Global Settings & Styles (theme.json)

- Rationale
  - Settings for the block editor
  - Settings can be controlled per block
  - Styles are managed
  - CSS Custom Properties: presets & custom
- Specification
  - Version
  - Settings
  - Opt-in into UI controls
    - Backward compatibility with add_theme_support
    - Presets
    - Custom
    - Settings examples
  - Styles
  - Top-level styles
  - Block styles
    - Referencing a style
    - Element styles
      - Element pseudo selectors
    - Responsive styles
    - Variations
      - Variation Theme.json Partials
      - Programmatically Registering Variation Styles
  - customTemplates
  - templateParts
  - patterns

- Developing with theme.json
- Frequently Asked Questions
  - The naming schema of CSS Custom Properties
  - Why using -- as a separator?
  - How settings under “custom” create new CSS Custom Properties
  - Global Stylesheet
  - Specificity for link colors provided by the user
  - What is blockGap and how can I use it?
  - Why does it take so long to update the styles in the browser?

WordPress 5.8 comes with a new mechanism to configure the editor that enables a finer-grained control and introduces the first step in managing styles for future WordPress releases: the `theme.json` file.

Rationale

The Block Editor API has evolved at different velocities and there are some growing pains, specially in areas that affect themes. Examples of this are: the ability to control the editor programmatically, or a block style system that facilitates user, theme, and core style preferences.

This describes the current efforts to consolidate the various APIs related to styles into a single point – a `theme.json` file that should be located inside the root of the theme directory.

Settings for the block editor

Instead of the proliferation of theme support flags or alternative methods, the `theme.json` files provides a canonical way to define the settings of the block editor. These settings includes things like:

- What customization options should be made available or hidden from the user.
- What are the default colors, font sizes… available to the user.
- Defines the default layout of the editor (widths and available alignments).

Settings can be controlled per block

For more granularity, these settings also work at the block level in `theme.json`.

Examples of what can be achieved are:

- Use a particular preset for a block (e.g.: table) but the common one for the rest of blocks.
- Enable font size UI controls for all blocks but the headings block.
- etc.

Styles are managed

By using the `theme.json` file to set style properties in a structured way, the Block Editor can "manage" the CSS that comes from different origins (user, theme, and core CSS). For example, if a theme and a user set the font size for paragraphs, we only enqueue the style coming from the user and not the theme's.

Some of the advantages are:

- Reduce the amount of CSS enqueued.
- Prevent specificity wars.

CSS Custom Properties: presets & custom

There are some areas of styling that would benefit from having shared values that can change across a site.

To address this need, we've started to experiment with CSS Custom Properties, aka CSS Variables, in some places:

- **Presets**: color palettes, font sizes, or gradients declared by the theme are converted to CSS Custom Properties and enqueued both the front-end and the editors.

Input

Output

```
{
    "version": 3,
    "settings": {
        "color": {
            "palette": [
                {
                    "name": "Black",
                    "slug": "black",
                    "color": "#000000"
```

```
                },
                {
                    "name": "White",
                    "slug": "white",
                    "color": "#ffffff"
                }
            ]
        }
    }
}
```

```
body {
    --wp--preset--color--black: #000000;
    --wp--preset--color--white: #ffffff;
}
```

- **Custom properties**: there’s also a mechanism to create your own CSS Custom Properties.

Input

Output

```
{
    "version": 3,
    "settings": {
        "custom": {
            "line-height": {
                "body": 1.7,
                "heading": 1.3
            }
        }
    }
}
```

```
body {
    --wp--custom--line-height--body: 1.7;
    --wp--custom--line-height--heading: 1.3;
}
```

Specification

This specification is the same for the three different origins that use this format: core, themes, and users. Themes can override core's defaults by creating a file called `theme.json`. Users, via the site editor, will also be able to override theme's or core's preferences via a user interface that is being worked on.

```
{
    "version": 3,
    "settings": {},
    "styles": {},
    "customTemplates": {},
    "templateParts": {}
}
```

Version

This field describes the format of the `theme.json` file. The latest version is version 3 introduced in WordPress 6.6.

New versions are introduced when a breaking change needs to be made. This allows theme authors to choose when to opt-in to the breaking changes and migrate their theme.json files to the new format.

Older versions of `theme.json` are backwards-compatible and will continue to work with newer versions of WordPress and the Gutenberg plugin. However new features will be developed on the latest version.

Follow the instructions in migrating to newer versions for details on updating to the latest version.

Settings

> The Gutenberg plugin extends the settings available from WordPress 5.8, so they can be used with other WordPress versions and they go through a maturation process before being ported to core.
>
> The tabs below show WordPress 5.8 supported settings and the ones supported by the Gutenberg plugin.

The settings section has the following structure:

