VITEPRESS Docs - English

Table of contents

API Reference

VitePress offers several built in API to let you access app data. VitePress also comes with few built-in component that can be used globally.

The helper methods are globally importable from `vitepress` and are typically used in custom theme Vue components. However, they are also usable inside `.md` pages because markdown files are compiled into Vue single-file components.

Methods that start with `use*` indicates that it is a Vue 3 Composition API function that can only be used inside `setup()` or `<script setup>`.

`useData`

Returns page-specific data. The returned object has the following type:

```
interface VitePressData {
  site: Ref<SiteData>
  page: Ref<PageData>
  theme: Ref<any> // themeConfig from .vitepress/config.js
  frontmatter: Ref<PageData['frontmatter']>
  lang: Ref<string>
  title: Ref<string>
  description: Ref<string>
  localePath: Ref<string>
  isDark: Ref<boolean>
}
```

Example:

```
<script setup>
import { useData } from 'vitepress'

const { theme } = useData()
</script>

<template>
  <h1>{{ theme.footer.copyright }}</h1>
</template>
```

`useRoute`

Returns the current route object with the following type:

```
interface Route {
  path: string
  data: PageData
  component: Component | null
}
```

`useRouter`

Returns the VitePress router instance so you can programmatically navigate to another page.

```
interface Router {
  route: Route
  go: (href?: string) => Promise<void>
}
```

`withBase`

- **Type**: `(path: string) => string`

Appends the configured `base` to a given URL path. Also see Base URL.

`<Content />`

The `<Content />` component displays the rendered markdown contents. Useful when creating your own theme.

```
<template>
  <h1>Custom Layout!</h1>
  <Content />
</template>
```

`<ClientOnly />`

The `<ClientOnly />` component renders its slot only at client side.

Because VitePress applications are server-rendered in Node.js when generating static builds, any Vue usage must conform to the universal code requirements. In short, make sure to only access Browser / DOM APIs in beforeMount or mounted hooks.

If you are using or demoing components that are not SSR-friendly (for example, contain custom directives), you can wrap them inside the `ClientOnly` component.

```
<ClientOnly>
  <NonSSRFriendlyComponent />
</ClientOnly>
```

Go to TOC

Asset Handling

All Markdown files are compiled into Vue components and processed by Vite. You can, **and should**, reference any assets using relative URLs:

```
![An image](./image.png.html)
```

You can reference static assets in your markdown files, your `*.vue` components in the theme, styles and plain `.css` files either using absolute public paths (based on project root) or relative paths (based on your file system). The latter is similar to the behavior you are used to if you have used `vue-cli` or webpack's `file-loader`.

Common image, media, and font filetypes are detected and included as assets automatically.

All referenced assets, including those using absolute paths, will be copied to the dist folder with a hashed file name in the production build. Never-referenced assets will not be copied. Similar to `vue-cli`, image assets smaller than 4kb will be base64 inlined.

All **static** path references, including absolute paths, should be based on your working directory structure.

Public Files

Sometimes you may need to provide static assets that are not directly referenced in any of your Markdown or theme components (for example, favicons and PWA icons). The `public` directory under project root (`docs` folder if you're running `vitepress build docs`) can be used as an escape hatch to provide static assets that either are never referenced in source code (e.g. `robots.txt`), or must retain the exact same file name (without hashing).

Assets placed in `public` will be copied to the root of the dist directory as-is.

Note that you should reference files placed in `public` using root absolute path - for example, `public/icon.png` should always be referenced in source code as `/icon.png`.

Base URL

If your site is deployed to a non-root URL, you will need to set the `base` option in `.vitepress/config.js`. For example, if you plan to deploy your site to `https://foo.github.io/bar/`, then `base` should be set to `'/bar/'` (it should always start and end with a slash).

All your static asset paths are automatically processed to adjust for different `base` config values. For example, if you have an absolute reference to an asset under `public` in your markdown:

```
![An image](/image-inside-public.png)
```

You do **not** need to update it when you change the `base` config value in this case.

However, if you are authoring a theme component that links to assets dynamically, e.g. an image whose `src` is based on a theme config value:

```
<img :src="theme.logoPath" />
```

In this case it is recommended to wrap the path with the `withBase` helper provided by VitePress:

```
<script setup>
import { withBase, useData } from 'vitepress'

const { theme } = useData()
</script>

<template>
  <img :src="withBase(theme.logoPath)" />
</template>
```

Go to TOC

Configuration

Without any configuration, the page is pretty minimal, and the user has no way to navigate around the site. To customize your site, let's first create a `.vitepress` directory inside your docs directory. This is where all VitePress-specific files will be placed. Your project structure is probably like this:

```
.
├─ docs
│  ├─ .vitepress
│  │  └─ config.js
│  └─ index.md
└─ package.json
```

The essential file for configuring a VitePress site is `.vitepress/config.js`, which should export a JavaScript object:

```
export default {
  title: 'VitePress',
  description: 'Just playing around.'
}
```

In the above example, the site will have the title of `VitePress`, and `Just playing around.` as the description meta tag.

Learn everything about VitePress features at Theme: Introduction to find how to configure specific features within this config file.

You may also find all configuration references at Configs.

Go to TOC

Deploying

The following guides are based on some shared assumptions:

- You are placing your docs inside the `docs` directory of your project.
- You are using the default build output location (`.vitepress/dist`).
- VitePress is installed as a local dependency in your project, and you have set up the following scripts in your `package.json` :

  ```
  {
    "scripts": {
      "docs:build": "vitepress build docs",
      "docs:serve": "vitepress serve docs"
    }
  }
  ```

::: tip

If your site is to be served at a subdirectory (`https://example.com/subdir/`), then you have to set `'/subdir/'` as the `base` in your `docs/.vitepress/config.js` .

Example: If you're using Github (or GitLab) Pages and deploying to `user.github.io/repo/` , then set your `base` to `/repo/` .

:::

Build and Test Locally

- You may run this command to build the docs:

  ```
  $ yarn docs:build
  ```

- Once you've built the docs, you can test them locally by running:

  ```
  $ yarn docs:serve
  ```

  The `serve` command will boot up a local static web server that will serve the files from `.vitepress/dist` at `http://localhost:4173` . It's an easy way to check if the production build looks fine in your local environment.

- You can configure the port of the server by passing `--port` as an argument.

  ```
  {
    "scripts": {
      "docs:serve": "vitepress serve docs --port 8080"
    }
  }
  ```

Now the `docs:serve` method will launch the server at `http://localhost:8080`.

