VitePress

Source	https://vitepress.dev
Exported	2026-10-14
Pages	4
Platform	vitepress

Table of Contents

What is VitePress? 3
- Use Cases 3
- Developer Experience 3
- Performance 4
- What About VuePress? 4

Getting Started 5
- Try It Online 5
- Installation 5
- File Structure 7
- Up and Running 8
- What's Next? 9

Markdown Extensions 10
- Header Anchors 10
- Links 10
- Frontmatter 11
- GitHub-Style Tables 11
- Task Lists 12
- Footnotes 12
- Emoji 12
- Table of Contents 12
- Custom Containers 13
- GitHub-flavored Alerts 17
- Syntax Highlighting in Code Blocks 18
- Line Highlighting in Code Blocks 19
- Focus in Code Blocks 21
- Colored Diffs in Code Blocks 21
- Errors and Warnings in Code Blocks 22
- Line Numbers 22
- Import Code Snippets 23
- Code Groups 25
- Markdown File Inclusion 27
- Math Equations 30
- Image Lazy Loading 31
- Advanced Configuration 32

Extending the Default Theme 33
- Customizing CSS 33

Using Different Fonts 34
Registering Global Components 35
Layout Slots 36
Using View Transitions API 38
Overriding Internal Components 40

What is VitePress?

VitePress is a Static Site Generator (SSG) designed for building fast, content-centric websites. In a nutshell, VitePress takes your source content written in Markdown, applies a theme to it, and generates static HTML pages that can be easily deployed anywhere.

> **TIP**
>
> Just want to try it out? Skip to the Quickstart.

Use Cases

- **Documentation**

  VitePress ships with a default theme designed for technical documentation. It powers this page you are reading right now, along with the documentation for Vite, Rollup, Pinia, VueUse, Vitest, D3, UnoCSS, Iconify and many more.

  The official Vue.js documentation is also based on VitePress, but uses a custom theme shared between multiple translations.

- **Blogs, Portfolios, and Marketing Sites**

  VitePress supports fully customized themes, with the developer experience of a standard Vite + Vue application. Being built on Vite also means you can directly leverage Vite plugins from its rich ecosystem. In addition, VitePress provides flexible APIs to load data (local or remote) and dynamically generate routes. You can use it to build almost anything as long as the data can be determined at build time.

  The official Vue.js blog is a simple blog that generates its index page based on local content.

Developer Experience

VitePress aims to provide a great Developer Experience (DX) when working with Markdown content.

- **Vite-Powered:** instant server start, with edits always instantly reflected (<100ms) without page reload.
- **Built-in Markdown Extensions:** Frontmatter, tables, syntax highlighting... you name it. Specifically, VitePress provides many advanced features for working with code blocks, making it ideal for highly technical documentation.
- **Vue-Enhanced Markdown:** each Markdown page is also a Vue Single-File Component, thanks to Vue template's 100% syntax compatibility with HTML. You can embed interactivity in your static content using Vue templating features or imported Vue components.

Performance

Unlike many traditional SSGs where each navigation results in a full page reload, a website generated by VitePress serves static HTML on the initial visit, but becomes a Single Page Application (SPA) for subsequent navigation within the site. This model, in our opinion, provides an optimal balance for performance:

- **Fast Initial Load**

  The initial visit to any page will be served the static, pre-rendered HTML for fast loading speed and optimal SEO. The page then loads a JavaScript bundle that turns the page into a Vue SPA ("hydration"). Contrary to common assumptions of SPA hydration being slow, this process is actually extremely fast thanks to Vue 3's raw performance and compiler optimizations. On PageSpeed Insights, typical VitePress sites achieve near-perfect performance scores even on low-end mobile devices with a slow network.

- **Fast Post-load Navigation**

  More importantly, the SPA model leads to better user experience **after** the initial load. Subsequent navigation within the site will no longer cause a full page reload. Instead, the incoming page's content will be fetched and dynamically updated. VitePress also automatically pre-fetches page chunks for links that are within viewport. In most cases, post-load navigation will feel instant.

- **Interactivity Without Penalty**

  To be able to hydrate the dynamic Vue parts embedded inside static Markdown, each Markdown page is processed as a Vue component and compiled into JavaScript. This may sound inefficient, but the Vue compiler is smart enough to separate the static and dynamic parts, minimizing both the hydration cost and payload size. For the initial page load, the static parts are automatically eliminated from the JavaScript payload and skipped during hydration.

What About VuePress?

VitePress is the spiritual successor of VuePress 1. The original VuePress 1 was based on Vue 2 and webpack. With Vue 3 and Vite under the hood, VitePress provides significantly better DX, better production performance, a more polished default theme, and a more flexible customization API.

The API difference between VitePress and VuePress 1 mostly lies in theming and customization. If you are using VuePress 1 with the default theme, it should be relatively straightforward to migrate to VitePress.

Maintaining two SSGs in parallel isn't sustainable, so the Vue team has decided to focus on VitePress as the main recommended SSG in the long run. Now VuePress 1 has been deprecated, and VuePress 2 has been handed over to the VuePress community team for further development and maintenance.

Getting Started

Try It Online

You can try VitePress directly in your browser on StackBlitz.

Installation

Prerequisites

- Node.js version 22 or higher.
- Terminal for accessing VitePress via its command line interface (CLI).
- Text Editor with Markdown syntax support.
  - VSCode is recommended, along with the official Vue extension.

VitePress can be used on its own, or be installed into an existing project. In both cases, you can install it with:

```
$ npm add -D vitepress@next
```

```
$ pnpm add -D vitepress@next
```

```
$ yarn add -D vitepress@next vue
```

```
$ bun add -D vitepress@next
```

```
$ deno add -D vitepress@next
```

> **NOTE**
>
> VitePress is an ESM-only package. Don't use `require()` to import it, and make sure your nearest `package.json` contains `"type": "module"`, or change the file extension of your relevant files like `.vitepress/config.js` to `.mjs`/`.mts`. Refer to Vite's troubleshooting guide for more details. Also, inside async CJS contexts, you can use `await import('vitepress')` instead.