WordPress

Gutenberg

```
{
    "version": 3,
    "settings": {
        "border": {
            "radius": false,
            "color": false,
            "style": false,
            "width": false
        },
        "color": {
            "custom": true,
            "customDuotone": true,
            "customGradient": true,
            "duotone": [],
            "gradients": [],
            "link": false,
            "palette": [],
            "text": true,
            "background": true,
            "defaultGradients": true,
            "defaultPalette": true
        },
        "custom": {},
        "layout": {
            "contentSize": "800px",
            "wideSize": "1000px"
        },
        "spacing": {
            "margin": false,
            "padding": false,
            "blockGap": null,
            "units": [ "px", "em", "rem", "vh", "vw" ]
        },
        "typography": {
```

```
            "customFontSize": true,
            "lineHeight": false,
            "dropCap": true,
            "fluid": false,
            "fontStyle": true,
            "fontWeight": true,
            "letterSpacing": true,
            "textDecoration": true,
            "textTransform": true,
            "fontSizes": [],
            "fontFamilies": []
        },
        "blocks": {
            "core/paragraph": {
                "color": {},
                "custom": {},
                "layout": {},
                "spacing": {},
                "typography": {}
            },
            "core/heading": {},
            "etc": {}
        }
    }
}
```

```
{
    "version": 3,
    "settings": {
        "appearanceTools": false,
        "border": {
            "color": false,
            "radius": false,
            "style": false,
            "width": false
        },
        "color": {
            "background": true,
            "custom": true,
```

```
        "customDuotone": true,
        "customGradient": true,
        "defaultGradients": true,
        "defaultPalette": true,
        "duotone": [],
        "gradients": [],
        "link": false,
        "palette": [],
        "text": true
    },
    "custom": {},
    "dimensions": {
        "aspectRatio": false,
        "height": false,
        "minHeight": false,
        "minWidth": false,
        "width": false,
    },
    "layout": {
        "contentSize": "800px",
        "wideSize": "1000px"
    },
    "spacing": {
        "blockGap": null,
        "margin": false,
        "padding": false,
        "customSpacingSize": true,
        "units": [ "px", "em", "rem", "vh", "vw" ],
        "spacingScale": {
            "operator": "*",
            "increment": 1.5,
            "steps": 7,
            "mediumStep": 1.5,
            "unit": "rem"
        },
        "spacingSizes": []
    },
    "typography": {
        "customFontSize": true,
        "dropCap": true,
```

```
            "fluid": false,
            "fontFamilies": [],
            "fontSizes": [],
            "fontStyle": true,
            "fontWeight": true,
            "letterSpacing": true,
            "lineHeight": false,
            "textAlign": true,
            "textColumns": false,
            "textDecoration": true,
            "textIndent": true,
            "textTransform": true
        },
        "blocks": {
            "core/paragraph": {
                "border": {},
                "color": {},
                "custom": {},
                "layout": {},
                "spacing": {},
                "typography": {}
            },
            "core/heading": {},
            "etc": {}
        }
    }
}
```

Each block can configure any of these settings separately, providing a more fine-grained control over what exists via `add_theme_support`. The settings declared at the top-level affect to all blocks, unless a particular block overwrites it. It's a way to provide inheritance and configure all blocks at once.

Note, however, that not all settings are relevant for all blocks. The settings section provides an opt-in/opt-out mechanism for themes, but it's the block's responsibility to add support for the features that are relevant to it. For example, if a block doesn't implement the `dropCap` feature, a theme can't enable it for such a block through `theme.json`.

Opt-in into UI controls

There’s one special setting property, `appearanceTools`, which is a boolean and its default value is false. Themes can use this setting to enable the following ones:

- background: backgroundImage, backgroundSize
- border: color, radius, style, width
- color: link
- dimensions: aspectRatio, height, minHeight, minWidth, width
- position: sticky
- spacing: blockGap, margin, padding
- typography: lineHeight

Backward compatibility with add_theme_support

To retain backward compatibility, the existing `add_theme_support` declarations that configure the block editor are retrofit in the proper categories for the top-level section. For example, if a theme uses `add_theme_support('disable-custom-colors')`, it’ll be the same as setting `settings.color.custom` to `false`. If the `theme.json` contains any settings, these will take precedence over the values declared via `add_theme_support`. This is the complete list of equivalences:

add_theme_support	theme.json setting
`custom-line-height`	Set `typography.lineHeight` to `true`.
`custom-spacing`	Set `spacing.padding` to `true`.
`custom-units`	Provide the list of units via `spacing.units`.
`disable-custom-colors`	Set `color.custom` to `false`.
`disable-custom-font-sizes`	Set `typography.customFontSize` to `false`.
`disable-custom-gradients`	Set `color.customGradient` to `false`.
`editor-color-palette`	Provide the list of colors via `color.palette`.
`editor-font-sizes`	Provide the list of font size via `typography.fontSizes`.
`editor-gradient-presets`	Provide the list of gradients via `color.gradients`.
`editor-spacing-sizes`	Provide the list of spacing sizes via `spacing.spacingSizes`.
`appearance-tools`	Set `appearanceTools` to `true`.
`border`	Set `border: color, radius, style, width` to `true`.
`link-color`	Set `color.link` to `true`.

Presets

Presets are part of the settings section. They are values that are shown to the user via some UI controls. By defining them via `theme.json` the engine can do more for themes, such as automatically translate the preset name or enqueue the corresponding CSS classes and custom properties.

The following presets can be defined via `theme.json`:

- `color.duotone`: doesn't generate classes or custom properties.
- `color.gradients`: generates a single class and custom property per preset value.
- `color.palette`:
  - generates 3 classes per preset value: color, background-color, and border-color.
  - generates a single custom property per preset value.
- `spacing.spacingSizes` / `spacing.spacingScale`: generates a single custom property per preset value.
- `typography.fontSizes`: generates a single class and custom property per preset value.
- `typography.fontFamilies`: generates a single custom property per preset value.
- `typography.textShadowPresets`: generates a single class and custom property per preset value.

The naming schema for the classes and the custom properties is as follows:

- Custom Properties: `--wp--preset--{preset-category}--{preset-slug}` such as `--wp--preset--color--black`
- Classes: `.has-{preset-slug}-{preset-category}` such as `.has-black-color`.