Netlify, Vercel, AWS Amplify, Cloudflare Pages, Render

Set up a new project and change these settings using your dashboard:

- **Build Command:** `yarn docs:build`
- **Output Directory:** `docs/.vitepress/dist`
- **Node Version:** `14` (or above, by default it usually will be 14 or 16, but on Cloudflare Pages the default is still 12, so you may need to change that)

::: warning Don't enable options like *Auto Minify* for HTML code. It will remove comments from output which have meaning to Vue. You may see hydration mismatch errors if they get removed. :::

GitHub Pages

Using GitHub Actions

1. In your theme config file, `docs/.vitepress/config.js`, set the `base` property to the name of your GitHub repository. If you plan to deploy your site to `https://foo.github.io/bar/`, then you should set base to `'/bar/'`. It should always start and end with a slash.

2. Create a file named `deploy.yml` inside `.github/workflows` directory of your project with the following content:

```yaml
name: Deploy

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - uses: actions/setup-node@v3
        with:
          node-version: 16
          cache: yarn
      - run: yarn install --frozen-lockfile

      - name: Build
        run: yarn docs:build

      - name: Deploy
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: docs/.vitepress/dist
          # cname: example.com # if wanna deploy to custom domain
```

::: tip Please replace the corresponding branch name. For example, if the branch you want to build is `master`, then you should replace `main` with `master` in the above file. :::

3. Now commit your code and push it to the `main` branch.
4. Wait for actions to complete.
5. In your repository's Settings under Pages menu item, select `gh-pages` branch as GitHub Pages source. Now your docs will automatically deploy each time you push.

GitLab Pages

Using GitLab CI

1. Set `outDir` in `docs/.vitepress/config.js` to `../public`.
2. Create a file called `.gitlab-ci.yml` in the root of your project with the content below. This will build and deploy your site whenever you make changes to your content:

```
image: node:16
pages:
  cache:
    paths:
      - node_modules/
  script:
    - yarn install
    - yarn docs:build
  artifacts:
    paths:
      - public
  only:
    - main
```

Azure Static Web Apps

1. Follow the official documentation.
2. Set these values in your configuration file (and remove the ones you don't require, like `api_location`):
   - **`app_location`** : `/`
   - **`output_location`** : `docs/.vitepress/dist`
   - **`app_build_command`** : `yarn docs:build`

Firebase

1. Create `firebase.json` and `.firebaserc` at the root of your project:

   `firebase.json` :

```json
{
  "hosting": {
    "public": "docs/.vitepress/dist",
    "ignore": []
  }
}
```

`.firebaserc`:

```json
{
  "projects": {
    "default": "<YOUR_FIREBASE_ID>"
  }
}
```

2. After running `yarn docs:build`, run this command to deploy:

```
firebase deploy
```

Surge

1. After running `yarn docs:build`, run this command to deploy:

```
npx surge docs/.vitepress/dist
```

Heroku

1. Follow documentation and guide given in `heroku-buildpack-static`.

2. Create a file called `static.json` in the root of your project with the below content:

```json
{
  "root": "docs/.vitepress/dist"
}
```

Layer0

Refer Creating and Deploying a VitePress App with Layer0.

Frontmatter

Any Markdown file that contains a YAML frontmatter block will be processed by gray-matter. The frontmatter must be at the top of the Markdown file, and must take the form of valid YAML set between triple-dashed lines. Example:

```
---
title: Docs with VitePress
editLink: true
---
```

Between the triple-dashed lines, you can set predefined variables, or even create custom ones of your own. These variables can be used via the special `$frontmatter` variable.

Here's an example of how you could use it in your Markdown file:

```
---
title: Docs with VitePress
editLink: true
---

# {{ $frontmatter.title }}

Guide content
```

Alternative Frontmatter Formats

VitePress also supports JSON frontmatter syntax, starting and ending in curly braces:

```
---
{
  "title": "Blogging Like a Hacker",
  "editLink": true
}
---
```

Go to TOC

Getting Started

This section will help you build a basic VitePress documentation site from ground up. If you already have an existing project and would like to keep documentation inside the project, start from Step 2.

You can also try VitePress online on StackBlitz. It runs the VitePress-based site directly in the browser, so it is almost identical to the local setup but doesn't require installing anything on your machine.

::: warning VitePress is currently in `alpha` status. It is already suitable for out-of-the-box documentation use, but the config and theming API may still change between minor releases. :::

Step. 1: Create a new project

Create and change into a new directory.

```
$ mkdir vitepress-starter && cd vitepress-starter
```

Then, initialize with your preferred package manager.

```
$ yarn init
```

Step. 2: Install VitePress

Add VitePress and Vue as dev dependencies for the project.

```
$ yarn add --dev vitepress vue
```

::: details Getting missing peer deps warnings? `@docsearch/js` has certain issues with its peer dependencies. If you see some commands failing due to them, you can try this workaround for now:

If using PNPM, add this in your `package.json`:

```
"pnpm": {
  "peerDependencyRules": {
    "ignoreMissing": [
      "@algolia/client-search"
    ]
  }
}
```

:::

Create your first document.

```
$ mkdir docs && echo '# Hello VitePress' > docs/index.md
```

Step. 3: Boot up dev environment

Add some scripts to `package.json`.

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

Serve the documentation site in the local server.

```
$ yarn docs:dev
```

VitePress will start a hot-reloading development server at `http://localhost:5173`.

Step. 4: Add more pages

Let's add another page to the site. Create a file name `getting-started.md` along with `index.md` you've created in Step. 2. Now your directory structure should look like this.

```
.
├─ docs
│  ├─ getting-started.md
│  └─ index.md
└─ package.json
```

Then, try to access `http://localhost:5173/getting-started.html` and you should see the content of `getting-started.md` is shown.

This is how VitePress works basically. The directory structure corresponds with the URL path. You add files, and just try to access it.

What's next?

By now, you should have a basic but functional VitePress documentation site. But currently, the user has no way to navigate around the site because it's missing for example sidebar menu we have on this site.

To enable those navigations, we must add some configurations to the site. Head to configuration guide to learn how to configure VitePress.

If you would like to know more about what you can do within the page, for example, writing markdown contents, or using Vue Component, check out the "Writing" section of the docs. Markdown guide would be a great starting point.

If you want to know how to customize how the site looks (Theme), and find out the features VitePress's default theme provides, visit Theme: Introduction.

