anchor
68. Styles API - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/styles-api

69. Mantine API Overview, accessed June 1, 2025, https://www.dash-mantine-components.com/mantine-api
70. Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/table
71. Dash Mantine Components 1.3.0 Release - Dash Python - Plotly ..., accessed June 1, 2025, https://community.plotly.com/t/dash-mantine-components-1-3-0-release/92379
72. Carousel - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/carousel
73. Dash Mantine Components | Timeline, accessed June 1, 2025, https://www.dash-mantine-components.com/components/timeline
74. Dash Mantine Components | Colors, accessed June 1, 2025, https://www.dash-mantine-components.com/components/stepper
75. CodeHighlight - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/code-highlight
76. List - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/list
77. Center - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/center
78. ActionIcon - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/actionicon
79. Highlight - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/highlight
80. Dash Mantine Components | Pagination, accessed June 1, 2025, https://www.dash-mantine-components.com/components/pagination
81. Tree - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/tree
82. Grid - Dash Mantine Components, accessed June 1, 2025, https://www.dash-mantine-components.com/components/grid