Input

Output

```
{
    "version": 3,
    "settings": {
        "color": {
            "duotone": [
                {
                    "colors": [ "#000", "#FFF" ],
                    "slug": "black-and-white",
                    "name": "Black and White"
                }
            ],
            "gradients": [
                {
                    "slug": "blush-bordeaux",
                    "gradient": "linear-gradient(135deg,rgb(254,205,165) 0%,rgb(254,45,45)
50%,rgb(107,0,62) 100%)",
                    "name": "Blush bordeaux"
                },
                {
```

```
                    "slug": "blush-light-purple",
                    "gradient": "linear-gradient(135deg,rgb(255,206,236) 0%,rgb(152,150,240)
100%)",
                    "name": "Blush light purple"
                }
            ],
            "palette": [
                {
                    "slug": "strong-magenta",
                    "color": "#a156b4",
                    "name": "Strong magenta"
                },
                {
                    "slug": "very-dark-grey",
                    "color": "rgb(131, 12, 8)",
                    "name": "Very dark grey"
                }
            ]
        },
        "typography": {
            "fontFamilies": [
                {
                    "fontFamily": "-apple-system,BlinkMacSystemFont,\"Segoe
UI\",Roboto,Oxygen-Sans,Ubuntu,Cantarell, \"Helvetica Neue\",sans-serif",
                    "slug": "system-font",
                    "name": "System Font"
                },
                {
                    "fontFamily": "Helvetica Neue, Helvetica, Arial, sans-serif",
                    "slug": "helvetica-arial",
                    "name": "Helvetica or Arial"
                }
            ],
            "fontSizes": [
                {
                    "slug": "big",
                    "size": 32,
                    "name": "Big"
                },
                {
```

```
            "slug": "x-large",
            "size": 46,
            "name": "Large"
        }
    ]
},
"spacing": {
    "spacingScale": {
        "operator": "*",
        "increment": 1.5,
        "steps": 7,
        "mediumStep": 1.5,
        "unit": "rem"
    },
    "spacingSizes": [
        {
            "slug": "40",
            "size": "1rem",
            "name": "Small"
        },
        {
            "slug": "50",
            "size": "1.5rem",
            "name": "Medium"
        },
        {
            "slug": "60",
            "size": "2rem",
            "name": "Large"
        }
    ]
},
"blocks": {
    "core/group": {
        "color": {
            "palette": [
                {
                    "slug": "black",
                    "color": "#000000",
                    "name": "Black"
```

```
                        },
                        {
                            "slug": "white",
                            "color": "#ffffff",
                            "name": "White"
                        }
                    ]
                }
            }
        }
    }
}
```

```
/* Top-level custom properties */
body {
    --wp--preset--color--strong-magenta: #a156b4;
    --wp--preset--color--very-dark-grey: #444;
    --wp--preset--gradient--blush-bordeaux: linear-gradient( 135deg, rgb( 254, 205, 165 ) 0%, rgb( 254, 45, 45 ) 50%, rgb( 107, 0, 62 ) 100% );
    --wp--preset--gradient--blush-light-purple: linear-gradient( 135deg, rgb( 255, 206, 236 ) 0%, rgb( 152, 150, 240 ) 100% );
    --wp--preset--font-size--x-large: 46;
    --wp--preset--font-size--big: 32;
    --wp--preset--font-family--helvetica-arial: Helvetica Neue, Helvetica, Arial, sans-serif;
    --wp--preset--font-family--system: -apple-system,BlinkMacSystemFont,\"Segoe UI\",Roboto,Oxygen-Sans,Ubuntu,Cantarell, \"Helvetica Neue\",sans-serif;
    --wp--preset--spacing--20: 0.44rem;
    --wp--preset--spacing--30: 0.67rem;
    --wp--preset--spacing--40: 1rem;
    --wp--preset--spacing--50: 1.5rem;
    --wp--preset--spacing--60: 2.25rem;
    --wp--preset--spacing--70: 3.38rem;
    --wp--preset--spacing--80: 5.06rem;
}

/* Block-level custom properties (bounded to the group block) */
.wp-block-group {
    --wp--preset--color--black: #000000;
    --wp--preset--color--white: #ffffff;
```

```
}

/* Top-level classes */
.has-strong-magenta-color { color: #a156b4 !important; }
.has-strong-magenta-background-color { background-color: #a156b4 !important; }
.has-strong-magenta-border-color { border-color: #a156b4 !important; }
.has-very-dark-grey-color { color: #444 !important; }
.has-very-dark-grey-background-color { background-color: #444 !important; }
.has-very-dark-grey-border-color { border-color: #444 !important; }
.has-blush-bordeaux-background { background: linear-gradient( 135deg, rgb( 254, 205, 165 ) 0%, rgb( 254, 45, 45 ) 50%, rgb( 107, 0, 62 ) 100% ) !important; }
.has-blush-light-purple-background { background: linear-gradient( 135deg, rgb( 255, 206, 236 ) 0%, rgb( 152, 150, 240 ) 100% ) !important; }
.has-big-font-size { font-size: 32; }
.has-normal-font-size { font-size: 16; }

/* Block-level classes (bounded to the group block, wrapped in :where() to keep the same specificity as top-level classes) */
:where(.wp-block-group).has-black-color { color: #a156b4 !important; }
:where(.wp-block-group).has-black-background-color { background-color: #a156b4 !important; }
:where(.wp-block-group).has-black-border-color { border-color: #a156b4 !important; }
:where(.wp-block-group).has-white-color { color: #444 !important; }
:where(.wp-block-group).has-white-background-color { background-color: #444 !important; }
:where(.wp-block-group).has-white-border-color { border-color: #444 !important; }
```

To maintain backward compatibility, the presets declared via `add_theme_support` will also generate the CSS Custom Properties. If the `theme.json` contains any presets, these will take precedence over the ones declared via `add_theme_support`.