When your documentation site starts to take shape, be sure to read the deployment guide.

Go to TOC

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.

Links

Both internal and external links gets special treatments.

Internal Links

Internal links are converted to router link for SPA navigation. Also, every `index.md` contained in each subdirectory 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](./.html#heading.html) <!-- 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
zebra stripes	are neat	$1

Emoji :tada:

Input

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

Output

:tada: :100:

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 view the code
```js
console.log('Hello, VitePress!')
```
:::
~~~

Output

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

::: details Click me to view the code

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

:::

`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. You might also wanna check out whyframe for better isolation.

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:

::: details

- Install required deps with your preferred package manager:

  ```
  $ yarn add -D postcss postcss-prefix-selector
  ```

- Create a file named `docs/.postcssrc.cjs` and add this to it:

  ```
  module.exports = {
    plugins: {
      'postcss-prefix-selector': {
        prefix: ':not(:where(.vp-raw *))',
        includeFiles: [/vp-doc\.css/],
        transform(prefix, _selector) {
          const [selector, pseudo = ''] = _selector.split(/(:\S*)$/)
          return selector + prefix + pseudo
        }
      }
    }
  }
  ```

:::

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

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

```html
<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 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 hl]` comment.

Input

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

```
  }
}
```

Output

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

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

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

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

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

Errors and warnings

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

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

Line Numbers

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

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

Please see `markdown` options for more details.

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

<<< @/snippets/snippet.js

Output

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

::: tip The value of `@` corresponds to the source root. By default it's the VitePress project root, unless `srcDir` is configured. :::

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

<<< @/snippets/snippet-with-region.js

Output

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

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#}
```

This is helpful if source language cannot be inferred from your file extension.

Markdown File Inclusion

You can include a markdown file in another markdown file like 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`.
```

::: warning Note that this does not throw errors if your file is not present. Hence, when using this feature make sure that the contents are being rendered as expected. :::

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`:

```
const anchor = require('markdown-it-anchor')

module.exports = {
  markdown: {
    // options for markdown-it-anchor
    // https://github.com/valeriangalliat/markdown-it-anchor#usage
    anchor: {
      permalink: 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(require('markdown-it-xxx'))
    }
  }
}
```

See full list of configurable properties in Configs: App Configs.

Go to TOC

Migration from VitePress 0.x

If you're coming from VitePress 0.x version, there're several breaking changes due to new features and enhancement. Please follow this guide to see how to migrate your app over to the latest VitePress.

App Config

- The internationalization feature is not yet implemented.

Theme Config

- `sidebar` option has changed its structure.
  - `children` key is now named `items`.
  - Top level item may not contain `link` at the moment. We're planning to bring it back.
- `repo`, `repoLabel`, `docsDir`, `docsBranch`, `editLinks`, `editLinkText` are removed in favor of more flexible api.
  - For adding GitHub link with icon to the nav, use Social Links feature.
  - For adding "Edit this page" feature, use Edit Link feature.
- `lastUpdated` option is now split into `config.lastUpdated` and `themeConfig.lastUpdatedText`.
- `carbonAds.carbon` is changed to `carbonAds.code`.

Frontmatter Config

- `home: true` option has changed to `layout: home`. Also, many Homepage related settings have been modified to provide additional features. See Home Page guide for details.
- `footer` option is moved to `themeConfig.footer`.

Go to TOC

Migration from VuePress

Config

Sidebar

The sidebar is no longer automatically populated from frontmatter. You can read the frontmatter yourself to dynamically populate the sidebar. Additional utilities for this may be provided in the future.

Markdown

Images

Unlike VuePress, VitePress handles `base` of your config automatically when you use static image.

Hence, now you can render images without `img` tag.

```diff
- <img :src="$withBase('/foo.png')" alt="foo">
+ ![foo](/foo.png)
```

::: warning For dynamic images you still need `withBase` as shown in Base URL guide. :::

Use `<img.*withBase\('(.*)'\).*alt="([^"]*)".*>` regex to find and replace it with `![$2]($1)` to replace all the images with `![](...)` syntax.

---

more to follow...

Go to TOC

Badge

The badge lets you add status to your headers. For example, it could be useful to specify the section's type, or supported version.

Usage

You may use the `Badge` component which is globally available.

```
### Title <Badge type="info" text="default" />
### Title <Badge type="tip" text="^1.9.0" />
### Title <Badge type="warning" text="beta" />
### Title <Badge type="danger" text="caution" />
```

Code above renders like:

Title

Title

Title

Title

Custom Children

`<Badge>` accept `children`, which will be displayed in the badge.

```
### Title <Badge type="info">custom element</Badge>
```

Title custom element

Customize Type Color

you can customize `background-color` of typed `<Badge />` by override css vars. The following is he default values.

```
:root {
  --vp-badge-info-border: var(--vp-c-divider-light);
  --vp-badge-info-text: var(--vp-c-text-2);
  --vp-badge-info-bg: var(--vp-c-white-soft);

  --vp-badge-tip-border: var(--vp-c-green-dimm-1);
  --vp-badge-tip-text: var(--vp-c-green-darker);
  --vp-badge-tip-bg: var(--vp-c-green-dimm-3);

  --vp-badge-warning-border: var(--vp-c-yellow-dimm-1);
  --vp-badge-warning-text: var(--vp-c-yellow-darker);
  --vp-badge-warning-bg: var(--vp-c-yellow-dimm-3);
  --vp-badge-danger-border: var(--vp-c-red-dimm-1);
  --vp-badge-danger-text: var(--vp-c-red-darker);
  --vp-badge-danger-bg: var(--vp-c-red-dimm-3);
}

.dark {
  --vp-badge-info-border: var(--vp-c-divider-light);
  --vp-badge-info-bg: var(--vp-c-black-mute);

  --vp-badge-tip-border: var(--vp-c-green-dimm-2);
  --vp-badge-tip-text: var(--vp-c-green-light);

  --vp-badge-warning-border: var(--vp-c-yellow-dimm-2);
  --vp-badge-warning-text: var(--vp-c-yellow-light);

  --vp-badge-danger-border: var(--vp-c-red-dimm-2);
  --vp-badge-danger-text: var(--vp-c-red-light);
}
```

`<Badge>`

`<Badge>` component accepts following props:

```
interface Props {
  // When `<slot>` is passed, this value gets ignored.
  text?: string

  // Defaults to `tip`.
  type?: 'info' | 'tip' | 'warning' | 'danger'
}
```

Go to TOC

Carbon Ads

VitePress has built in native support for Carbon Ads. By defining the Carbon Ads credentials in config, VitePress will display ads on the page.

```
export default {
  themeConfig: {
    carbonAds: {
      code: 'your-carbon-code',
      placement: 'your-carbon-placement'
    }
  }
}
```

These values are used to call carbon CDN script as shown below.

```
`//cdn.carbonads.com/carbon.js?serve=${code}&placement=${placement}`
```

To learn more about Carbon Ads configuration, please visit Carbon Ads website.