Setup Wizard

VitePress ships with a command line setup wizard that will help you scaffold a basic project. After installation, start the wizard by running:

```
$ npx vitepress init
```

```
$ pnpm vitepress init
```

```
$ yarn vitepress init
```

```
$ bun vitepress init
```

You will be greeted with a few simple questions:

init.ansi

```
+  Welcome to VitePress!
|
?  Where should VitePress initialize the config?
|  ./docs
|
?  Where should VitePress look for your markdown files?
|  ./docs
|
?  Site title:
|  My Awesome Project
|
?  Site description:
|  A VitePress Site
|
?  Theme:
|  Default Theme
|
?  Use TypeScript for config and theme files?
|  Yes
|
?  Add VitePress npm scripts to package.json?
|  Yes
|
?  Add a prefix for VitePress npm scripts?
|  Yes
|
?  Prefix for VitePress npm scripts:
|  docs
|
+  Done! Now run pnpm run docs:dev and start writing.
```

> **VUE AS PEER DEPENDENCY**
>
> If you intend to perform customization that uses Vue components or APIs, you should also explicitly install `vue` as a dependency.

File Structure

If you are building a standalone VitePress site, you can scaffold the site in your current directory (`./`). However, if you are installing VitePress in an existing project alongside other source code, it is recommended to scaffold the site in a nested directory (e.g. `./docs`) so that it is separate from the rest of the project.

Assuming you chose to scaffold the VitePress project in `./docs`, the generated file structure should look like this:

```
.
+- docs
|  +- .vitepress
|  |  +- config.js
|  +- api-examples.md
|  +- markdown-examples.md
|  +- index.md
+- package.json
```

The `docs` directory is considered the **project root** of the VitePress site. The `.vitepress` directory is a reserved location for VitePress' config file, dev server cache, build output, and optional theme customization code.

> **TIP**
>
> By default, VitePress stores its dev server cache in `.vitepress/cache`, and the production build output in `.vitepress/dist`. If using Git, you should add them to your `.gitignore` file. These locations can also be configured.

The Config File

The config file (`.vitepress/config.js`) allows you to customize various aspects of your VitePress site, with the most basic options being the title and description of the site:

.vitepress/config.js

```
export default {
  // site-level options
  title: 'VitePress',
  description: 'Just playing around.',

  themeConfig: {
    // theme-level options
  }
}
```

You can also configure the behavior of the theme via the `themeConfig` option. Consult the Config Reference for full details on all config options.

Source Files

Markdown files outside the `.vitepress` directory are considered **source files**.

VitePress uses **file-based routing**: each `.md` file is compiled into a corresponding `.html` file with the same path. For example, `index.md` will be compiled into `index.html`, and can be visited at the root path `/` of the resulting VitePress site.

VitePress also provides the ability to generate clean URLs, rewrite paths, and dynamically generate pages. These will be covered in the Routing Guide.

Up and Running

The tool should have also injected the following npm scripts to your `package.json` if you allowed it to do so during the setup process:

package.json

```
{
  ...
  "scripts": {
    "docs:dev": "vitepress dev docs",
    "docs:build": "vitepress build docs",
    "docs:preview": "vitepress preview docs"
  },
  ...
}
```

The `docs:dev` script will start a local dev server with instant hot updates. Run it with the following command:

```
$ npm run docs:dev
```

```
$ pnpm run docs:dev
```

```
$ yarn docs:dev
```

```
$ bun run docs:dev
```

Instead of npm scripts, you can also invoke VitePress directly with:

```
$ npx vitepress dev docs
```

```
$ pnpm vitepress dev docs
```

```
$ yarn vitepress dev docs
```

```
$ bun vitepress dev docs
```

More command line usage is documented in the CLI Reference.

The dev server should be running at `http://localhost:5173`. Visit the URL in your browser to see your new site in action!

What's Next?

- To better understand how markdown files are mapped to generated HTML, proceed to the Routing Guide.
- To discover more about what you can do on the page, such as writing markdown content or using Vue Components, refer to the "Writing" section of the guide. A great place to start would be to learn about Markdown Extensions.
- To explore the features provided by the default documentation theme, check out the Default Theme Config Reference.
- If you want to further customize the appearance of your site, explore how to either Extend the Default Theme or Build a Custom Theme.
- Once your documentation site takes shape, make sure to read the Deployment Guide.

Markdown Extensions

VitePress comes with built in Markdown Extensions.

Header Anchors

Headers automatically get anchor links applied. Rendering of anchors can be configured using the `markdown.anchor` option.

Custom anchors

To specify a custom anchor tag for a heading instead of using the auto-generated one, add a suffix to the heading:

```
# Using custom anchors {#my-anchor}
```

This allows you to link to the heading as `#my-anchor` instead of the default `#using-custom-anchors`.

Links

Both internal and external links get special treatment.

Internal Links

Internal links are converted to router links for SPA navigation. Also, every `index.md` contained in each sub-directory will automatically be converted to `index.html`, with corresponding URL `/`.

For example, given the following directory structure:

```
.
+- index.md
+- foo
|  +- index.md
|  +- one.md
|  +- two.md
+- bar
   +- index.md
   +- three.md
   +- four.md
```

And providing you are in `foo/one.md`:

```
[Home](/) <!-- sends the user to the root index.md -->
[foo](/foo/) <!-- sends the user to index.html of directory foo -->
[foo heading](./#heading) <!-- anchors user to a heading in the foo index file -->
[bar - three](../bar/three) <!-- you can omit extension -->
[bar - three](../bar/three.md) <!-- you can append .md -->
[bar - four](../bar/four.html) <!-- or you can append .html -->
```

Page Suffix

Pages and internal links get generated with the `.html` suffix by default.