Preset classes are attached to the content of a post by some user action. That’s why the engine will add `!important` to these, because user styles should take precedence over theme styles.

Custom

In addition to create CSS Custom Properties for the presets, the `theme.json` also allows for themes to create their own, so they don’t have to be enqueued separately. Any values declared within the `custom` field will be transformed to CSS Custom Properties following this naming schema: `--wp--custom--<variable-name>`.

For example:

Input

Output

```
{
    "version": 3,
    "settings": {
        "custom": {
            "baseFont": 16,
            "lineHeight": {
                "small": 1.2,
                "medium": 1.4,
                "large": 1.8
            }
        },
        "blocks": {
            "core/group": {
                "custom": {
                    "baseFont": 32
                }
            }
        }
    }
}
```

```
body {
    --wp--custom--base-font: 16;
    --wp--custom--line-height--small: 1.2;
    --wp--custom--line-height--medium: 1.4;
    --wp--custom--line-height--large: 1.8;
}
.wp-block-group {
    --wp--custom--base-font: 32;
}
```

Note that the name of the variable is created by adding `--` in between each nesting level and `camelCase` fields are transformed to `kebab-case`.

Settings examples

- Enable custom colors only for the paragraph block:

```
{
    "version": 3,
    "settings": {
        "color": {
            "custom": false
        },
        "blocks": {
            "core/paragraph": {
                "color": {
                    "custom": true
                }
            }
        }
    }
}
```

- Disable border radius for the button block:

```
{
    "version": 3,
    "settings": {
        "blocks": {
            "core/button": {
                "border": {
                    "radius": false
                }
            }
        }
    }
}
```

- Provide the group block a different palette than the rest:

```
{
    "version": 3,
    "settings": {
        "color": {
            "palette": [
                {
                    "slug": "black",
```

```
                "color": "#000000",
                "name": "Black"
            },
            {
                "slug": "white",
                "color": "#FFFFFF",
                "name": "White"
            },
            {
                "slug": "red",
                "color": "#FF0000",
                "name": "Red"
            },
            {
                "slug": "green",
                "color": "#00FF00",
                "name": "Green"
            },
            {
                "slug": "blue",
                "color": "#0000FF",
                "name": "Blue"
            }
        ]
    },
    "blocks": {
        "core/group": {
            "color": {
                "palette": [
                    {
                        "slug": "black",
                        "color": "#000000",
                        "name": "Black"
                    },
                    {
                        "slug": "white",
                        "color": "#FFF",
                        "name": "White"
                    }
```

```
                    ]
                }
            }
        }
    }
}
```

Styles

> The Gutenberg plugin extends the styles available from WordPress 5.8, so they can be used with other WordPress versions and they go through a maturation process before being ported to core.
>
> The tabs below show WordPress 5.8 supported styles and the ones supported by the Gutenberg plugin.

Each block declares which style properties it exposes via the block supports mechanism. The support declarations are used to automatically generate the UI controls for the block in the editor. Themes can use any style property via the `theme.json` for any block — it's the theme's responsibility to verify that it works properly according to the block markup, etc.

WordPress

Gutenberg

```
{
    "version": 3,
    "styles": {
        "border": {
            "radius": "value",
            "color": "value",
            "style": "value",
            "width": "value"
        },
        "filter": {
            "duotone": "value"
        },
        "color": {
```

```
        "background": "value",
        "gradient": "value",
        "text": "value"
    },
    "spacing": {
        "blockGap": "value",
        "margin": {
            "top": "value",
            "right": "value",
            "bottom": "value",
            "left": "value",
        },
        "padding": {
            "top": "value",
            "right": "value",
            "bottom": "value",
            "left": "value"
        }
    },
    "typography": {
        "fontSize": "value",
        "fontStyle": "value",
        "fontWeight": "value",
        "letterSpacing": "value",
        "lineHeight": "value",
        "textDecoration": "value",
        "textTransform": "value"
    },
    "elements": {
        "link": {
            "border": {},
            "color": {},
            "spacing": {},
            "typography": {}
        },
        "h1": {},
        "h2": {},
        "h3": {},
        "h4": {},
```

```
            "h5": {},
            "h6": {}
        },
        "blocks": {
            "core/group": {
                "border": {},
                "color": {},
                "spacing": {},
                "typography": {},
                "elements": {
                    "link": {},
                    "h1": {},
                    "h2": {},
                    "h3": {},
                    "h4": {},
                    "h5": {},
                    "h6": {}
                }
            },
            "etc": {}
        }
    }
}
```

```
{
    "version": 3,
    "styles": {
        "border": {
            "color": "value",
            "radius": "value",
            "style": "value",
            "width": "value"
        },
        "color": {
            "background": "value",
            "gradient": "value",
            "text": "value"
        },
        "dimensions": {
```

```
        "aspectRatio": "value",
        "height": "value"
        "minHeight": "value"
        "minWidth": "value"
        "width": "value"
    },
    "filter": {
        "duotone": "value"
    },
    "spacing": {
        "blockGap": "value",
        "margin": {
            "top": "value",
            "right": "value",
            "bottom": "value",
            "left": "value"
        },
        "padding": {
            "top": "value",
            "right": "value",
            "bottom": "value",
            "left": "value"
        }
    },
    "typography": {
        "fontFamily": "value",
        "fontSize": "value",
        "fontStyle": "value",
        "fontWeight": "value",
        "letterSpacing": "value",
        "lineHeight": "value",
        "textColumns": "value",
        "textDecoration": "value",
        "textTransform": "value"
    },
    "elements": {
        "link": {
            "border": {},
            "color": {},
```

```
                "spacing": {},
                "typography": {}
            },
            "h1": {},
            "h2": {},
            "h3": {},
            "h4": {},
            "h5": {},
            "h6": {},
            "heading": {},
            "button": {},
            "caption": {}
        },
        "blocks": {
            "core/group": {
                "border": {},
                "color": {},
                "dimensions": {},
                "spacing": {},
                "typography": {},
                "elements": {
                    "link": {},
                    "h1": {},
                    "h2": {},
                    "h3": {},
                    "h4": {},
                    "h5": {},
                    "h6": {}
                }
            },
            "etc": {}
        }
    }
}
```