Go to TOC

Edit Link

Edit Link lets you display a link to edit the page on Git management services such as GitHub, or GitLab. To enable it, add `themeConfig.editLink` options to your config.

```
export default {
  themeConfig: {
    editLink: {
      pattern: 'https://github.com/vuejs/vitepress/edit/main/docs/:path'
    }
  }
}
```

The `pattern` option defines the URL structure for the link, and `:path` is going to be replaced with the page path.

By default, this will add the link text "Edit this page" at the bottom of the doc page. You may customize this text by defining the `text` option.

```
export default {
  themeConfig: {
    editLink: {
      pattern: 'https://github.com/vuejs/vitepress/edit/main/docs/:path',
      text: 'Edit this page on GitHub'
    }
  }
}
```

Go to TOC

Footer

VitePress will display global footer at the bottom of the page when `themeConfig.footer` is present.

```
export default {
  themeConfig: {
    footer: {
      message: 'Released under the MIT License.',
      copyright: 'Copyright © 2019-present Evan You'
    }
  }
}
```

```
export interface Footer {
  // The message shown right before copyright.
  message?: string

  // The actual copyright text.
  copyright?: string
}
```

The above configuration also supports HTML strings. So, for example, if you want to configure footer text to have some links, you can adjust the configuration as follows:

```
export default {
  themeConfig: {
    footer: {
      message: 'Released under the <a
href="https://github.com/vuejs/vitepress/blob/main/LICENSE">MIT License</a>.',
      copyright: 'Copyright © 2019-present <a
href="https://github.com/yyx990803">Evan You</a>'
    }
  }
}
```

Note that footer will not be displayed when the SideBar is visible.

Go to TOC

Home Page

VitePress default theme provides a homepage layout, which you can also see used on the homepage of this site. You may use it on any of your pages by specifying `layout: home` in the frontmatter.

```
---
layout: home
---
```

However, this option alone wouldn't do much. You can add several different pre templated "sections" to the homepage by setting additional other options such as `hero` and `features`.

Hero Section

The Hero section comes at the top of the homepage. Here's how you can configure the Hero section.

```
---
layout: home

hero:
  name: VitePress
  text: Vite & Vue powered static site generator.
  tagline: Lorem ipsum...
  image:
    src: /logo.png
    alt: VitePress
  actions:
    - theme: brand
      text: Get Started
      link: /guide/what-is-vitepress
    - theme: alt
      text: View on GitHub
      link: https://github.com/vuejs/vitepress
---
```

```
interface Hero {
  // The string shown top of `text`. Comes with brand color
  // and expected to be short, such as product name.
  name?: string

  // The main text for the hero section. This will be defined
  // as `h1` tag.
  text: string

  // Tagline displayed below `text`.
  tagline?: string

  // Action buttons to display in home hero section.
  actions?: HeroAction[]
}

interface HeroAction {
  // Color theme of the button. Defaults to `brand`.
  theme?: 'brand' | 'alt'
  // Label of the button.
  text: string

  // Destination link of the button.
  link: string
}
```

Customizing the name color

VitePress uses the brand color (`--vp-c-brand`) for the `name`. However, you may customize this color by overriding `--vp-home-hero-name-color` variable.

```
:root {
  --vp-home-hero-name-color: blue;
}
```

Also you may customize it further by combining `--vp-home-hero-name-background` to give the `name` gradient color.

```
:root {
  --vp-home-hero-name-color: transparent;
  --vp-home-hero-name-background: -webkit-linear-gradient(120deg, #bd34fe,
#41d1ff);
}
```

Features Section

In Features section, you can list any number of features you would like to show right after the Hero section. To configure it, pass `features` option to the frontmatter.

```
---
layout: home

features:
  - icon: ⚡️
    title: Vite, The DX that can't be beat
    details: Lorem ipsum...
  - icon: 🖖
    title: Power of Vue meets Markdown
    details: Lorem ipsum...
  - icon: 🛠️
    title: Simple and minimal, always
    details: Lorem ipsum...
---
```

```
interface Feature {
  // Show icon on each feature box. Currently, only emojis
  // are supported.
  icon?: string

  // Title of the feature.
  title: string

  // Details of the feature.
  details: string

  // Link when clicked on feature component. The link can
```

```
    // be both internal or external.
    //
    // e.g. `guide/theme-home-page` or `htttps://example.com`
    link?: string

    // Link text to be shown inside feature component. Best
    // used with `link` option.
    //
    // e.g. `Learn more`, `Visit page`, etc.
    linkText?: string
  }
```

Go to TOC

Theme Introduction

VitePress comes with its default theme providing many features out of the box. Learn more about each feature on its dedicated page listed below.

- Nav
- Sidebar
- Prev Next Link
- Edit Link
- Last Updated
- Layout
- Home Page
- Team Page
- Badge
- Footer
- Search
- Carbon Ads

If you don't find the features you're looking for, or you would rather create your own theme, you may customize VitePress to fit your requirements. In the following sections, we'll go through each way of customizing the VitePress theme.