External Links

Outbound links automatically get `target="_blank" rel="noreferrer"`:

- vuejs.org
- VitePress on GitHub

Frontmatter

YAML frontmatter is supported out of the box:

```
---
title: Blogging Like a Hacker
lang: en-US
---
```

This data will be available to the rest of the page, along with all custom and theming components.

For more details, see Frontmatter.

GitHub-Style Tables

Input

```
| Tables        |      Are      |  Cool |
| ------------- | :-----------: | ----: |
| col 3 is      | right-aligned | $1600 |
| col 2 is      |   centered    |   $12 |
| zebra stripes |   are neat    |    $1 |
```

Output

Tables	Are	Cool
col 3 is	right-aligned	$1600
col 2 is	centered	$12

Tables	Are	Cool
zebra stripes	are neat	$1

Task Lists

Input

```
- [ ] Write the press release
- [x] Update the website
```

Output

- Write the press release
- Update the website

Footnotes

Input

```
Footnotes are supported[^1], including inline ones^[This is an inline footnote.].

[^1]: Definitions can contain **markdown** and are rendered at the end of the page.
```

Output

Footnotes are supported[1], including inline ones[2].

Emoji

Input

```
:tada: :100:
```

Output

A list of all emojis is available.

Table of Contents

Input

```
[[toc]]
```

Output

Rendering of the TOC can be configured using the `markdown.toc` option.

Custom Containers

Custom containers can be defined by their types, titles, and contents.

Default Title

Input

```
::: info
This is an info box.
:::

::: tip
This is a tip.
:::

::: warning
This is a warning.
:::

::: danger
This is a dangerous warning.
:::

::: details
This is a details block.
:::
```

Output

> **INFO**
>
> This is an info box.

> **TIP**
>
> This is a tip.

> **WARNING**
>
> This is a warning.

> **DANGER**
>
> This is a dangerous warning.

Details

This is a details block.

Custom Title

You may set custom title by appending the text right after the "type" of the container.

Input

````
::: danger STOP
Danger zone, do not proceed
:::

::: details Click me to toggle the code
```js
console.log('Hello, VitePress!')
```
:::
````

Output

> **STOP**
>
> Danger zone, do not proceed

Click me to toggle the code

```
console.log('Hello, VitePress!')
```

Also, you may set custom titles globally by adding the following content in site config, helpful if not writing in English:

```
// config.ts
export default defineConfig({
  // ...
  markdown: {
    container: {
      tipLabel: '??',
      warningLabel: '??',
      dangerLabel: '??',
      infoLabel: '??',
      detailsLabel: '????'
    }
  }
  // ...
})
```

On multilingual sites, these labels can also be overridden per locale - see Per-locale Markdown Strings.

Registering New Containers

Beyond the built-in types, you can register additional containers by mapping their names to their default titles:

```ts
// config.ts
export default defineConfig({
  // ...
  markdown: {
    container: {
      customContainers: {
        success: 'SUCCESS'
      }
    }
  }
  // ...
})
```

Registered names work like the built-in ones - including custom titles, attributes, and the GitHub-style alert syntax:

```
::: success
You have completed the walkthrough!
:::

> [!SUCCESS] Custom title
> This renders the same way.
```

New containers ship without any styling, so add some in your theme using the container name as the class. For this example, the default theme's palette already provides fitting colors:

```css
/* .vitepress/theme/custom.css */
.custom-block.success {
  border-color: transparent;
  color: var(--vp-c-text-1);
  background-color: var(--vp-c-success-soft);
}
```

Nesting

The `:::` markers follow the same rules as fenced code blocks (```` ``` ````): a fence is only closed by a matching fence that is **at least as long** as the one that opened it. To nest containers (or to mix them with code groups) make the outer fence longer than the ones inside it.

Input

````
:::: info Outer container
This box contains another container.

::: details Inner container
```js
console.log('Hello, VitePress!')
```
:::
::::
````

Output

> **OUTER CONTAINER**
>
> This box contains another container.
>
> Inner container
>
> ```
> console.log('Hello, VitePress!')
> ```

Additional Attributes

You can add additional attributes to the custom containers. We use @mdit/plugin-attrs for this feature, and it is supported on almost all markdown elements. For example, you can set the `open` attribute to make the details block open by default:

Input

````
::: details Click me to toggle the code {open}
```js
console.log('Hello, VitePress!')
```
:::
````

Output

Click me to toggle the code

```
console.log('Hello, VitePress!')
```

The special `no-title` attribute renders a container without a title element (it has no effect on `details`, which always needs its summary):

Input

```
::: tip {no-title}
Just want to try it out? Skip to the [Quickstart](./getting-started).
:::
```

Output

> **TIP**
>
> Just want to try it out? Skip to the Quickstart.

raw

This is a special container that can be used to prevent style and router conflicts with VitePress. This is especially useful when you're documenting component libraries.

Syntax

```
::: raw
Wraps in a `<div class="vp-raw">`
:::
```

`vp-raw` class can be directly used on elements too. Style isolation is currently opt-in:

- Install `postcss` with your preferred package manager:

  ```
  $ npm add -D postcss
  ```

- Create a file named `docs/postcss.config.mjs` and add this to it:

  ```
  import { postcssIsolateStyles } from 'vitepress'

  export default {
    plugins: [postcssIsolateStyles()]
  }
  ```

  You can pass its options like this:

  ```
  postcssIsolateStyles({
    includeFiles: [/custom\.css/] // defaults to [/vp-doc\.css/, /base\.css/]
  })
  ```

GitHub-flavored Alerts

VitePress also supports GitHub-flavored alerts to render as callouts. They will be rendered the same as the custom containers. Unlike on GitHub, text placed right after the marker becomes the title of the alert (`> [!NOTE] Custom Title`), and containers you registered yourself work here too.

```
> [!NOTE]
> Highlights information that users should take into account, even when skimming.

> [!TIP]
> Optional information to help a user be more successful.

> [!IMPORTANT]
> Crucial information necessary for users to succeed.

> [!WARNING]
> Critical content demanding immediate user attention due to potential risks.

> [!CAUTION]
> Negative potential consequences of an action.
```

> **NOTE**
>
> Highlights information that users should take into account, even when skimming.

