@wordpress/theme

- Documentation
- Public API
- Design Tokens
  - Using Design Tokens
    - Delivery model
    - Within WordPress
    - Outside WordPress
  - Developer Tools
  - Accessibility
- Theme Provider
  - TypeScript props and warnings
  - Cursor and corner radius
  - Light and dark themes
  - Nesting Providers
  - isRoot and the containing document
  - Across documents (iframes and other portals)
  - Legacy compatibility
  - Building
- Contributing to this package

A theming package that's part of the WordPress Design System. It has two parts:

- **Design Tokens**: A comprehensive system of design tokens for colors, spacing, typography, and more.
- **Theme System**: A flexible theming provider for consistent theming across applications.

This package is not a WordPress block theme, site theme, or `theme.json` API. It provides the WordPress Design System's design tokens and React theming primitives for JavaScript packages and applications.

Documentation

This README is the entry point for package consumers. It covers how to load design tokens and use `ThemeProvider`.

- To use design tokens and `ThemeProvider`, start here.
- To pick the right design token or browse every available token, see the generated Design Tokens Reference.
- To edit token source files, see the Design Tokens Maintainer's Guide.
- To validate token usage or inject fallback values at build time, see `@wordpress/theme-build-tools`.

Public API

Entrypoint	Supported use
`@wordpress/theme`	`ThemeProvider` and the generated token scale types, such as `PaddingSize` and `GapSize`. Derive provider props and callback types from the component as shown below.
`@wordpress/theme/design-tokens.css`	Default semantic `--wpds-*` custom properties. Load once per document.
`@wordpress/theme/design-tokens.js`	Default token-name array, plus `tokens` metadata and `groups` named exports. Does not load styles.

The `privateApis` export exists for temporary compatibility with older WordPress bundles. It is not a supported consumer API. Token source JSON, ramp builders, generated fallback maps, and other paths not listed in the package's `exports` are implementation details.

Runtime APIs supplied by WordPress follow its backward compatibility policy. Installing a newer npm package does not upgrade the runtime supplied by WordPress. Check the target WordPress version before using a runtime API. Tooling entrypoints such as `@wordpress/theme/design-tokens.js` and `@wordpress/theme-build-tools` use the installed npm packages. Removing or renaming semantic tokens is a compatibility change; meaningful value changes belong in the changelog.

To inspect token names in development tooling:

```
import { tokens } from '@wordpress/theme/design-tokens.js';


const tokenNames = Object.keys( tokens );
const isKnownToken = tokenNames.includes( '--wpds-dimension-gap-sm' );
```

To inspect token metadata and default values:

```
import { tokens, groups } from '@wordpress/theme/design-tokens.js';

const brand = tokens[ '--wpds-color-background-interactive-brand-strong' ];
// brand.modes.default.css: '#3858e9'
// brand.fallback: 'var(--wp-admin-theme-color, #3858e9)'

const radiusTokenNames = groups.border.groups.radius.tokens;
const fontFamilyTokenNames = groups.typography.groups[ 'font-family' ].tokens;
```

`tokens` is keyed by semantic CSS custom property name. Each entry has `$description`, `$type`, `group`, `fallback`, and `modes.default`, containing the resolved DTCG `$value` and CSS string `css`. `group` identifies the top-level namespace, such as `border`, `color`, or `typography`.

`groups` preserves the source token hierarchy, keyed by top-level namespace. Each group has a `tokens` array of CSS custom property names belonging directly to it and a `groups` object containing its child groups. These are empty when the group has no direct tokens or children, respectively. A group that has a `$description` in the token source also exposes it as `$description`, for example `groups.dimension.groups.gap.$description`.

`modes.default` contains build-time defaults and does not reflect theme overrides. Use CSS custom properties to follow the current theme. `fallback` is the expression used by the build plugins when a token variable is unavailable and can differ from the default value, as shown above.

Literal token types are preserved in TypeScript declarations, allowing narrowing by `$type`.

To build a fallback map for a custom plugin:

```
import { tokens } from '@wordpress/theme/design-tokens.js';

const tokenFallbacks = Object.fromEntries(
    Object.entries( tokens ).map( ( [ name, token ] ) => [
        name,
        token.fallback,
    ] )
);
```

Design Tokens

Design tokens are named values that describe the visual purpose of a value. Rather than hardcoding values like `#3858e9` or `16px`, use semantic custom properties like `--wpds-color-background-interactive-brand-strong` or `--wpds-dimension-padding-2xl`.

The **Design Tokens Reference** explains the naming pattern, how to choose a token, and the complete generated list of available design tokens.

Using Design Tokens

Design tokens are delivered as CSS custom properties (e.g. `var(--wpds-color-foreground-content-neutral)`). To use them, a stylesheet defining the token values must be loaded on the page.

The `ThemeProvider` component can be used to customize token values like colors for a specific part of your application.

Delivery model

The design system splits token delivery into two complementary layers:

- **Static stylesheet (`design-tokens.css`)** — defines the default value for every `--wpds-*` custom property at the document `:root`. Loaded once per document (the main page, *and* each iframe you render React into). Provides a working baseline even before any JavaScript runs.
- **Runtime `<ThemeProvider>`** — applies per-instance overrides for a subtree, on top of the static defaults. Use it to override individual settings (e.g. `color.primary`, `cursor.control`).

Within WordPress

Stylesheets are managed on your behalf in standard WordPress admin and editor screens. WordPress registers the design tokens stylesheet as the `wp-theme` style handle and loads it where WordPress admin and editor packages already depend on the shared base styles. This includes the admin page and the block editor's content iframe.

If your plugin renders a separate app shell, iframe, popup window, or package bundle outside those WordPress-managed style dependencies, enqueue the `wp-theme` stylesheet in that document instead of bundling `@wordpress/theme/design-tokens.css` into your plugin. See the wp_enqueue_style documentation for how to specify stylesheet dependencies.

Outside WordPress

Outside of WordPress, install and load the design tokens stylesheet to support the full range of theming capabilities:

```
npm install @wordpress/theme
```

```
import '@wordpress/theme/design-tokens.css';
```

The package's JavaScript entrypoints are ESM-only and require Node.js `>=22.13.0`. Use `import` syntax from ESM or TypeScript configuration files.

TypeScript consumers require TypeScript 5 or newer.

This stylesheet is universal and does not have a separate RTL version.

If your application renders React content into additional documents (an iframe, a popup window, etc.), each of those documents needs the same stylesheet loaded in its own `<head>`. See Across documents (iframes and other portals).

Developer Tools

The `@wordpress/theme-build-tools` package provides Stylelint plugins to validate token usage and build plugins to inject generated fallback values. `@wordpress/build` applies the build plugins automatically.

Accessibility

The semantic color tokens are designed so the default foreground/background pairs used by the design system meet WCAG AA text contrast. Components still need to choose the correct semantic pair for the UI state they render, and must not rely on color alone to convey information.

Design tokens do not replace component-level accessibility handling:

- **Forced colors:** components are still responsible for `forced-colors` overrides where native high-contrast rendering would otherwise hide borders, icons, focus rings, or state indicators.
- **Reduced motion:** motion tokens provide shared durations and easing curves, but consuming components are still responsible for respecting `prefers-reduced-motion` for non-essential animations.

For example, define non-essential transitions only when the user has not requested reduced motion:

```
@media not ( prefers-reduced-motion ) {
    .example {
```

```
        transition-property: opacity;
        transition-duration: var( --wpds-motion-duration-md );
        transition-timing-function: var( --wpds-motion-easing-subtle );
    }
}
```

Theme Provider

The `ThemeProvider` is a React component that should wrap your application to provide design tokens and theme context to the child UI components. It accepts a set of customizable seed values and automatically generates a set of design tokens, which are exposed as CSS custom properties for use throughout the application.

```
import { ThemeProvider } from '@wordpress/theme';

function App() {
    return (
        <ThemeProvider color={ { primary: 'blue' } }>
            { /* Your app content */ }
        </ThemeProvider>
    );
}
```

When a setting is omitted, it inherits from the closest parent `ThemeProvider`. If there is no parent value, the prebuilt defaults from the design-tokens stylesheet apply.

`ThemeProvider` does not accept wrapper customization props such as `className`, `style`, `as`, `render`, or `ref`.

The `color` prop accepts an object with the following optional properties:

- `primary`: The primary/accent seed color (default: `'#3858e9'`).
- `background`: The background seed color (default: `'#fcfcfc'`).

Both properties accept a fully opaque sRGB-parseable string: a hex value (e.g. `#3858e9`), an `rgb()`/`rgba()` string, or a CSS named color (e.g. `'blue'`). Non-opaque alpha values, `transparent`, and other CSS color spaces (e.g. `hsl()`, `oklch()`, `lab()`) are not accepted and will throw an error.

The seed values guide the generated color ramps but are not guaranteed to appear unchanged in the resulting tokens. The ramp builder may adjust their lightness to meet its contrast targets.

Use `onColorWarnings` to receive structured warnings after the provider calculates its colors. Warnings identify generated ramp steps or semantic foreground/background pairs that do not meet their contrast targets:

```
<ThemeProvider
    color={ {
        primary: '#ff0000',
        background: '#0000ff',
    } }
    onColorWarnings={ ( warnings ) => {
        // Format or display the warnings for your users.
    } }
>
    { /* Your app content */ }
</ThemeProvider>
```

The callback reports failures from the generated ramp checks and a defined set of semantic foreground/background pairs. It does not validate every possible token pairing. It receives an empty array when all checked targets are met. Ramp warnings identify the affected ramp and step. Contrast warnings identify the semantic foreground/background token pair and include the required and achieved contrast values. React may invoke the callback more than once in development under Strict Mode.

The contrast warnings also cover the focus-ring color and the resting and active strong brand fills against all three neutral surfaces and the brand surface at a 3:1 minimum. They use the final generated colors after seed adjustments. These checks report failures without adjusting the colors. Text pairs retain their 4.5:1 minimum.

For example, with the default primary `#3858e9` and background `#1e1e1e`, the resting strong brand fill has 2.97:1 contrast against `background.surface.neutral` and 2.76:1 against `background.surface.neutral-strong`. The focus ring, active fill, and text pairs still meet their targets. Choose a different primary or background seed to resolve the reported pair; for example, the default primary with background `#000000` meets these checked contrast targets.

TypeScript props and warnings

Derive types from the public component instead of importing package internals:

```
import type { ComponentProps } from 'react';
import { ThemeProvider } from '@wordpress/theme';

type ThemeProviderProps = ComponentProps< typeof ThemeProvider >;
```

```
type OnColorWarnings = NonNullable< ThemeProviderProps[ 'onColorWarnings' ] >;
type ThemeProviderColorWarning = Parameters< OnColorWarnings >[ 0 ][ number ];

const onColorWarnings: OnColorWarnings = ( warnings ) => {
    for ( const warning of warnings ) {
        if ( warning.type === 'contrast' ) {
            console.log( warning.foregroundToken, warning.achievedContrast );
        } else {
            console.log( warning.ramp, warning.step );
        }
    }
};
```

The derived warning type is a union. Check `type` before accessing the fields specific to ramp or contrast warnings. Derive individual settings in the same way, for example `ThemeProviderProps[ 'cornerRadius' ]`.

Cursor and corner radius

The `cursor` prop accepts an object with the following optional properties:

- `control`: The cursor style for interactive controls that are not links (e.g. buttons, checkboxes, and toggles). Accepts `'default'` or `'pointer'` (default: `'pointer'`).

The `cornerRadius` prop sets the overall roundness preset for the theme subtree. Accepts `'none'` (square corners), `'subtle'`, `'moderate'`, or `'pronounced'` (most rounded) (default: `'subtle'`). This scales the primitive `--wpds-border-radius-*` tokens for the provider subtree. The preset sets the overall amount of roundness, not an individual border-radius token size.

Light and dark themes

`ThemeProvider` has no light or dark mode. It accepts color seeds and turns them into semantic design tokens. Consumers define light and dark themes by choosing those seeds: a light background seed makes the generated color ramp favor darker contrasting colors, while a dark background seed makes it favor lighter contrasting colors. The primary seed controls the accent color rather than the overall light or dark appearance.

Nesting Providers

The provider can be used recursively to override or modify the theme for a specific subtree.

```
<ThemeProvider color={ { background: 'white' } }>
    { /* light-themed UI components */ }
    <ThemeProvider color={ { background: '#1e1e1e' } }>
        { /* dark-themed UI components */ }
        <ThemeProvider color={ { primary: 'red' } }>
            { /* dark-themed with red accent */ }
        </ThemeProvider>
    </ThemeProvider>
    { /* light-themed UI components */ }
</ThemeProvider>
```

The `ThemeProvider` redefines some of the design system tokens. Components consuming semantic design system tokens will automatically follow the chosen theme. Note that the tokens are defined and inherited using the CSS cascade, and therefore the DOM tree, not the React tree. This is very important when using React portals.

`isRoot` and the containing document

By default, the styles a `<ThemeProvider>` emits are scoped to the provider's wrapper `<div>`, so overrides apply only to its subtree.

Setting `isRoot` additionally hoists those overrides to the containing document's `:root`, so anything rendered into that document — including overlays portalled outside the provider's React tree — picks them up.

```
<ThemeProvider color={ { primary: '#a00' } } isRoot>
    { /* …app… */ }
</ThemeProvider>
```

Use `isRoot` on the top-level provider for an application or page. It's also the recommended pattern for the topmost provider rendered into a separate document (iframe, popup window).

Render at most one root provider per document. Multiple `isRoot` providers that share the same document are unsupported because each one would try to define the document-level token values. Nested and sibling providers can still be used normally when `isRoot` is omitted, and separate documents can each have their own root provider.

The static design-tokens stylesheet still provides the default values; `isRoot` is only needed when you want a `<ThemeProvider>`'s overrides to reach the whole document.

Across documents (iframes and other portals)

When you render React content into a different document (typically an iframe), two things must be true for design tokens to work correctly in that document:

1. **The design-tokens stylesheet is present in the document's `<head>`.** This is the static `:root` block that defines every `--wpds-*` custom property.
   Inside WordPress, this is enqueued automatically for both the admin page and the block editor's content iframe.
   For custom iframes, the consumer is responsible for loading it in the iframe document. Within WordPress, enqueue the `wp-theme` stylesheet for that document. Outside WordPress, import `@wordpress/theme/design-tokens.css` from a stylesheet that the iframe already loads, or inject the CSS string directly.
2. **CSS Module styles are registered in the iframe document.** Styles from CSS Modules built with `@wordpress/build` are injected at runtime through `@wordpress/style-runtime`. Wrap the iframe subtree in `<StyleProvider document={ iframeDocument }>` so registered styles reach that document's `<head>` before children measure layout.

The canonical pattern combines both with a `<ThemeProvider isRoot>` to apply any overrides to the iframe's `:root`:

```
import { __experimentalStyleProvider as StyleProvider } from '@wordpress/components';
import { ThemeProvider } from '@wordpress/theme';
import { createPortal } from 'react-dom';

function IframeContent( { iframeDocument, children } ) {
    return createPortal(
        <StyleProvider document={ iframeDocument }>
            <ThemeProvider isRoot color={ { primary: '#a00' } }>
                { children }
            </ThemeProvider>
        </StyleProvider>,
        iframeDocument.body
    );
}
```

The static stylesheet inside the iframe provides every default; `<ThemeProvider isRoot>` adds (or omits) overrides on top, exactly like in the main document.

Legacy compatibility

The public token surface is the semantic `--wpds-*` custom properties documented in the Design Tokens Reference.

`@wordpress/theme` may also maintain legacy compatibility aliases for existing WordPress admin and `@wordpress/components` internals. Those aliases are transitional implementation details, including the `--wp-components-*` namespace, and are not a supported API for consumers. New code should use semantic `--wpds-*` design tokens instead.

Building

This package is built in two steps. When `npm run build` is run at the root of the repo, it will first run the “prebuild” step of this package, which is defined in the `build` script of this package’s package.json.

This step will:

1. Generate default color ramps.
2. Generate primitive tokens.
3. Build CSS and JavaScript token files.
4. Update the design tokens documentation.
5. Generate complete Figma border collection mode files in `prebuilt/figma`.
6. Format all generated files.

The files generated in this step will all be committed to the repo.

After the prebuild step, the package will be built into its final form via the repo’s standard package build script.

Contributing to this package

This is an individual package that’s part of the Gutenberg project. The project is organized as a monorepo. It’s made up of multiple self-contained software packages, each with a specific purpose.

The packages in this monorepo are published to npm and used by WordPress as well as other software projects.

To find out more about contributing to this package or Gutenberg as a whole, please read the project’s main contributor guide.

□□□ #2
□ □□□ □□□ 2026-10-31 18:21:50 UTC

□ □□□ □□□ 2026-10-31 18:21:52 UTC