Using a Custom Theme

You can enable a custom theme by adding the `.vitepress/theme/index.js` or `.vitepress/theme/index.ts` file (the "theme entry file").

```
.
├─ docs
│  ├─ .vitepress
│  │  ├─ theme
│  │  │  └─ index.js
│  │  └─ config.js
│  └─ index.md
└─ package.json
```

A VitePress custom theme is simply an object containing four properties and is defined as follows:

```
interface Theme {
  Layout: Component // Vue 3 component
  NotFound?: Component
  enhanceApp?: (ctx: EnhanceAppContext) => void
  setup?: () => void
}

interface EnhanceAppContext {
  app: App // Vue 3 app instance
  router: Router // VitePress router instance
  siteData: Ref<SiteData>
}
```

The theme entry file should export the theme as its default export:

```
// .vitepress/theme/index.js
import Layout from './Layout.vue'

export default {
  // root component to wrap each page
  Layout,

  // this is a Vue 3 functional component
  NotFound: () => 'custom 404',

  enhanceApp({ app, router, siteData }) {
    // app is the Vue 3 app instance from `createApp()`.
    // router is VitePress' custom router. `siteData` is
    // a `ref` of current site-level metadata.
  },

  setup() {
    // this function will be executed inside VitePressApp's
    // setup hook. all composition APIs are available here.
  }
}
```

...where the `Layout` component could look like this:

```
<!-- .vitepress/theme/Layout.vue -->
<template>
  <h1>Custom Layout!</h1>

  <!-- this is where markdown content will be rendered -->
  <Content />
</template>
```

The default export is the only contract for a custom theme. Inside your custom theme, it works just like a normal Vite + Vue 3 application. Do note the theme also needs to be SSR-compatible.

To distribute a theme, simply export the object in your package entry. To consume an external theme, import and re-export it from the custom theme entry:

```
// .vitepress/theme/index.js
import Theme from 'awesome-vitepress-theme'

export default Theme
```

Extending the Default Theme

If you want to extend and customize the default theme, you can import it from `vitepress/theme` and augment it in a custom theme entry. Here are some examples of common customizations:

Registering Global Components

```js
// .vitepress/theme/index.js
import DefaultTheme from 'vitepress/theme'

export default {
  ...DefaultTheme,
  enhanceApp(ctx) {
    // extend default theme custom behaviour.
    DefaultTheme.enhanceApp(ctx)

    // register your custom global components
    ctx.app.component('MyGlobalComponent' /* ... */)
  }
}
```

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

Customizing CSS

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

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

export default DefaultTheme
```

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

See default theme CSS variables that can be overridden.

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:

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

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

```vue
<!--.vitepress/theme/MyLayout.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 DefaultTheme from 'vitepress/theme'
import MyComponent from './MyComponent.vue'

export default {
  ...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-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-after`
  - `home-features-before`
  - `home-features-after`
- 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`

Go to TOC

Go to TOC

Last Updated

Documentation coming soon...

Go to TOC

Layout

You may choose the page layout by setting `layout` option to the page frontmatter. There are 3 layout options, `doc`, `page`, and `home`. If nothing is specified, then the page is treated as `doc` page.

```
---
layout: doc
---
```

Doc Layout

Option `doc` is the default layout and it styles the whole Markdown content into "documentation" look. It works by wrapping whole content within `vp-doc` css class, and applying styles to elements underneath it.

Almost all generic elements such as `p`, or `h2` get special styling. Therefore, keep in mind that if you add any custom HTML inside a Markdown content, those will get affected by those styles as well.

It also provides documentation specific features listed below. These features are only enabled in this layout.

- Edit Link
- Prev Next Link
- Outline
- Carbon Ads

Page Layout

Option `page` is treated as "blank page". The Markdown will still be parsed, and all of the Markdown Extensions work as same as `doc` layout, but it wouldn't get any default stylings.

The page layout will let you style everything by you without VitePress theme affecting the markup. This is useful when you want to create your own custom page.

Note that even in this layout, sidebar will still show up if the page has a matching sidebar config.

Home Layout

Option `home` will generate templated "Homepage". In this layout, you can set extra options such as `hero` and `features` to customize the content further. Please visit Theme: Home Page for more details.

No Layout

If you don't want any layout, you can pass `layout: false` through frontmatter. This option is helpful if you want a fully-customizable landing page (without any sidebar, navbar, or footer by default).

Go to TOC

Nav

The Nav is the navigation bar displayed on top of the page. It contains the site title, global menu links, etc.

Site Title and Logo

By default, nav shows the title of the site referencing `config.title` value. If you would like to change what's displayed on nav, you may define custom text in `themeConfig.siteTitle` option.

```
export default {
  themeConfig: {
    siteTitle: 'My Custom Title'
  }
}
```

If you have a logo for your site, you can display it by passing in the path to the image. You should place the logo within `public` directly, and define the absolute path to it.

```
export default {
  themeConfig: {
    logo: '/my-logo.svg'
  }
}
```

When adding a logo, it gets displayed along with the site title. If your logo is all you need and if you would like to hide the site title text, set `false` to the `siteTitle` option.

```
export default {
  themeConfig: {
    logo: '/my-logo.svg',
    siteTitle: false
  }
}
```

You can also pass an object as logo if you want to add `alt` attribute or customize it based on dark/light mode. Refer `themeConfig.logo` for details.

Navigation Links

You may define `themeConfig.nav` option to add links to your nav.

```
export default {
  themeConfig: {
    nav: [
      { text: 'Guide', link: '/guide' },
      { text: 'Configs', link: '/configs' },
      { text: 'Changelog', link: 'https://github.com/...' }
    ]
  }
}
```

The `text` is the actual text displayed in nav, and the `link` is the link that will be navigated to when the text is clicked. For the link, set path to the actual file without `.md` prefix, and always start with `/`.

Nav links can also be dropdown menus. To do this, set `items` key on link option.

```
export default {
  themeConfig: {
    nav: [
      { text: 'Guide', link: '/guide' },
      {
        text: 'Dropdown Menu',
        items: [
          { text: 'Item A', link: '/item-1' },
          { text: 'Item B', link: '/item-2' },
          { text: 'Item C', link: '/item-3' }
        ]
      }
    ]
  }
}
```

Note that dropdown menu title (`Dropdown Menu` in the above example) can not have `link` property since it becomes a button to open dropdown dialog.

You may further add "sections" to the dropdown menu items as well by passing in more nested items.

```
export default {
  themeConfig: {
    nav: [
      { text: 'Guide', link: '/guide' },
      {
        text: 'Dropdown Menu',
        items: [
          {
            // Title for the section.
            text: 'Section A Title',
            items: [
              { text: 'Section A Item A', link: '...' },
              { text: 'Section B Item B', link: '...' }
            ]
          }
        ]
      },
      {
        text: 'Dropdown Menu',
        items: [
          {
            // You may also omit the title.
            items: [
              { text: 'Section A Item A', link: '...' },
              { text: 'Section B Item B', link: '...' }
            ]
          }
        ]
      }
    ]
  }
}
```

Customize link's "active" state

Nav menu items will be highlighted when the current page is under the matching path. if you would like to customize the path to be matched, define `activeMatch` property and regex as a string value.

```
export default {
  themeConfig: {
    nav: [
      // This link gets active state when the user is
      // on `/config/` path.
      {
        text: 'Guide',
        link: '/guide',
        activeMatch: '/config/'
      }
    ]
  }
}
```

::: warning `activeMatch` is expected to be a regex string, but you must define it as a string. We can't use actual RegExp object here because it isn't serializable during the build time. :::

Social Links

Refer `socialLinks`.