> **TIP**
>
> Optional information to help a user be more successful.

> **IMPORTANT**
>
> Crucial information necessary for users to succeed.

> **WARNING**
>
> Critical content demanding immediate user attention due to potential risks.

> **CAUTION**
>
> Negative potential consequences of an action.

By default, alert colors match GitHub's, with caution and danger both rendering in red. Enable `themeConfig.gradedContainers` to use a graded severity scale: danger (red), warning (orange), and caution (yellow). Note that `[!DANGER]` is a VitePress extension and will render as a regular blockquote on GitHub.

Syntax Highlighting in Code Blocks

VitePress uses Shiki to highlight language syntax in Markdown code blocks, using coloured text. Shiki supports a wide variety of programming languages. All you need to do is append a valid language alias to the beginning backticks for the code block:

Input

````
```js
export default {
  name: 'MyComponent',
  // ...
}
```
````

````
```html
<ul>
  <li v-for="todo in todos" :key="todo.id">
    {{ todo.text }}
  </li>
</ul>
```
````

Output

```
export default {
  name: 'MyComponent'
  // ...
}
```

```
<ul>
  <li v-for="todo in todos" :key="todo.id">
    {{ todo.text }}
  </li>
</ul>
```

A list of valid languages is available on Shiki's repository.

You may also customize syntax highlight theme, configure language aliases, and set custom language labels in app config. Please see `markdown` options for more details.

Line Highlighting in Code Blocks

Input

````
```js{4}
export default {
  data () {
    return {
      msg: 'Highlighted!'
    }
  }
}
```
````

Output

```
export default {
  data () {
    return {
      msg: 'Highlighted!'
    }
  }
}
```

In addition to a single line, you can also specify multiple single lines, ranges, or both:

- Line ranges: for example `{5-8}`, `{3-10}`, `{10-17}`
- Multiple single lines: for example `{4,7,9}`
- Line ranges and single lines: for example `{4,7-13,16,23-27,40}`

Input

````
```js{1,4,6-8}
export default { // Highlighted
  data () {
    return {
      msg: `Highlighted!
      This line isn't highlighted,
      but this and the next 2 are.`,
      motd: 'VitePress is awesome',
      lorem: 'ipsum'
    }
  }
}
```
````

Output

```
export default { // Highlighted
  data () {
    return {
      msg: `Highlighted!
      This line isn't highlighted,
      but this and the next 2 are.`,
      motd: 'VitePress is awesome',
      lorem: 'ipsum',
    }
  }
}
```

Alternatively, it's possible to highlight directly in the line by using the `// [!code highlight]` comment.

Input

````
```js
export default {
  data () {
    return {
      msg: 'Highlighted!' // [!code highlight]
    }
  }
}
```
````

Output

```
export default {
  data() {
    return {
      msg: 'Highlighted!'
    }
  }
}
```

Focus in Code Blocks

Adding the `// [!code focus]` comment on a line will focus it and blur the other parts of the code.

Additionally, you can define a number of lines to focus using `// [!code focus:<lines>]`.

Input

````
```js
export default {
  data () {
    return {
      msg: 'Focused!' // [!code focus]
    }
  }
}
```
````

Output

```
export default {
  data() {
    return {
      msg: 'Focused!'
    }
  }
}
```

Colored Diffs in Code Blocks

Adding the `// [!code --]` or `// [!code ++]` comments on a line will create a diff of that line, while keeping the colors of the codeblock.

Input

````
```js
export default {
  data () {
    return {
      msg: 'Removed' // [!code --]
      msg: 'Added' // [!code ++]
    }
  }
}
```
````

Output

```
export default {
  data () {
    return {
      msg: 'Removed'
      msg: 'Added'
    }
  }
}
```

Errors and Warnings in Code Blocks

Adding the `// [!code warning]` or `// [!code error]` comments on a line will color it accordingly.

Input

````
```js
export default {
  data () {
    return {
      msg: 'Error', // [!code error]
      msg: 'Warning' // [!code warning]
    }
  }
}
```
````

Output

```
export default {
  data() {
    return {
      msg: 'Error',
      msg: 'Warning'
    }
  }
}
```

Line Numbers

You can enable line numbers for each code blocks via config:

```
export default {
  markdown: {
    lineNumbers: true
  }
}
```

Please see `markdown` options for more details.

You can add `:line-numbers` / `:no-line-numbers` mark in your fenced code blocks to override the value

set in config.

You can also customize the starting line number by adding `=` after `:line-numbers`. For example, `:line-numbers=2` means the line numbers in code blocks will start from `2`.

Input

````
```ts {1}
// line-numbers is disabled by default
const line2 = 'This is line 2'
const line3 = 'This is line 3'
```

```ts:line-numbers {1}
// line-numbers is enabled
const line2 = 'This is line 2'
const line3 = 'This is line 3'
```

```ts:line-numbers=2 {1}
// line-numbers is enabled and start from 2
const line3 = 'This is line 3'
const line4 = 'This is line 4'
```
````

Output

```ts
// line-numbers is disabled by default
const line2 = 'This is line 2'
const line3 = 'This is line 3'
```

```ts
// line-numbers is enabled
const line2 = 'This is line 2'
const line3 = 'This is line 3'
```

```ts
// line-numbers is enabled and start from 2
const line3 = 'This is line 3'
const line4 = 'This is line 4'
```

Import Code Snippets

You can import code snippets from existing files via following syntax:

```
<<< @/filepath
```

It also supports line highlighting:

```
<<< @/filepath{highlightLines}
```

Input

```
<<< @/snippets/snippet.js{2}
```

Code file

snippet.js

```
export default function () {
  // ..
}
```

Output

snippet.js

```
export default function () {
  // ..
}
```

> **TIP**
>
> The value of `@` corresponds to the source root. By default it's the VitePress project root, unless `srcDir` is configured. Alternatively, you can also import from relative paths:
>
> ```
> <<< ../snippets/snippet.js
> ```

You can also use a VS Code region to only include the corresponding part of the code file. You can provide a custom region name after a `#` following the filepath:

Input

```
<<< @/snippets/snippet-with-region.js#snippet{1}
```

Code file

snippet-with-region.js

```
// #region snippet
function foo() {
  // ..
}
// #endregion snippet

export default foo
```

Output

snippet-with-region.js

```
function foo() {
  // ..
}
```

If a file contains multiple regions with the same name, all of them are imported and concatenated — including regions written in different comment styles, such as a `<!-- #region -->` in the template and a `// #region` in the script of the same Vue SFC. The marker comments delimiting them are removed from the output; set `markdown.snippet.stripRegionMarkers` to `'all'` to also remove markers of other comment styles nested inside the region, or to `false` to keep all of them.

> **TIP**
>
> Region names may contain letters, digits, `_`, `-` and `.`. Since the name is taken from the end of the path, a file whose name itself contains a `#` needs an explicit region — write `<<< ./my#file.js#region` rather than `<<< ./my#file.js`.

> **WARNING**
>
> Importing a file or region that does not exist throws a build error. Set `markdown.snippet.silent: true` to log a warning and render nothing instead.

You can also specify the language inside the braces (`{}`) like this:

```
<<< @/snippets/snippet.cs{c#}

<!-- with line highlighting: -->

<<< @/snippets/snippet.cs{1,2,4-6 c#}

<!-- with line numbers: -->

<<< @/snippets/snippet.cs{1,2,4-6 c#:line-numbers}
```

This is helpful if source language cannot be inferred from your file extension. Only alphanumeric extensions are inferred, so files like `main.c++` or `scss.code-snippets` need the language spelled out this way.

Anything after the language inside the braces is passed along to the code block as extra attributes — for example, `<<< @/snippets/snippet.ts{ts twoslash}` enables twoslash processing when `@shikijs/vitepress-twoslash` is configured. Note that attributes may not contain square brackets.

Code Groups

You can group multiple code blocks like this:

Input

````
::: code-group

```js [config.js]
/**
 * @type {import('vitepress').UserConfig}
 */
const config = {
  // ...
}

export default config
```

```ts [config.ts]
import type { UserConfig } from 'vitepress'

const config: UserConfig = {
  // ...
}

export default config
```

:::
````

Output

```
/**
 * @type {import('vitepress').UserConfig}
 */
const config = {
  // ...
}

export default config
```

```
import type { UserConfig } from 'vitepress'

const config: UserConfig = {
  // ...
}

export default config
```

You can also import snippets in code groups:

Input

```
::: code-group

<!-- filename is used as title by default -->

<<< @/snippets/snippet.js

<!-- you can provide a custom one too -->

<<< @/snippets/snippet-with-region.js#snippet{1,2 ts:line-numbers} [snippet with region]

:::
```

Output

```
export default function () {
  // ..
}
```

```
function foo() {
  // ..
}
```

Markdown File Inclusion

You can include a markdown file in another markdown file, even nested.

> **TIP**
>
> You can also prefix the markdown path with `@`, and it will act as the source root. By default, the source root is the VitePress project root, unless `srcDir` is configured.

For example, you can include a relative markdown file using this:

Input

```
# Docs

## Basics

<!--@include: ./parts/basics.md-->
```

Part file (`parts/basics.md`)

```
Some getting started stuff.

### Configuration

Can be created using `.foorc.json`.
```

Equivalent code

```
# Docs

## Basics

Some getting started stuff.

### Configuration

Can be created using `.foorc.json`.
```

It also supports selecting a line range:

Input

```
# Docs

## Basics

<!--@include: ./parts/basics.md{3,}-->
```

Part file (`parts/basics.md`)

```
Some getting started stuff.

### Configuration

Can be created using `.foorc.json`.
```

Equivalent code

```
# Docs

## Basics

### Configuration

Can be created using `.foorc.json`.
```

The format of the selected line range can be: `{3,}`, `{,10}`, `{1,10}`

You can also use a VS Code region to only include the corresponding part of the code file. You can provide a custom region name after a `#` following the filepath:

Input

```
# Docs

## Basics

<!--@include: ./parts/basics.md#basic-usage{,2}-->
<!--@include: ./parts/basics.md#basic-usage{5,}-->
```

Part file (`parts/basics.md`)

```
<!-- #region basic-usage -->
## Usage Line 1

## Usage Line 2

## Usage Line 3
<!-- #endregion basic-usage -->
```

Equivalent code

```
# Docs

## Basics

## Usage Line 1

## Usage Line 3
```

> **WARNING**
>
> Including a missing file, region, heading anchor, or an out-of-range line selection throws a build error. Set `markdown.include.silent: true` to log a warning and skip the inclusion instead.

Instead of VS Code regions, you can also use header anchors to include a specific section of the file. For example, if you have a header in your markdown file like this:

```
## My Base Section

Some content here.

### My Sub Section

Some more content here.

## Another Section

Content outside `My Base Section`.
```

You can include the `My Base Section` section like this:

```
## My Extended Section
<!--@include: ./parts/basics.md#my-base-section-->
```

Equivalent code

```
## My Extended Section

Some content here.

### My Sub Section

Some more content here.
```

Here, `my-base-section` is the generated id of the heading element. In case it's not easily guessable, you can open the part file in your browser and click on the heading anchor (`#` symbol left to the heading when hovered) to see the id in the URL bar. Or use browser dev tools to inspect the element. Alternatively, you can also specify the id to the part file like this:

```
## My Base Section {#custom-id}
```

and include it like this:

```
<!--@include: ./parts/basics.md#custom-id-->
```

Relative links and images inside included files resolve from the *included* file's location, so a partial can link to its neighbors no matter which page includes it. Set `markdown.include.rebaseRelativeUrls: false` to leave them resolving relative to the including page instead.