Top-level styles

Styles found at the top-level will be enqueued using the `body` selector.

Input

Output

```
{
    "version": 3,
    "styles": {
        "color": {
            "text": "var(--wp--preset--color--primary)"
        }
    }
}
```

```
body {
    color: var( --wp--preset--color--primary );
}
```

Block styles

Styles found within a block will be enqueued using the block selector.

By default, the block selector is generated based on its name such as `.wp-block-<blockname-without-namespace>`. For example, `.wp-block-group` for the `core/group` block. There are some blocks that want to opt-out from this default behavior. They can do so by explicitly telling the system which selector to use for them via the `__experimentalSelector` key within the `supports` section of its `block.json` file. Note that the block needs to be registered server-side for the `__experimentalSelector` field to be available to the style engine.

Input

Output

```
{
    "version": 3,
    "styles": {
        "color": {
            "text": "var(--wp--preset--color--primary)"
        },
        "blocks": {
```

```
            "core/paragraph": {
                "color": {
                    "text": "var(--wp--preset--color--secondary)"
                }
            },
            "core/group": {
                "color": {
                    "text": "var(--wp--preset--color--tertiary)"
                }
            }
        }
    }
}
```

```
body {
    color: var( --wp--preset--color--primary );
}
p { /* The core/paragraph opts out from the default behaviour and uses p as a selector. */
    color: var( --wp--preset--color--secondary );
}
.wp-block-group {
    color: var( --wp--preset--color--tertiary );
}
```

Referencing a style

A block can be styled using a reference to a root level style. This feature is supported by Gutenberg.
If you register a background color for the root using styles.color.background:

```
"styles": {
        "color": {
            "background": "var(--wp--preset--color--primary)"
        }
    }
```

You can use `ref: "styles.color.background"` to reuse the style for a block:

```
{
    "color": {
```

```
        "text": { "ref": "styles.color.background" }
    }
}
```

Element styles

In addition to top-level and block-level styles, there’s the concept of elements that can be used in both places. There’s a closed set of them:

Supported by Gutenberg:

- `button`: maps to the `wp-element-button` CSS class. Also maps to `wp-block-button__link` for backwards compatibility.
- `caption`: maps to the `.wp-element-caption, .wp-block-audio figcaption, .wp-block-embed figcaption, .wp-block-gallery figcaption, .wp-block-image figcaption, .wp-block-table figcaption, .wp-block-video figcaption` CSS classes.
- `heading`: maps to all headings, the `h1 to h6` CSS selectors.

Supported by WordPress:

- `h1`: maps to the `h1` CSS selector.
- `h2`: maps to the `h2` CSS selector.
- `h3`: maps to the `h3` CSS selector.
- `h4`: maps to the `h4` CSS selector.
- `h5`: maps to the `h5` CSS selector.
- `h6`: maps to the `h6` CSS selector.
- `link`: maps to the `a` CSS selector.

If they’re found in the top-level the element selector will be used. If they’re found within a block, the selector to be used will be the element’s appended to the corresponding block.

Input

Output

```
{
    "version": 3,
    "styles": {
        "typography": {
            "fontSize": "var(--wp--preset--font-size--normal)"
        },
        "elements": {
            "h1": {
```

```
                "typography": {
                    "fontSize": "var(--wp--preset--font-size--huge)"
                }
            },
            "h2": {
                "typography": {
                    "fontSize": "var(--wp--preset--font-size--big)"
                }
            },
            "h3": {
                "typography": {
                    "fontSize": "var(--wp--preset--font-size--medium)"
                }
            }
        },
        "blocks": {
            "core/group": {
                "elements": {
                    "h2": {
                        "typography": {
                            "fontSize": "var(--wp--preset--font-size--small)"
                        }
                    },
                    "h3": {
                        "typography": {
                            "fontSize": "var(--wp--preset--font-size--smaller)"
                        }
                    }
                }
            }
        }
    }
}
```

```
body {
    font-size: var( --wp--preset--font-size--normal );
}
h1 {
    font-size: var( --wp--preset--font-size--huge );
```

```
}
h2 {
    font-size: var( --wp--preset--font-size--big );
}
h3 {
    font-size: var( --wp--preset--font-size--medium );
}
.wp-block-group h2 {
    font-size: var( --wp--preset--font-size--small );
}
.wp-block-group h3 {
    font-size: var( --wp--preset--font-size--smaller );
}
```

Element pseudo selectors

Pseudo selectors are supported for these elements:

- `link` and `button`: `:hover`, `:focus`, `:focus-visible`, `:visited`, `:active`, `:link`, `:any-link`.
- `textInput`: `:required`, `:user-valid`, `:user-invalid`, `:focus`, `:focus-visible`, and the `::placeholder` pseudo-element.
- `select`: `:hover`, `:focus`, `:focus-visible`, `:active`.

```
"elements": {
        "link": {
            "color": {
                "text": "green"
            },
            ":hover": {
                "color": {
                    "text": "hotpink"
                }
            }
        }
    }
```

`:user-valid` and `:user-invalid` only match after a visitor has interacted with the field, so empty required fields aren't marked as invalid when the page loads. `::placeholder` styles the placeholder text, not the field itself:

```
"elements": {
        "textInput": {
            ":user-invalid": {
                "border": {
                    "color": "#d63638"
                }
            },
            "::placeholder": {
                "color": {
                    "text": "#757575"
                }
            }
        }
    }
```

Responsive styles

Block styles can be scoped to two named breakpoints: `@mobile` and `@tablet`. Any style property that is valid at the block or element level can be nested under one of these keys.

Key	Media query applied
`@mobile`	`@media (width <= 480px)`
`@tablet`	`@media (480px < width <= 782px)`

Themes can override the default breakpoint widths with `settings.viewport.mobile` and `settings.viewport.tablet`. Breakpoint values can use `px`, `em`, or `rem` units.

If only one valid breakpoint is configured, the breakpoint keeps its viewport key and uses a single max-width query. For example, a theme with only `settings.viewport.tablet` will expose `@tablet` styles using `@media (width <= value)`.

```
{
    "version": 3,
    "settings": {
        "viewport": {
            "mobile": "640px",
            "tablet": "960px"
        }
    }
}
```

Responsive overrides can be placed directly on a block node:

```
{
    "version": 3,
    "styles": {
        "blocks": {
            "core/group": {
                "color": {
                    "text": "black"
                },
                "@mobile": {
                    "color": {
                        "text": "hotpink"
                    }
                }
            }
        }
    }
}
```

```
:root :where(.wp-block-group) { color: black; }
@media (width <= 480px) { :root :where(.wp-block-group) { color: hotpink; } }
```

A breakpoint can also carry styles for a block’s elements, including their pseudo selectors:

```
{
    "version": 3,
    "styles": {
        "blocks": {
            "core/group": {
                "elements": {
                    "link": {
                        "color": { "text": "blue" },
                        ":hover": {
                            "color": { "text": "navy" }
                        }
                    }
                },
                "@mobile": {
                    "elements": {
```

```
                        "link": {
                            "color": { "text": "red" },
                            ":hover": {
                                "color": { "text": "darkred" }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
:root :where(.wp-block-group a:where(:not(.wp-element-button))) { color: blue; }
@media (width <= 480px) { :root :where(.wp-block-group a:where(:not(.wp-element-button))) {
color: red; } }
:root :where(.wp-block-group a:where(:not(.wp-element-button)):hover) { color: navy; }
@media (width <= 480px) { :root :where(.wp-block-group a:where(:not(.wp-element-
button)):hover) { color: darkred; } }
```

Responsive overrides are always output after the default styles they override, so the cascade order is preserved without needing to increase specificity.

Variations

A block can have a “style variation,” as defined in the block.json specification. Theme authors can define the style attributes for an existing style variation using the `theme.json` file. Styles for unregistered style variations will be ignored.

Note that variations are a “block concept”—they only exist when bound to blocks. The `theme.json` specification respects this distinction by only allowing `variations` at the block level, not the top level. It’s also worth highlighting that only variations defined in the `block.json` file of the block or via `register_block_style` on the server are considered “registered” for `theme.json` styling purposes.

For example, this is how to provide styles for the existing `plain` variation for the `core/quote` block:

```
{
    "version": 3,
    "styles": {
        "blocks": {
            "core/quote": {
```

```
                "variations": {
                    "plain": {
                        "color": {
                            "background": "red"
                        }
                    }
                }
            }
        }
    }
}
```

The resulting CSS output is:

```
.wp-block-quote.is-style-plain {
    background-color: red;
}
```

It is also possible for multiple block types to share the same variation styles. There are two recommended ways to define such shared styles:

1. `theme.json` partial files
2. programmatically, using `register_block_style`

Variation Theme.json Partials

Like theme style variation partials, those for block style variations reside within a theme's `/styles` directory. However, they are differentiated from theme style variations by the introduction of a top-level property called `blockTypes`. The `blockTypes` property is an array of block types for which the block style variation has been registered.

Additionally, a `slug` property is available to provide consistency between the different sources that may define block style variations and to decouple the `slug` from the translatable `title` property.

The following is an example of a `theme.json` partial that defines styles for the "Variation A" block style for the Group, Columns, and Media & Text block types:

```
{
    "$schema": "https://schemas.wp.org/trunk/theme.json",
    "version": 3,
    "title": "Variation A",
    "slug": "variation-a",
    "blockTypes": [ "core/group", "core/columns", "core/media-text" ],
```

```
    "styles": {
        "color": {
            "background": "#eed8d3",
            "text": "#201819"
        },
        "elements": {
            "heading": {
                "color": {
                    "text": "#201819"
                }
            }
        },
        "blocks": {
            "core/group": {
                "color": {
                    "background": "#825f58",
                    "text": "#eed8d3"
                },
                "elements": {
                    "heading": {
                        "color": {
                            "text": "#eed8d3"
                        }
                    }
                }
            }
        }
    }
}
```

Programmatically Registering Variation Styles

As an alternative to `theme.json` partials, you can register variation styles at the same time as registering the variation itself through `register_block_style`. This is done by registering the block style for an array of block types while also passing a “style object” within the `style_data` option.

The example below registers a “Green” variation for the Group and Columns blocks. Note that the style object passed via `style_data` follows the same shape as the `styles` property of a `theme.json` partial.

```
register_block_style(
    array( 'core/group', 'core/columns' ),
    array(
        'name'       => 'green',
        'label'      => __( 'Green' ),
        'style_data' => array(
            'color'    => array(
                'background' => '#4f6f52',
                'text'       => '#d2e3c8',
            ),
            'blocks'   => array(
                'core/group' => array(
                    'color' => array(
                        'background' => '#739072',
                        'text'       => '#e3eedd',
                    ),
                ),
            ),
            'elements' => array(
                'link'   => array(
                    'color'  => array(
                        'text' => '#ead196',
                    ),
                    ':hover' => array(
                        'color' => array(
                            'text' => '#ebd9b4',
                        ),
                    ),
                ),
            ),
        ),
    )
);
```

customTemplates

> Supported in WordPress from version 5.9.

Within this field themes can list the custom templates present in the `templates` folder. For example, for a custom template named `my-custom-template.html`, the `theme.json` can declare what post types can use it and what's the title to show the user:

- name: mandatory.
- title: mandatory, translatable.
- postTypes: optional, only applies to the `page` by default.

```
{
    "version": 3,
    "customTemplates": [
        {
            "name": "my-custom-template",
            "title": "The template title",
            "postTypes": [
                "page",
                "post",
                "my-cpt"
            ]
        }
    ]
}
```

templateParts

> Supported in WordPress from version 5.9.

Within this field themes can list the template parts present in the `parts` folder. For example, for a template part named `my-template-part.html`, the `theme.json` can declare the area term for the template part entity which is responsible for rendering the corresponding block variation (Header block, Footer block, etc.) in the editor. Defining this area term in the json will allow the setting to persist across all uses of that template part entity, as opposed to a block attribute that would only affect one block. Defining area as a block attribute is not recommended as this is only used 'behind the scenes' to aid in bridging the gap between placeholder flows and entity creation.

Currently block variations exist for "header" and "footer" values of the area term, any other values and template parts not defined in the json will default to the general template part block. Variations will be denoted by specific icons within the editor's interface, will default to the corresponding semantic HTML element for the wrapper (this can also be overridden by the `tagName` attribute set on the template part block), and will contextualize the template part allowing more custom flows in future editor improvements.

- name: mandatory.
- title: optional, translatable.
- area: optional, will be set to `uncategorized` by default and trigger no block variation.

```
{
    "version": 3,
    "templateParts": [
        {
            "name": "my-template-part",
            "title": "Header",
            "area": "header"
        }
    ]
}
```

patterns

> Supported in WordPress from version 6.0.

Within this field themes can list patterns to register from Pattern Directory. The `patterns` field is an array of pattern `slugs` from the Pattern Directory. Pattern slugs can be extracted by the `url` in single pattern view at the Pattern Directory. For example in this url `https://wordpress.org/patterns/pattern/partner-logos` the slug is` partner-logos`.

```
{
    "version": 3,
    "patterns": [ "short-text-surrounded-by-round-images", "partner-logos" ]
}
```

Developing with theme.json

It can be difficult to remember the theme.json settings and properties and which versions of WordPress support which settings while you develop, so it can be helpful to use the provided JSON schema for theme.json.

Many code editors support JSON schema and can provide help like tooltips, autocomplete, or schema validation right in your editor.

Theme.json schemas for each WordPress version are available at `https://schemas.wp.org/wp/{{version}}/theme.json`. For example a schema for WordPress 5.8 is available at `https://schemas.wp.org/wp/5.8/theme.json`. To ensure that you're only using features available to your users, it's best to use the oldest version that your theme supports.

The latest schema including all the latest changes from the Gutenberg plugin is available at `https://schemas.wp.org/trunk/theme.json`.

Check your editor's documentation for JSON schema support. In Visual Studio Code, for example, you need to add `"$schema": "https://schemas.wp.org/wp/x.x/theme.json"` as a top-level property of your theme.json file, but other editors may be configured differently.

Example using validation with schema

Frequently Asked Questions

The naming schema of CSS Custom Properties

One thing you may have noticed is the naming schema used for the CSS Custom Properties the system creates, including the use of double hyphen, `--`, to separate the different "concepts". Take the following examples.

Presets such as `--wp--preset--color--black` can be divided into the following chunks:

- `--wp`: prefix to namespace the CSS variable.
- `preset`: indicates that this is a CSS variable that belongs to the presets.
- `color`: indicates which preset category the variable belongs to. It can be `color`, `font-size`, `gradients`.
- `black`: the `slug` of the particular preset value.

Custom properties such as `--wp--custom--line-height--body`, which can be divided into the following chunks:

- `--wp`: prefix to namespace the CSS variable.
- `custom`: indicates that this is a "free-form" CSS variable created by the theme.
- `line-height--body`: the result of converting the "custom" object keys into a string.

The `--` as a separator has two functions:

- Readability, for human understanding. It can be thought as similar to the BEM naming schema, it separates "categories".

- Parsability, for machine understanding. Using a defined structure allows machines to understand the meaning of the property `--wp--preset--color--black`: it's a value bounded to the color preset whose slug is "black", which then gives us room to do more things with them.

Why using `--` as a separator?

We could have used any other separator, such as a single `-`.

However, that'd have been problematic, as it'd have been impossible to tell how `--wp-custom-line-height-template-header` should be converted back into an object, unless we force theme authors not to use `-` in their variable names.

By reserving `--` as a category separator and let theme authors use `-` for word-boundaries, the naming is clearer: `--wp--custom--line-height--template-header`.

How settings under "custom" create new CSS Custom Properties

The algorithm to create CSS Variables out of the settings under the "custom" key works this way:

This is for clarity, but also because we want a mechanism to parse back a variable name such `--wp--custom--line-height--body` to its object form in theme.json. We use the same separation for presets.

For example:

Input

Output

```
{
    "version": 3,
    "settings": {
        "custom": {
            "lineHeight": {
                "body": 1.7
            },
            "font-primary": "-apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen-
Sans, Ubuntu, Cantarell, 'Helvetica Neue', sans-serif"
```

```
        }
    }
}
```

```
body {
    --wp--custom--line-height--body: 1.7;
    --wp--custom--font-primary: "-apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
Oxygen-Sans, Ubuntu, Cantarell, 'Helvetica Neue', sans-serif";
}
```

A few notes about this process:

- `camelCased` keys are transformed into its `kebab-case` form, as to follow the CSS property naming schema. Example: `lineHeight` is transformed into `line-height`.
- Keys at different depth levels are separated by `--`. That's why `line-height` and `body` are separated by `--`.
- You shouldn't use `--` in the names of the keys within the `custom` object. Example, **don't do** this:

```
{
    "version": 3,
    "settings": {
        "custom": {
            "line--height": { // DO NOT DO THIS
                "body": 1.7
            }
        }
    }
}
```

Global Stylesheet

In WordPress 5.8, the CSS for some of the presets defined by WordPress (font sizes, colors, and gradients) was loaded twice for most themes: in the block-library stylesheet plus in the global stylesheet. Additionally, there were slight differences in the CSS in both places.

In WordPress 5.9 release, CSS of presets are consolidated into the global stylesheet, that is now loaded for all themes. Each preset value generates a single CSS Custom Property and a class, as in:

```
/* CSS Custom Properties for the preset values */
body {
```

```
  --wp--preset--<PRESET_TYPE>--<PRESET_SLUG>: <DEFAULT_VALUE>;
  --wp--preset--color--pale-pink: #f78da7;
  --wp--preset--font-size--large: 36px;
  /* etc. */
}

/* CSS classes for the preset values */
.has-<PRESET_SLUG>-<PRESET_TYPE> { ... }
.has-pale-pink-color { color: var(--wp--preset--color--pale-pink) !important; }
.has-large-font-size { font-size: var(--wp--preset--font-size--large) !important; }
```

For themes to override the default values they can use the `theme.json` and provide the same slug. Themes that do not use a `theme.json` can still override the default values by enqueuing some CSS that sets the corresponding CSS Custom Property.

`Example` (sets a new value for the default large font size):

```
body {
 --wp--preset--font-size--large: <NEW_VALUE>;
}
```

Specificity for link colors provided by the user

In WordPress 5.8, when a user selected a link color for a specific block we attached a class to that block in the form of `.wp-element-<ID>` and then enqueued the following style:

```
.wp-element-<ID> a { color: <USER_COLOR_VALUE> !important; }
```

While this preserved user preferences at all times, the specificity was too strong and conflicted with some blocks with legit uses of an HTML element that shouldn't be considered links. To address this issue, in WordPress 5.9 release, the `!important` was removed and updated the corresponding blocks to style the a elements with a specificity higher than the user link color, which now is:

```
.wp-element-<ID> a { color: <USER_COLOR_VALUE>; }
```

As a result of this change, it's now the block author and theme author's responsibility to make sure the user choices are respected at all times and that the link color provided by the user (specificity 011) is not overridden.

What is blockGap and how can I use it?

For blocks that contain inner blocks, such as Group, Columns, Buttons, and Social Icons, `blockGap` controls the spacing between inner blocks. For `blockGap` to work, the block must also opt in to the `layout` block support, which provides layout styles that can be adjusted via the block spacing controls. Depending on the layout of the block, the `blockGap` value will be output as either a vertical margin or a `gap` value. In the editor, the control for the `blockGap` value is called *Block spacing*, located in the Dimensions panel.

```
{
    "version": 3,
    "settings": {
        "spacing": {
            "blockGap": true,
        }
    },
    "styles": {
        "spacing": {
            "blockGap": "1.5rem"
        }
    }
}
```

The setting for `blockGap` is either a boolean or `null` value and is `null` by default. This allows an extra level of control over style output. The `settings.spacing.blockGap` setting in a `theme.json` file accepts the following values:

- `true`: Opt into displaying *Block spacing* controls in the editor UI and output `blockGap` styles.
- `false`: Opt out of displaying *Block spacing* controls in the editor UI, with `blockGap` styles stored in `theme.json` still being rendered. This allows themes to use `blockGap` values without allowing users to make changes within the editor.
- `null` (default): Opt out of displaying *Block spacing* controls, *and* prevent the output of `blockGap` styles.

The value defined for the root `styles.spacing.blockGap` style is also output as a CSS property, named `--wp--style--block-gap`.

Why does it take so long to update the styles in the browser?

When you are actively developing with theme.json you may notice it takes 30+ seconds for your changes to show up in the browser, this is because `theme.json` is cached. To remove this caching issue, set either `WP_DEBUG` or `SCRIPT_DEBUG` to ‘true’ in your `wp-config.php`. This tells WordPress to skip the cache and always use fresh data.

□□□ #2
□ □□□ □□□ 2026-10-31 18:26:27 UTC
□ □□□ □□□ 2026-10-31 18:26:30 UTC