Go to TOC

Prev Next Link

You can customize the text of previous and next links. This is helpful if you want to show different text on previous/next links than what you have on your sidebar.

prev

- Type: `string`
- Details:

  Specify the text to show on the link to the previous page.

  If you don't set this in frontmatter, the text will be inferred from the sidebar config.
- Example:

```
---
prev: 'Get Started | Markdown'
---
```

next

- Type: `string`
- Details:

  Same as `prev` but for the next page.

Search

Documentation coming soon...

Go to TOC

Sidebar

The sidebar is the main navigation block for your documentation. You can configure the sidebar menu in `themeConfig.sidebar`.

```
export default {
  themeConfig: {
    sidebar: [
      {
        text: 'Guide',
        items: [
          { text: 'Introduction', link: '/introduction' },
          { text: 'Getting Started', link: '/getting-started' },
          ...
        ]
      }
    ]
  }
}
```

The Basics

The simplest form of the sidebar menu is passing in a single array of links. The first level item defines the "section" for the sidebar. It should contain `text`, which is the title of the section, and `items` which are the actual navigation links.

```
export default {
  themeConfig: {
    sidebar: [
      {
        text: 'Section Title A',
        items: [
          { text: 'Item A', link: '/item-a' },
          { text: 'Item B', link: '/item-b' },
          ...
        ]
      },
      {
        text: 'Section Title B',
        items: [
          { text: 'Item C', link: '/item-c' },
          { text: 'Item D', link: '/item-d' },
          ...
        ]
      }
    ]
  }
}
```

Each `link` should specify the path to the actual file starting with `/`. If you add trailing slash to the end of link, it will show `index.md` of the corresponding directory.

```js
export default {
  themeConfig: {
    sidebar: [
      {
        text: 'Guide',
        items: [
          // This shows `/guide/index.md` page.
          { text: 'Introduction', link: '/guide/' }
        ]
      }
    ]
  }
}
```

Multiple Sidebars

You may show different sidebar depending on the page path. For example, as shown on this site, you might want to create a separate sections of content in your documentation like "Guide" page and "Config" page.

To do so, first organize your pages into directories for each desired section:

```
.
├─ guide/
│  ├─ index.md
│  ├─ one.md
│  └─ two.md
└─ config/
   ├─ index.md
   ├─ three.md
   └─ four.md
```

Then, update your configuration to define your sidebar for each section. This time, you should pass an object instead of an array.

```js
export default {
  themeConfig: {
    sidebar: {
      // This sidebar gets displayed when user is
      // under `guide` directory.
      '/guide/': [
        {
          text: 'Guide',
          items: [
            // This shows `/guide/index.md` page.
            { text: 'Index', link: '/guide/' }, // /guide/index.md
            { text: 'One', link: '/guide/one' }, // /guide/one.md
            { text: 'Two', link: '/guide/two' } // /guide/two.md
          ]
        }
      ],

      // This sidebar gets displayed when user is
      // under `config` directory.
      '/config/': [
        {
          text: 'Config',
          items: [
            // This shows `/config/index.md` page.
```

```
              { text: 'Index', link: '/config/' }, // /config/index.md
              { text: 'Three', link: '/config/three' }, // /config/three.md
              { text: 'Four', link: '/config/four' } // /config/four.md
            ]
          }
        ]
      }
    }
  }
```

Collapsible Sidebar Groups

By adding `collapsible` option to the sidebar group, it shows a toggle button to hide/show each section.

```
export default {
  themeConfig: {
    sidebar: [
      {
        text: 'Section Title A',
        collapsible: true,
        items: [...]
      },
      {
        text: 'Section Title B',
        collapsible: true,
        items: [...]
      }
    ]
  }
}
```

All sections are "open" by default. If you would like them to be "closed" on initial page load, set `collapsed` option to `true`.

```
export default {
  themeConfig: {
    sidebar: [
      {
        text: 'Section Title A',
        collapsible: true,
        collapsed: true,
        items: [...]
      }
    ]
  }
}
```

Go to TOC

Team Page

If you would like to introduce your team, you may use Team components to construct the Team Page. There are two ways of using these components. One is to embed it in doc page, and another is to create a full Team Page.

Show team members in a page

You may use `<VPTeamMembers>` component exposed from `vitepress/theme` to display a list of team members on any page.

```
<script setup>
import { VPTeamMembers } from 'vitepress/theme'

const members = [
  {
    avatar: 'https://www.github.com/yyx990803.png',
    name: 'Evan You',
    title: 'Creator',
    links: [
      { icon: 'github', link: 'https://github.com/yyx990803' },
      { icon: 'twitter', link: 'https://twitter.com/youyuxi' }
    ]
  },
  ...
]
</script>

# Our Team

Say hello to our awesome team.

<VPTeamMembers size="small" :members="members" />
```

The above will display a team member in card looking element. It should display something similar to below.

`<VPTeamMembers>` component comes in 2 different sizes, `small` and `medium`. While it boils down to your preference, usually `small` size should fit better when used in doc page. Also, you may add more properties to each member such as adding "description" or "sponsor" button. Learn more about it in `<VPTeamMembers>`.

Embedding team members in doc page is good for small size team where having dedicated full team page might be too much, or introducing partial members as a reference to documentation context.

If you have large number of members, or simply would like to have more space to show team members, consider creating a full team page.

Create a full Team Page

Instead of adding team members to doc page, you may also create a full Team Page, similar to how you can create a custom Home Page.