Including Code Files

Since inclusion happens before code blocks are parsed, the directive also works inside fences. Combined with a line range, this lets you show only part of a code file — an alternative to importing snippets when regions are not an option:

Input

````
```js
<!--@include: @/snippets/snippet-with-region.js{2,4}-->
```
````

Output

```
function foo() {
  // ..
}
```

Note that the included lines are inserted verbatim (indentation is preserved), and content containing backticks needs a longer outer fence.

Math Equations

This is currently opt-in. To enable it, you need to install `markdown-it-mathjax3` and set `markdown.math` to `true` in your config file:

```
npm add -D markdown-it-mathjax3@^4
```

.vitepress/config.ts

```
export default {
  markdown: {
    math: true
  }
}
```

Input

```
When $a \ne 0$, there are two solutions to $(ax^2 + bx + c = 0)$ and they are
$$ x = {-b \pm \sqrt{b^2-4ac} \over 2a} $$

**Maxwell's equations:**

| equation
                                                                                  | descriptio
  n                                                                              |
| ---------------------------------------------------------------------------------------------
  ---------------------------------------------------------------------------- | ----------
  ---------------------------------------------------------------------------- |
| $\nabla \cdot \vec{\mathbf{B}}  = 0$
                                                                                  | divergence
   of $\vec{\mathbf{B}}$ is zero                                                 |
| $\nabla \times \vec{\mathbf{E}}\, +\, \frac1c\, \frac{\partial\vec{\mathbf{B}}}{\partial t}
    = \vec{\mathbf{0}}$                                                           | curl of $\
  vec{\mathbf{E}}$ is proportional to the rate of change of $\vec{\mathbf{B}}$ |
| $\nabla \times \vec{\mathbf{B}} -\, \frac1c\, \frac{\partial\vec{\mathbf{E}}}{\partial t} =
   \frac{4\pi}{c}\vec{\mathbf{j}}    \nabla \cdot \vec{\mathbf{E}} = 4 \pi \rho$ | _wha?_
                                                                                |
```

Output

When a!=0, there are two solutions to (ax2+bx+c=0) and they are

x=-b±b2-4ac2a

Maxwell's equations:

equation	description
??B->=0	divergence of B-> is zero
?×E->+1c?B->?t=0->	curl of E-> is proportional to the rate of change of B->
?×B->-1c?E->?t=4?cj->??E->=4??	wha?

Image Lazy Loading

You can enable lazy loading for each image added via markdown by setting `lazyLoad` to `true` in your config file:

```
export default {
  markdown: {
    image: {
      // image lazy loading is disabled by default
      lazyLoad: true
    }
  }
}
```

Advanced Configuration

VitePress uses markdown-it as the Markdown renderer. A lot of the extensions above are implemented via custom plugins. You can further customize the `markdown-it` instance using the `markdown` option in `.vitepress/config.js`:

```
import { defineConfig } from 'vitepress'
import { headerLink } from '@mdit/plugin-anchor'
import markdownItFoo from 'markdown-it-foo'

export default defineConfig({
  markdown: {
    // options for @mdit/plugin-anchor
    // https://mdit-plugins.github.io/anchor.html
    anchor: {
      permalink: headerLink()
    },

    // options for @mdit-vue/plugin-toc
    // https://github.com/mdit-vue/mdit-vue/tree/main/packages/plugin-toc#options
    toc: { level: [1, 2] },

    config: (md) => {
      // use more markdown-it plugins!
      md.use(markdownItFoo)
    }
  }
})
```

See full list of configurable properties in Config Reference: App Config.

---

1. Definitions can contain **markdown** and are rendered at the end of the page.
2. This is an inline footnote.

Extending the Default Theme

VitePress' default theme is optimized for documentation, and can be customized. Consult the Default Theme Config Overview for a comprehensive list of options.

However, there are a number of cases where configuration alone won't be enough. For example:

1. You need to tweak the CSS styling;
2. You need to modify the Vue app instance, for example to register global components;
3. You need to inject custom content into the theme via layout slots.

These advanced customizations will require using a custom theme that "extends" the default theme.

> **TIP**
>
> Before proceeding, make sure to first read Using a Custom Theme to understand how custom themes work.

Customizing CSS

The default theme CSS is customizable by overriding root level CSS variables:

.vitepress/theme/index.js

```
import DefaultTheme from 'vitepress/theme'
import './custom.css'

export default DefaultTheme
```

```
/* .vitepress/theme/custom.css */
:root {
  --vp-c-brand-1: #646cff;
  --vp-c-brand-2: #747bff;
}
```

See default theme CSS variables that can be overridden.

Navbar

The navbar draws a single background surface controlled by CSS variables, so its look can be changed without touching component internals:

```
:root {
  /* bar height and background */
  --vp-nav-height: 4rem;
  --vp-nav-bg-color: var(--vp-c-bg);

  /* background while on top of the home page (unscrolled);
     set to var(--vp-nav-bg-color) to opt out of the transparent treatment */
```

```
  --vp-nav-home-bg-color: transparent;

  /* filter applied to the content behind the bar */
  --vp-nav-backdrop-filter: none;

  /* the bar's bottom rule and the mobile menu background */
  --vp-nav-divider-color: var(--vp-c-gutter);
  --vp-nav-screen-bg-color: var(--vp-c-bg);
}
```

For example, a frosted-glass navbar:

```
:root {
  --vp-nav-bg-color: color-mix(in srgb, var(--vp-c-bg) 65%, transparent);
  --vp-nav-backdrop-filter: saturate(180%) blur(8px);
}
```

The same treatment carries over to the local nav: `--vp-local-nav-bg-color` follows the navbar surface color by default, and where the two bars meet they share a single blurred surface, so the glass stays continuous across them.

> **WARNING**
>
> `backdrop-filter` has a measurable scroll performance cost, especially on large or high-DPI screens. When using a translucent bar, also check text contrast over your page content. Safari 17 and earlier don't apply variable-driven backdrop filters, so they show the translucent color without the blur.

When the nav items don't fit the available width, they move into the `?` menu at the end of the navbar instead of being clipped, starting with the social links, the appearance switch and the locale switcher, followed by the nav items right-to-left. Its button label can be localized with `extraMenuLabel`.

Using Different Fonts

VitePress uses Inter as the default font, and will include the fonts in the build output. The font is also auto preloaded in production. However, this may not be desirable if you want to use a different main font.

To avoid including Inter in the build output, import the theme from `vitepress/theme-without-fonts` instead:

.vitepress/theme/index.js

```
import DefaultTheme from 'vitepress/theme-without-fonts'
import './my-fonts.css'

export default DefaultTheme
```

```
/* .vitepress/theme/my-fonts.css */
:root {
  --vp-font-family-base: /* normal text font */
  --vp-font-family-mono: /* code font */
}
```

> **WARNING**
>
> If you are using optional components like the Team Page components, make sure to also import them from `vitepress/theme-without-fonts`!

If your font is a local file referenced via `@font-face`, it will be processed as an asset and included under `.vitepress/dist/assets` with hashed filename. To preload this file, use the transformHead build hook:

.vitepress/config.js

```
export default {
  transformHead({ assets }) {
    // adjust the regex accordingly to match your font
    const myFontFile = assets.find(file => /font-name\.[\w-]+\.woff2/.test(file))
    if (myFontFile) {
      return [
        [
          'link',
          {
            rel: 'preload',
            href: myFontFile,
            as: 'font',
            type: 'font/woff2',
            crossorigin: ''
          }
        ]
      ]
    }
  }
}
```

Registering Global Components

.vitepress/theme/index.js

```
import DefaultTheme from 'vitepress/theme'

/** @type {import('vitepress').Theme} */
export default {
  extends: DefaultTheme,
  enhanceApp({ app }) {
    // register your custom global components
    app.component('MyGlobalComponent' /* ... */)
  }
}
```

If you're using TypeScript:

.vitepress/theme/index.ts

```ts
import type { Theme } from 'vitepress'
import DefaultTheme from 'vitepress/theme'

export default {
  extends: DefaultTheme,
  enhanceApp({ app }) {
    // register your custom global components
    app.component('MyGlobalComponent' /* ... */)
  }
} satisfies Theme
```

Since we are using Vite, you can also leverage Vite's glob import feature to auto register a directory of components.

Layout Slots

The default theme's `<Layout/>` component has a few slots that can be used to inject content at certain locations of the page. Here's an example of injecting a component into the before outline:

.vitepress/theme/index.js

```js
import DefaultTheme from 'vitepress/theme'
import MyLayout from './MyLayout.vue'

export default {
  extends: DefaultTheme,
  // override the Layout with a wrapper component that
  // injects the slots
  Layout: MyLayout
}
```

.vitepress/theme/MyLayout.vue

```vue
<script setup>
import DefaultTheme from 'vitepress/theme'

const { Layout } = DefaultTheme
</script>

<template>
  <Layout>
    <template #aside-outline-before>
      My custom sidebar top content
    </template>
  </Layout>
</template>
```

Or you could use render function as well.

.vitepress/theme/index.js

```
import { h } from 'vue'
import DefaultTheme from 'vitepress/theme'
import MyComponent from './MyComponent.vue'

export default {
  extends: DefaultTheme,
  Layout() {
    return h(DefaultTheme.Layout, null, {
      'aside-outline-before': () => h(MyComponent)
    })
  }
}
```

Full list of slots available in the default theme layout:

- When `layout: 'doc'` (default) is enabled via frontmatter:
  - `doc-top`
  - `doc-bottom`
  - `doc-footer-before`
  - `doc-before`
  - `doc-after`
  - `sidebar-nav-before`
  - `sidebar-nav-after`
  - `aside-top`
  - `aside-bottom`
  - `aside-outline-before`
  - `aside-outline-after`
  - `aside-ads-before`
  - `aside-ads-after`
- When `layout: 'home'` is enabled via frontmatter:
  - `home-hero-before`
  - `home-hero-info-before`
  - `home-hero-info`
  - `home-hero-info-after`
  - `home-hero-actions-before-actions`
  - `home-hero-actions-after`
  - `home-hero-image`
  - `home-hero-after`
  - `home-features-before`
  - `home-features-after`
- When `layout: 'page'` is enabled via frontmatter:
  - `page-top`
  - `page-bottom`

- On not found (404) page:
  - `not-found`
- Always:
  - `layout-top`
  - `layout-bottom`
  - `nav-bar-title-before`
  - `nav-bar-title-after`
  - `nav-bar-content-before`
  - `nav-bar-content-after`
  - `nav-screen-content-before`
  - `nav-screen-content-after`

Using View Transitions API

On Appearance Toggle

You can extend the default theme to provide a custom transition when the color mode is toggled. An example:

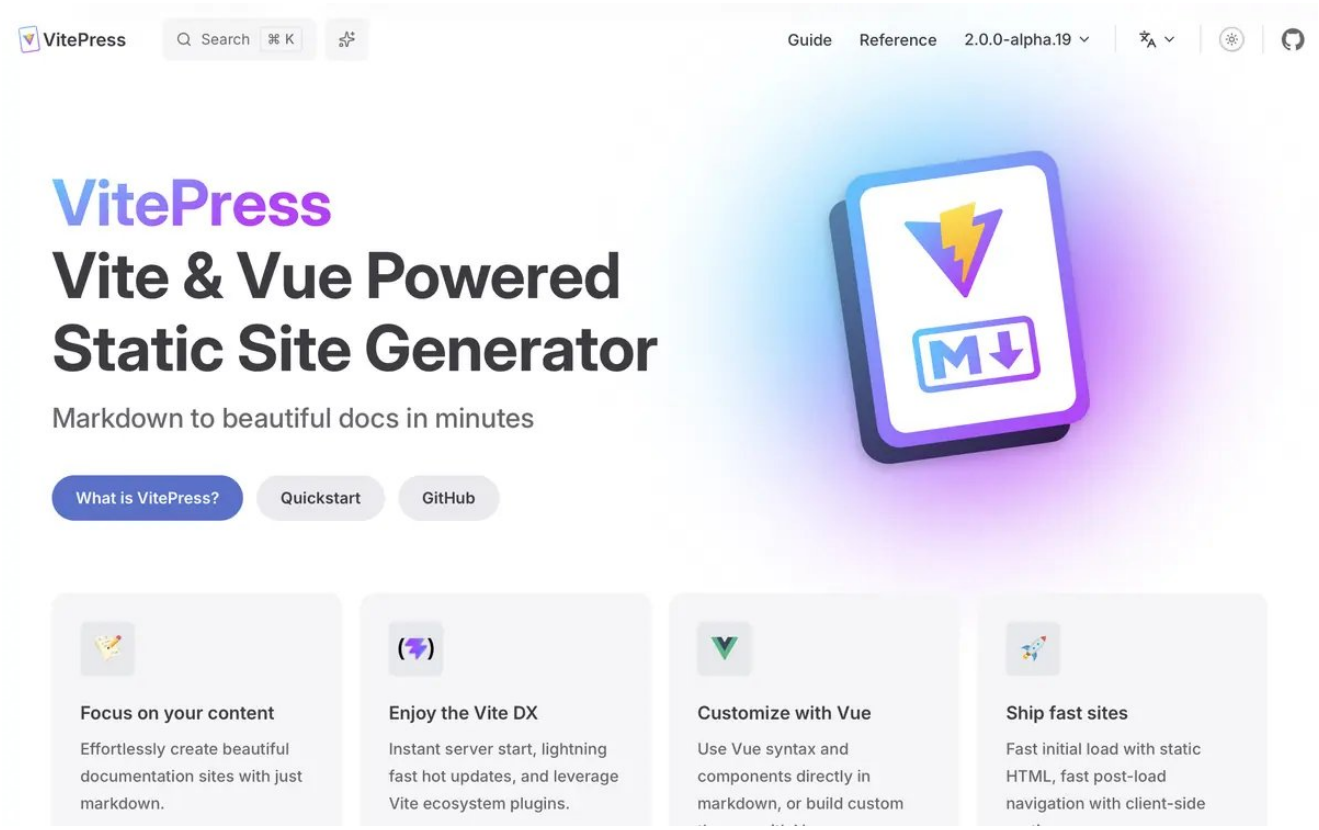
.vitepress/theme/Layout.vue

```
<script setup lang="ts">
import { useData } from 'vitepress'
import DefaultTheme from 'vitepress/theme'
import { nextTick, provide } from 'vue'

const { isDark } = useData()

const enableTransitions = () =>
  'startViewTransition' in document &&
  window.matchMedia('(prefers-reduced-motion: no-preference)').matches

provide('toggle-appearance', ({ clientX, clientY }: MouseEvent) => {
  if (!enableTransitions()) {
    isDark.value = !isDark.value
    return
  }

  const x = (100 * clientX) / innerWidth
  const y = (100 * clientY) / innerHeight
  const maxRadius =
    (100 *
      Math.hypot(
        Math.max(clientX, innerWidth - clientX),
        Math.max(clientY, innerHeight - clientY)
      )) /
    (Math.hypot(innerWidth, innerHeight) / Math.SQRT2)

  document.documentElement.style.setProperty('--switch-x', `${x}%`)
  document.documentElement.style.setProperty('--switch-y', `${y}%`)
```

```
  document.documentElement.style.setProperty('--switch-r', `${maxRadius}%`)

  document.startViewTransition(async () => {
    isDark.value = !isDark.value
    await nextTick()
  })
})
</script>

<template>
  <DefaultTheme.Layout />
</template>

<style>
::view-transition-old(root),
::view-transition-new(root) {
  animation: none;
  mix-blend-mode: normal;
}

::view-transition-new(root) {
  animation: switch-appearance 300ms ease-in;
}

.dark::view-transition-new(root) {
  animation: none;
}

.dark::view-transition-old(root) {
  animation: switch-appearance 300ms ease-in reverse forwards;
  z-index: 1;
}

@keyframes switch-appearance {
  from {
    clip-path: circle(0 at var(--switch-x) var(--switch-y));
  }
  to {
    clip-path: circle(var(--switch-r) at var(--switch-x) var(--switch-y));
  }
}

.VPSwitchAppearance {
  width: 1.375rem !important;
}

.VPSwitchAppearance .check {
  transform: none !important;
}
</style>
```

Result (**warning!**: flashing colors, sudden movements, bright lights):

Demo



Refer Chrome Docs from more details on view transitions.

On Route Change

Coming soon.

Overriding Internal Components

You can use Vite's aliases to replace default theme components with your custom ones:

```
import { fileURLToPath, URL } from 'node:url'
import { defineConfig } from 'vitepress'

export default defineConfig({
  vite: {
    resolve: {
      alias: [
        {
          find: /^.*\/VPNavBar\.vue$/,
          replacement: fileURLToPath(
            new URL('./theme/components/CustomNavBar.vue', import.meta.url)
          )
        }
      ]
    }
  }
})
```

To know the exact name of the component refer our source code. Since the components are internal, there is a slight chance their name is updated between minor releases.