To create a team page, first, create a new md file. The file name doesn't matter, but here lets call it `team.md`. In this file, set frontmatter option `layout: page`, and then you may compose your page structure using `TeamPage` components.

```
---
layout: page
---
<script setup>
import {
  VPTeamPage,
  VPTeamPageTitle,
  VPTeamMembers
} from 'vitepress/theme'

const members = [
  {
    avatar: 'https://www.github.com/yyx990803.png',
    name: 'Evan You',
    title: 'Creator',
    links: [
      { icon: 'github', link: 'https://github.com/yyx990803' },
      { icon: 'twitter', link: 'https://twitter.com/youyuxi' }
    ]
  },
  ...
]
</script>

<VPTeamPage>
  <VPTeamPageTitle>
    <template #title>
      Our Team
    </template>
    <template #lead>
      The development of VitePress is guided by an international
      team, some of whom have chosen to be featured below.
    </template>
  </VPTeamPageTitle>
  <VPTeamMembers
    :members="members"
  />
</VPTeamPage>
```

When creating a full team page, remember to wrap all components with `<VPTeamPage>` component. This component will ensure all nested team related components get the proper layout structure like spacings.

`<VPPageTitle>` component adds the page title section. The title being `<h1>` heading. Use `#title` and `#lead` slot to document about your team.

`<VPMembers>` works as same as when used in a doc page. It will display list of members.

Add sections to divide team members

You may add "sections" to the team page. For example, you may have different types of team members such as Core Team Members and Community Partners. You can divide these members into sections to better explain the roles of each group.

To do so, add `<VPTeamPageSection>` component to the `team.md` file we created previously.

```
---
layout: page
---
<script setup>
import {
  VPTeamPage,
  VPTeamPageTitle,
  VPTeamMembers,
  VPTeamPageSection
} from 'vitepress/theme'

const coreMembers = [...]
const partners = [...]
</script>

<VPTeamPage>
  <VPTeamPageTitle>
    <template #title>Our Team</template>
    <template #lead>...</template>
  </VPTeamPageTitle>
  <VPTeamMembers size="medium" :members="coreMembers" />
  <VPTeamPageSection>
    <template #title>Partners</template>
    <template #lead>...</template>
    <template #members>
      <VPTeamMembers size="small" :members="partners" />
    </template>
  </VPTeamPageSection>
</VPTeamPage>
```

The `<VPTeamPageSection>` component can have `#title` and `#lead` slot similar to `VPTeamPageTitle` component, and also `#members` slot for displaying team members.

Remember to put in `<VPTeamMembers>` component within `#members` slot.

`<VPTeamMembers>`

The `<VPTeamMembers>` component displays a given list of members.

```
<VPTeamMembers
  size="medium"
  :members="[
    { avatar: '...', name: '...' },
    { avatar: '...', name: '...' },
    ...
  ]"
/>
```

```
interface Props {
  // Size of each members. Defaults to `medium`.
  size?: 'small' | 'medium'

  // List of members to display.
  members: TeamMember[]
}

interface TeamMember {
  // Avatar image for the member.
  avatar: string

  // Name of the member.
  name: string

  // Title to be shown below member's name.
  // e.g. Developer, Software Engineer, etc.
  title?: string

  // Organization that the member belongs.
  org?: string

  // URL for the organization.
  orgLink?: string

  // Description for the member.
  desc?: string

  // Social links. e.g. GitHub, Twitter, etc. You may pass in
  // the Social Links object here.
  // See: https://vitepress.vuejs.org/config/theme-configs.html#sociallinks
  links?: SocialLink[]

  // URL for the sponsor page for the member.
  sponsor?: string
}
```

`<VPTeamPage>`

The root component when creating a full team page. It only accepts a single slot. It will style all passed in team related components.

`<VPTeamPageTitle>`

Adds "title" section of the page. Best use at the very beginning under `<VPTeamPage>`. It accepts `#title` and `#lead` slot.

```
<VPTeamPage>
  <VPTeamPageTitle>
    <template #title>
      Our Team
    </template>
    <template #lead>
      The development of VitePress is guided by an international
      team, some of whom have chosen to be featured below.
    </template>
  </VPTeamPageTitle>
</VPTeamPage>
```

`<VPTeamPageSection>`

Creates a "section" with in team page. It accepts `#title`, `#lead`, and `#members` slot. You may add as many sections as you like inside `<VPTeamPage>`.

```
<VPTeamPage>
  ...
  <VPTeamPageSection>
    <template #title>Partners</template>
    <template #lead>Lorem ipsum...</template>
    <template #members>
      <VPTeamMembers :members="data" />
    </template>
  </VPTeamPageSection>
</VPTeamPage>
```

Go to TOC

Using Vue in Markdown

In VitePress, each markdown file is compiled into HTML and then processed as a Vue Single-File Component. This means you can use any Vue features inside the markdown, including dynamic templating, using Vue components, or arbitrary in-page Vue component logic by adding a `<script>` tag.

It is also important to know that VitePress leverages Vue 3's compiler to automatically detect and optimize the purely static parts of the markdown. Static contents are optimized into single placeholder nodes and eliminated from the page's JavaScript payload. They are also skipped during client-side hydration. In short, you only pay for the dynamic parts on any given page.

Templating

Interpolation

Each Markdown file is first compiled into HTML and then passed on as a Vue component to the Vite process pipeline. This means you can use Vue-style interpolation in text:

Input

```
{{ 1 + 1 }}
```

Output

```
{{ 1 + 1 }}
```

Directives

Directives also work:

Input

```
<span v-for="i in 3">{{ i }}</span>
```

Output

```
{{ i }}
```

Access to Site & Page Data

You can use the `useData` helper in a `<script>` block and expose the data to the page.

Input

```
<script setup>
import { useData } from 'vitepress'

const { page } = useData()
```

```
</script>

<pre>{{ page }}</pre>
```

Output

```
{
  "path": "/using-vue.html",
  "title": "Using Vue in Markdown",
  "frontmatter": {}
}
```

Escaping

By default, fenced code blocks are automatically wrapped with `v-pre`. To display raw mustaches or Vue-specific syntax inside inline code snippets or plain text, you need to wrap a paragraph with the `v-pre` custom container:

Input

```
::: v-pre
`{{ This will be displayed as-is }}`
:::
```

Output

::: v-pre `{{ This will be displayed as-is }}` :::

Using Components

When you need to have more flexibility, VitePress allows you to extend your authoring toolbox with your own Vue Components.

Importing components in markdown

If your components are going to be used in only a few places, the recommended way to use them is to importing the components in the file where it is used.

```
<script setup>
import CustomComponent from '../components/CustomComponent.vue'
</script>

# Docs

This is a .md using a custom component

<CustomComponent />

## More docs

...
```

Registering global components in the theme

If the components are going to be used across several pages in the docs, they can be registered globally in the theme (or as part of extending the default VitePress theme). Check out the Theming Guide for more information.

In `.vitepress/theme/index.js`, the `enhanceApp` function receives the Vue `app` instance so you can register components as you would do in a regular Vue application.

```
import DefaultTheme from 'vitepress/theme'

export default {
  ...DefaultTheme,
  enhanceApp(ctx) {
    DefaultTheme.enhanceApp(ctx)
    ctx.app.component('VueClickAwayExample', VueClickAwayExample)
  }
}
```

Later in your markdown files, the component can be interleaved between the content

```
# Vue Click Away

<VueClickAwayExample />
```

::: warning IMPORTANT Make sure a custom component's name either contains a hyphen or is in PascalCase. Otherwise, it will be treated as an inline element and wrapped inside a `<p>` tag, which will lead to hydration mismatch because `<p>` does not allow block elements to be placed inside it. :::

Using Components In Headers

You can use Vue components in the headers, but note the difference between the following syntaxes:

Markdown	Output HTML	Parsed Header
`# text <Tag/>`	`<h1>text <Tag/></h1>`	`text`
``# text `<Tag/>` ``	`<h1>text <code>&lt;Tag/&gt;</code></h1>`	`text <Tag/>`

The HTML wrapped by `<code>` will be displayed as-is; only the HTML that is **not** wrapped will be parsed by Vue.

::: tip The output HTML is accomplished by markdown-it, while the parsed headers are handled by VitePress (and used for both the sidebar and document title). :::

Using CSS Pre-processors

VitePress has built-in support for CSS pre-processors: `.scss`, `.sass`, `.less`, `.styl` and `.stylus` files. There is no need to install Vite-specific plugins for them, but the corresponding pre-processor itself must be installed:

```
# .scss and .sass
npm install -D sass

# .less
npm install -D less

# .styl and .stylus
npm install -D stylus
```

Then you can use the following in Markdown and theme components:

```
<style lang="sass">
.title
  font-size: 20px
</style>
```

Script & Style Hoisting

Sometimes you may need to apply some JavaScript or CSS only to the current page. In those cases, you can directly write root-level `<script>` or `<style>` blocks in the Markdown file. These will be hoisted out of the compiled HTML and used as the `<script>` and `<style>` blocks for the resulting Vue single-file component:

Built-In Components

VitePress provides Built-In Vue Components like `ClientOnly` and `OutboundLink`, check out the Global Component Guide for more information.

Also see:

- Using Components In Headers

Browser API Access Restrictions

Because VitePress applications are server-rendered in Node.js when generating static builds, any Vue usage must conform to the universal code requirements. In short, make sure to only access Browser / DOM APIs in `beforeMount` or `mounted` hooks.

If you are using or demoing components that are not SSR-friendly (for example, contain custom directives), you can wrap them inside the built-in `<ClientOnly>` component:

```
<ClientOnly>
  <NonSSRFriendlyComponent />
</ClientOnly>
```

Note this does not fix components or libraries that access Browser APIs **on import**. To use code that assumes a browser environment on import, you need to dynamically import them in proper lifecycle hooks:

```
<script>
export default {
  mounted() {
    import('./lib-that-access-window-on-import').then((module) => {
      // use code
    })
  }
}
</script>
```

If your module `export default` a Vue component, you can register it dynamically:

```
<template>
  <component
    v-if="dynamicComponent"
    :is="dynamicComponent">
  </component>
</template>

<script>
export default {
  data() {
    return {
      dynamicComponent: null
    }
  },

  mounted() {
    import('./lib-that-access-window-on-import').then((module) => {
      this.dynamicComponent = module.default
    })
  }
}
</script>
```

Also see:

- Vue.js > Dynamic Components

Go to TOC

What is VitePress?

VitePress is VuePress' little brother, built on top of Vite.

::: warning VitePress is currently in `alpha` status. It is already suitable for out-of-the-box documentation use, but the config and theming API may still change between minor releases. :::

Motivation

We love VuePress v1, but being built on top of Webpack, the time it takes to spin up the dev server for a simple doc site with a few pages is just becoming unbearable. Even HMR updates can take up to seconds to reflect in the browser!

Fundamentally, this is because VuePress v1 is a Webpack app under the hood. Even with just two pages, it's a full on Webpack project (including all the theme source files) being compiled. It gets even worse when the project has many pages - every page must first be fully compiled before the server can even display anything!

Incidentally, Vite solves these problems really well: nearly instant server start, an on-demand compilation that only compiles the page being served, and lightning-fast HMR. Plus, there are a few additional design issues I have noted in VuePress v1 over time but never had the time to fix due to the amount of refactoring it would require.

Now, with Vite and Vue 3, it is time to rethink what a "Vue-powered static site generator" can really be.

Improvements over VuePress v1

There're couple of things that are improved from VuePress v1....

It uses Vue 3

Leverages Vue 3's improved template static analysis to stringify static content as much as possible. Static content is sent as string literals instead of JavaScript render function code - the JS payload is therefore much cheaper to parse, and hydration also becomes faster.

Note the optimization is applied while still allowing the user to freely mix Vue components inside markdown content - the compiler does the static/dynamic separation for you automatically and you never need to think about it.

It uses Vite under the hood

- Faster dev server start
- Faster hot updates
- Faster build (uses Rollup internally)

Lighter page weight

Vue 3 tree-shaking + Rollup code splitting

- Does not ship metadata for every page on every request. This decouples page weight from total number of pages. Only the current page's metadata is sent. Client side navigation fetches the new page's component and metadata together.
- Does not use vue-router because the need of VitePress is very simple and specific - a simple custom router (under 200 LOC) is used instead.

Other differences

VitePress is more opinionated and less configurable: VitePress aims to scale back the complexity in the current VuePress and restart from its minimalist roots.

VitePress is future oriented: VitePress only targets browsers that support native ES module imports. It encourages the use of native JavaScript without transpilation, and CSS variables for theming.

Will this become the next vuepress in the future?

We already have vuepress-next, which would be the next major version of VuePress. It also makes lots of improvements over VuePress v1, and also supports Vite now.

VitePress is not compatible with the current VuePress ecosystem (mostly themes and plugins). The overall idea is that VitePress will have a drastically more minimal theming API (preferring JavaScript APIs instead of file layout conventions) and likely no plugins (all customization is done in themes).

There is an ongoing discussion about this topic. If you're curious, please leave your thoughts!

Go to TOC

Colophon

This book is created by using the following sources:

- Vitepress - English
- GitHub source: vuejs/vitepress/docs/guide
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf