

目录

- [Introduction](#)
- **Getting Started**
 - [Installation](#)
 - [Configuration](#)
 - [Playground](#)
 - [TypeScript Support](#)
- **Guides**
 - [Pages](#)
- **Docs**
 - [Create a doc](#)
 - **Sidebar**
 - [Sidebar items](#)
 - [Autogenerated](#)
 - [Using multiple sidebars](#)
 - [Versioning](#)
 - [Docs Multi-instance](#)
 - [Blog](#)
- **Markdown Features**
 - [MDX and React](#)
 - [Tabs](#)
 - [Code blocks](#)
 - [Admonitions](#)
 - [Headings and Table of contents](#)
 - [Assets](#)
 - [Markdown links](#)
 - [MDX Plugins](#)
 - [Math Equations](#)
 - [Diagrams](#)
 - [Head metadata](#)
 - [Styling and Layout](#)
 - [Swizzling](#)

[Static Assets](#)

[Search](#)

[Browser support](#)

[SEO](#)

[Using Plugins](#)

[Deployment](#)

» **Internationalization**

[Tutorial](#)

[Using Git](#)

[Using Crowdin](#)

[What's next?](#)

» **Advanced Guides**

[Architecture](#)

[Plugins](#)

[Routing](#)

[Static site generation](#)

[Client architecture](#)

» **Upgrading**

[To Docusaurus v3](#)

» **To Docusaurus v2**

[Overview](#)

[Automated migration](#)

[Manual migration](#)

[Versioned sites](#)

[Translated sites](#)

Introduction

- ⚡ Docusaurus will help you ship a **beautiful documentation site in no time**.
- 💵 Building a custom tech stack is expensive. Instead, **focus on your content** and just write Markdown files.
- 💥 Ready for more? Use **advanced features** like versioning, i18n, search and theme customizations.
- 🎨 Check the **best Docusaurus sites** for inspiration.
- 🤖 Docusaurus is a **static-site generator**. It builds a **single-page application** with fast client-side navigation, leveraging the full power of **React** to make your site interactive. It provides out-of-the-box **documentation features** but can be used to create **any kind of site** (personal website, product, blog, marketing landing pages, etc).



Fast Track ⏰

Understand Docusaurus in **5 minutes** by playing!

Create a new Docusaurus site and follow the **very short** embedded tutorial.

Install [Node.js](#) and create a new Docusaurus site:

```
npx create-docusaurus@latest my-website classic
```

Start the site:

```
cd my-website  
npx docusaurus start
```

Open <http://localhost:3000> and follow the tutorial.



TIP

Use [docusaurus.new](#) to test Docusaurus immediately in your browser!

Or read the [5-minute tutorial](#) online.

Docusaurus: Documentation Made Easy

In this presentation at [Algolia Community Event](#), [Meta Open Source team](#) shared a brief walk-through of Docusaurus. They covered how to get started with the project, enable plugins, and set up functionalities like documentation and blogging.



Migrating from v1

Docusaurus v2+ has been a total rewrite from Docusaurus v1, taking advantage of a completely modernized toolchain. After [v2's official release](#), we highly encourage you to **use Docusaurus v2+ over Docusaurus v1**, as Docusaurus v1 has been deprecated.

A [lot of users](#) are already using Docusaurus v2+ ([trends](#)).

Use Docusaurus v2+ if:

- You want a modern Jamstack documentation site
- You want a single-page application (SPA) with client-side routing
- You want the full power of React and MDX
- You do not need support for IE11

Use Docusaurus v1 if:

- ✗ You don't want a single-page application (SPA)
- ✗ You need support for IE11 (...do you? IE [has already reached end-of-life](#) and is no longer officially supported)

For existing v1 users that are seeking to upgrade to v2+, you can follow our [migration guides](#).

Features

Docusaurus is built with high attention to the developer and contributor experience.

-  **Built with ❤️ and React:**
 - Extend and customize with React
 - Gain full control of your site's browsing experience by providing your own React components
-  **Pluggable:**
 - Bootstrap your site with a basic template, then use advanced features and plugins
 - Open source your plugins to share with the community
-  **Developer experience:**
 - Start writing your docs right now
 - Universal configuration entry point to make it more maintainable by contributors
 - Hot reloading with lightning-fast incremental build on changes
 - Route-based code and data splitting
 - Publish to GitHub Pages, Netlify, Vercel, and other deployment services with ease

Our shared goal—to help your users quickly find what they need and understand your products better. We share our best practices to help you build your docs site right and well.

-  **SEO friendly:**
 - HTML files are statically generated for every possible path.
 - Page-specific SEO to help your users land on your official docs directly relating their problems at hand.
-  **Powered by MDX:**
 - Write interactive components via JSX and React embedded in Markdown.
 - Share your code in live editors to get your users to love your products on the spot.
-  **Search:** Your full site is searchable.
-  **Document Versioning:** Helps you keep documentation in sync with project releases.
-  **Internationalization (i18n):** Translate your site in multiple locales.

Docusaurus v2+ is born to be compassionately accessible to all your users, and lightning-fast.

-  **Lightning-fast.** Docusaurus v2+ follows the [PRPL Pattern](#) that makes sure your content loads blazing fast.
-  **Accessible.** Attention to accessibility, making your site equally accessible to all users.

Design principles

- **Little to learn.** Docusaurus should be easy to learn and use as the API is quite small. Most things will still be achievable by users, even if it takes them more code and more time to write. Not having abstractions is better than having the wrong abstractions, and we don't want users to have to hack around the wrong abstractions. Mandatory talk—[Minimal API Surface Area](#).
- **Intuitive.** Users will not feel overwhelmed when looking at the project directory of a Docusaurus project or adding new features. It should look intuitive and easy to build on top of, using approaches they are familiar with.
- **Layered architecture.** The separations of concerns between each layer of our stack (content/theming/styling) should be clear—well-abstracted and modular.
- **Sensible defaults.** Common and popular performance optimizations and configurations will be done for users but they are given the option to override them.
- **No vendor lock-in.** Users are not required to use the default plugins or CSS, although they are highly encouraged to. Certain core infrastructures like React Loadable and React Router cannot be swapped because we do default performance optimization on them, but not higher-level ones. Choice of Markdown engines, CSS frameworks, CSS methodology, and other architectures will be entirely up to users.

We believe that, as developers, knowing how a library works helps us become better at using it. Hence we're dedicating effort to explaining the architecture and various components of Docusaurus with the hope that users reading it will gain a deeper understanding of the tool and be even more proficient in using it.

Comparison with other tools

Across all static site generators, Docusaurus has a unique focus on documentation sites and has many out-of-the-box features.

We've also studied other main static site generators and would like to share our insights on the comparison, hopefully helping you navigate through the prismatic choices out there.

Gatsby

[Gatsby](#) is packed with a lot of features, has a rich ecosystem of plugins, and is capable of doing everything that Docusaurus does. Naturally, that comes at a cost of a higher learning curve. Gatsby does many things well and is suitable for building many types of websites. On the other hand, Docusaurus tries to do one thing super well - be the best tool for writing and publishing content.

GraphQL is also pretty core to Gatsby, although you don't necessarily need GraphQL to build a Gatsby site. In most cases when building static websites, you won't need the flexibility that GraphQL provides.

Many aspects of Docusaurus v2+ were inspired by the best things about Gatsby and it's a great alternative.

[Docz](#) is a Gatsby theme to build documentation websites. It is currently less featured than Docusaurus.

Next.js

[Next.js](#) is another very popular hybrid React framework. It can help you build a good documentation website, but it is not opinionated toward the documentation use-case, and it will require a lot more work to implement what Docusaurus provides out-of-the-box.

[Nextra](#) is an opinionated static site generator built on top of Next.js. It is currently less featured than Docusaurus.

VitePress

[VitePress](#) has many similarities with Docusaurus - both focus heavily on content-centric websites and provides tailored documentation features out of the box. However, VitePress is powered by Vue, while Docusaurus is powered by React. If you want a Vue-based solution, VitePress would be a decent choice.

MkDocs

[MkDocs](#) is a popular Python static site generator with value propositions similar to Docusaurus.

It is a good option if you don't need a single-page application and don't plan to leverage React.

[Material for MkDocs](#) is a beautiful theme.

Docsify

[Docsify](#) makes it easy to create a documentation website, but is not a static-site generator and is not SEO friendly.

GitBook

[GitBook](#) has a very clean design and has been used by many open source projects. With its focus shifting towards a commercial product rather than an open-source tool, many of its requirements no longer fit the needs of open source projects' documentation sites. As a result, many have turned to other products. You may read about Redux's switch to Docusaurus [here](#).

Currently, GitBook is only free for open-source and non-profit teams. Docusaurus is free for everyone.

Jekyll

[Jekyll](#) is one of the most mature static site generators around and has been a great tool to use — in fact, before Docusaurus, most of Facebook's Open Source websites are/were built on Jekyll! It is extremely simple to get started. We want to bring a similar developer experience as building a static site with Jekyll.

In comparison with statically generated HTML and interactivity added using `<script />` tags, Docusaurus sites are React apps. Using modern JavaScript ecosystem tooling, we hope to set new standards on doc sites' performance, asset building pipeline and optimizations, and ease to set up.

Rspress

[Rspress](#) is a fast static site generator based on Rspack, a Rust-based bundler. It supports content writing with MDX (Markdown with React components), integrated text search, multilingual support (i18n), and extensibility through plugins. Designed for creating elegant documentation and static websites, Rspress produces static HTML files that are easy to deploy.

Rspress and Docusaurus are quite similar. They both have their pros and cons. Rspress was created more recently and benefits from a modern infrastructure that enables faster site builds. Docusaurus stands out for its maturity, comprehensive feature set, flexibility, and strong community. It is also [modernizing its infrastructure](#) regularly to remain competitive in terms of performance.

Staying informed

- [GitHub](#)
- [X](#)
- [Blog](#)
- [Discord](#)

Something missing?

If you find issues with the documentation or have suggestions on how to improve the documentation or the project in general, please [file an issue](#) for us, or send a tweet mentioning the [@docusaurus X](#) account.

For new feature requests, you can create a post on our [feature requests board \(Canny\)](#), which is a handy tool for road-mapping and allows for sorting by upvotes, which gives the core team a better indicator of what features are in high demand, as compared to GitHub issues which are harder to triage. Refrain from making a Pull Request for new features (especially large ones) as someone might already be working on it or will be part of our roadmap. Talk to us first!

Installation

Docusaurus consists of a set of npm [packages](#).



TIP

Use the [Fast Track](#) to understand Docusaurus in **5 minutes** !

Use [docusaurus.new](#) to test Docusaurus immediately in your browser!

Requirements

- [Node.js](#) version 18.0 or above (which can be checked by running `node -v`). You can use [nvm](#) to manage multiple Node.js versions on a single machine.
 - When installing Node.js, it is recommended to check all checkboxes related to dependencies.

Scaffold project website

The easiest way to install Docusaurus is to use the [create-docusaurus](#) command line tool that helps you scaffold a skeleton Docusaurus website. You can run this command anywhere in a new empty repository or within an existing repository, it will create a new directory containing the scaffolded files.

```
npx create-docusaurus@latest my-website classic
```

We recommend the `classic` template so that you can get started quickly, and it contains features found in Docusaurus 1. The `classic` template contains `@docusaurus/preset-classic` which includes standard documentation, a blog, custom pages, and a CSS framework (with dark mode support). You can get up and running extremely quickly with the classic template and customize things later on when you have gained more familiarity with Docusaurus.

You can also use the template's TypeScript variant by passing the `--typescript` flag. See [TypeScript support](#) for more information.

```
npx create-docusaurus@latest my-website classic --typescript
```



META-ONLY

If you are setting up a new Docusaurus website for a Meta open source project, run this command inside an internal repository, which comes with some useful Meta-specific defaults:

```
scarf static-docs-bootstrap
```

▼ Alternative installation commands

Run `npx create-docusaurus@latest --help`, or check out its [API docs](#) for more information about all available flags.

Project structure

Assuming you chose the classic template and named your site `my-website`, you will see the following files generated under a new directory `my-website/`:

```
my-website
├── blog
│   ├── 2019-05-28-hola.md
│   ├── 2019-05-29-hello-world.md
│   └── 2020-05-30-welcome.md
├── docs
│   ├── doc1.md
│   ├── doc2.md
│   ├── doc3.md
│   └── mdx.md
└── src
    ├── css
    │   └── custom.css
    └── pages
        ├── styles.module.css
        └── index.js
└── static
    └── img
├── docusaurus.config.js
├── package.json
├── README.md
└── sidebars.js
└── yarn.lock
```

Project structure rundown

- `/blog/` - Contains the blog Markdown files. You can delete the directory if you've disabled the blog plugin, or you can change its name after setting the `path` option. More details can be found in the

blog guide

- `/docs/` - Contains the Markdown files for the docs. Customize the order of the docs sidebar in `sidebars.js`. You can delete the directory if you've disabled the docs plugin, or you can change its name after setting the `path` option. More details can be found in the [docs guide](#)
- `/src/` - Non-documentation files like pages or custom React components. You don't have to strictly put your non-documentation files here, but putting them under a centralized directory makes it easier to specify in case you need to do some sort of linting/processing
 - `/src/pages` - Any JSX/TSX/MDX file within this directory will be converted into a website page. More details can be found in the [pages guide](#)
- `/static/` - Static directory. Any contents inside here will be copied into the root of the final `build` directory
- `/docusaurus.config.js` - A config file containing the site configuration. This is the equivalent of `siteConfig.js` in Docusaurus v1
- `/package.json` - A Docusaurus website is a React app. You can install and use any npm packages you like in them
- `/sidebars.js` - Used by the documentation to specify the order of documents in the sidebar

Monorepos

If you are using Docusaurus for documentation of an existing project, a monorepo may be the solution for you. Monorepos allow you to share dependencies between similar projects. For example, your website may use your local packages to showcase latest features instead of depending on a released version. Then, your contributors can update the docs as they implement features. An example monorepo folder structure is below:

```
my-monorepo
├── package-a # Another package, your actual project
|   ├── src
|   └── package.json # Package A's dependencies
├── website # Docusaurus root
|   ├── docs
|   └── src
|       └── package.json # Docusaurus' dependencies
└── package.json # Monorepo's shared dependencies
```

In this case, you should run `npx create-docusaurus` within the `./my-monorepo` folder.

If you're using a hosting provider such as Netlify or Vercel, you will need to change the `Base directory` of the site to where your Docusaurus root is. In this case, that would be `./website`. Read more about configuring ignore commands in the [deployment docs](#).

Read more about monorepos in the [Yarn documentation](#) (Yarn is not the only way to set up a monorepo, but it's a common solution), or checkout [Docusaurus](#) and [Jest](#) for some real-world examples.

Running the development server

To preview your changes as you edit the files, you can run a local development server that will serve your website and reflect the latest changes.

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
cd my-website
npm run start
```

By default, a browser window will open at <http://localhost:3000>.

Congratulations! You have just created your first Docusaurus site! Browse around the site to see what's available.

Build

Docusaurus is a modern static website generator so we need to build the website into a directory of static contents and put it on a web server so that it can be viewed. To build the website:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run build
```

and contents will be generated within the `/build` directory, which can be copied to any static file hosting service like [GitHub pages](#), [Vercel](#) or [Netlify](#). Check out the docs on [deployment](#) for more details.

Updating your Docusaurus version

There are many ways to update your Docusaurus version. One guaranteed way is to manually change the version number in `package.json` to the desired version. Note that all `@docusaurus/-namespaced` packages should be using the same version.

package.json

```
{  
  "dependencies": {  
    "@docusaurus/core": "3.8.1",  
    "@docusaurus/preset-classic": "3.8.1",  
    // ...  
  }  
}
```

Then, in the directory containing `package.json`, run your package manager's install command:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install
```



TIP

`npm install` may report several vulnerabilities and recommend running `npm audit` to address them. Typically, these reported vulnerabilities, such as RegExp DOS vulnerabilities, are harmless and can be safely ignored. Also read this article, which reflects our thinking: [npm audit: Broken by Design](#).

To check that the update occurred successfully, run:

```
npx docusaurus --version
```

You should see the correct version as output.

Alternatively, if you are using Yarn, you can do:

```
yarn add @docusaurus/core @docusaurus/preset-classic
```



TIP

Use new unreleased features of Docusaurus with the [@canary npm dist tag](#)

Problems?

Ask for help on [Stack Overflow](#), on our [GitHub repository](#), our [Discord server](#), or [X](#).

Configuration

! INFO

Check the [docusaurus.config.js API reference](#) for an exhaustive list of options.

Docusaurus has a unique take on configurations. We encourage you to congregate information about your site into one place. We guard the fields of this file and facilitate making this data object accessible across your site.

Keeping a well-maintained `docusaurus.config.js` helps you, your collaborators, and your open source contributors to be able to focus on documentation while still being able to customize the site.

Syntax to declare `docusaurus.config.js`

The `docusaurus.config.js` file is run in Node.js and should export either:

- a **config object**
- a **function** that creates the config object

! INFO

The `docusaurus.config.js` file supports:

- [ES Modules](#)
- [CommonJS](#)
- [TypeScript](#)

Constraints:

- **Required:** use `export default /* your config */` (or `module.exports`) to export your Docusaurus config
- **Optional:** use `import Lib from 'lib'` (or `require('lib')`) to import Node.js packages

Docusaurus gives us the ability to declare its configuration in various **equivalent ways**, and all the following config examples lead to the exact same result:

`docusaurus.config.js`

```
export default {  
  title: 'Docusaurus',
```

```
url: 'https://docusaurus.io',
// your site config ...
};
```

docusaurus.config.js

```
module.exports = {
  title: 'Docusaurus',
  url: 'https://docusaurus.io',
  // your site config ...
};
```

docusaurus.config.ts

```
import type {Config} from '@docusaurus/types';

export default {
  title: 'Docusaurus',
  url: 'https://docusaurus.io',
  // your site config ...
} satisfies Config;
```

docusaurus.config.js

```
const config = {
  title: 'Docusaurus',
  url: 'https://docusaurus.io',
  // your site config ...
};

export default config;
```

docusaurus.config.js

```
export default function configCreator() {
  return {
    title: 'Docusaurus',
    url: 'https://docusaurus.io',
    // your site config ...
  };
}
```

docusaurus.config.js

```
export default async function createConfigAsync() {
  return {
    title: 'Docusaurus',
    url: 'https://docusaurus.io',
    // your site config ...
  };
}
```

USING ESM-ONLY PACKAGES

Using an async config creator can be useful to import ESM-only modules (notably most Remark plugins). It is possible to import such modules thanks to dynamic imports:

docusaurus.config.js

```
export default async function createConfigAsync() {
  // Use a dynamic import instead of require('esm-lib')
  const lib = await import('lib');

  return {
    title: 'Docusaurus',
    url: 'https://docusaurus.io',
    // rest of your site config...
  };
}
```

What goes into a `docusaurus.config.js`?

You should not have to write your `docusaurus.config.js` from scratch even if you are developing your site. All templates come with a `docusaurus.config.js` that includes defaults for the common options.

However, it can be helpful if you have a high-level understanding of how the configurations are designed and implemented.

The high-level overview of Docusaurus configuration can be categorized into:

Site metadata

Site metadata contains the essential global metadata such as `title`, `url`, `baseUrl`, and `favicon`.

They are used in several places such as your site's title and headings, browser tab icon, social sharing (Facebook, X) information or even to generate the correct path to serve your static files.

Deployment configurations

Deployment configurations such as `projectName`, `organizationName`, and optionally `deploymentBranch` are used when you deploy your site with the `deploy` command.

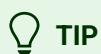
It is recommended to check the [deployment docs](#) for more information.

Theme, plugin, and preset configurations

List the `themes`, `plugins`, and `presets` for your site in the `themes`, `plugins`, and `presets` fields, respectively. These are typically npm packages:

`docusaurus.config.js`

```
export default {
  // ...
  plugins: [
    '@docusaurus/plugin-content-blog',
    '@docusaurus/plugin-content-pages',
  ],
  themes: ['@docusaurus/theme-classic'],
};
```



Docusaurus supports [module shorthands](#), allowing you to simplify the above configuration as:

`docusaurus.config.js`

```
export default {
  // ...
  plugins: ['content-blog', 'content-pages'],
  themes: ['classic'],
};
```

They can also be loaded from local directories:

`docusaurus.config.js`

```
import path from 'path';

export default {
  // ...
  themes: [path.resolve(__dirname, '/path/to/docusaurus-local-theme')],
};
```

To specify options for a plugin or theme, replace the name of the plugin or theme in the config file with an array containing the name and an options object:

docusaurus.config.js

```
export default {
  // ...
  plugins: [
    [
      'content-blog',
      {
        path: 'blog',
        routebasePath: 'blog',
        include: ['*.md', '*.mdx'],
        // ...
      },
    ],
    'content-pages',
  ],
};
```

To specify options for a plugin or theme that is bundled in a preset, pass the options through the `presets` field. In this example, `docs` refers to `@docusaurus/plugin-content-docs` and `theme` refers to `@docusaurus/theme-classic`.

docusaurus.config.js

```
export default {
  // ...
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          sidebarPath: './sidebars.js',
        },
        theme: {
          customCss: ['./src/css/custom.css'],
        }
      }
    ]
};
```

```
  },
  ],
],
};

};
```



TIP

The `presets: [['classic', {...}]]` shorthand works as well.

For further help configuring themes, plugins, and presets, see [Using Plugins](#).

Custom configurations

Docusaurus guards `docusaurus.config.js` from unknown fields. To add custom fields, define them in `customFields`.

Example:

```
docusaurus.config.js
```

```
export default {
  // ...
  customFields: {
    image: '',
    keywords: [],
  },
  // ...
};
```

Accessing configuration from components

Your configuration object will be made available to all the components of your site. And you may access them via React context as `siteConfig`.

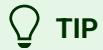
Basic example:

```
import React from 'react';
import useDocusaurusContext from '@docusaurus/useDocusaurusContext';

const Hello = () => {
  const {siteConfig} = useDocusaurusContext();
```

```
const {title, tagline} = siteConfig;

return <div>`${title} · ${tagline}`</div>;
};
```



If you just want to use those fields on the client side, you could create your own JS files and import them as ES6 modules, there is no need to put them in `docusaurus.config.js`.

Customizing Babel Configuration

Docusaurus transpiles your site's source code using Babel by default. If you want to customize the Babel configuration, you can do so by creating a `babel.config.js` file in your project root.

To use the built-in preset as a base configuration, install the following package and use it

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install --save @docusaurus/babel
```

Then use the preset in your `babel.config.js` file:

`babel.config.js`

```
export default {
  presets: ['@docusaurus/babel/preset'],
};
```

Most of the time, the default preset configuration will work just fine. If you want to customize your Babel configuration (e.g. to add support for Flow), you can directly edit this file. For your changes to take effect, you need to restart the Docusaurus dev server.

Playground

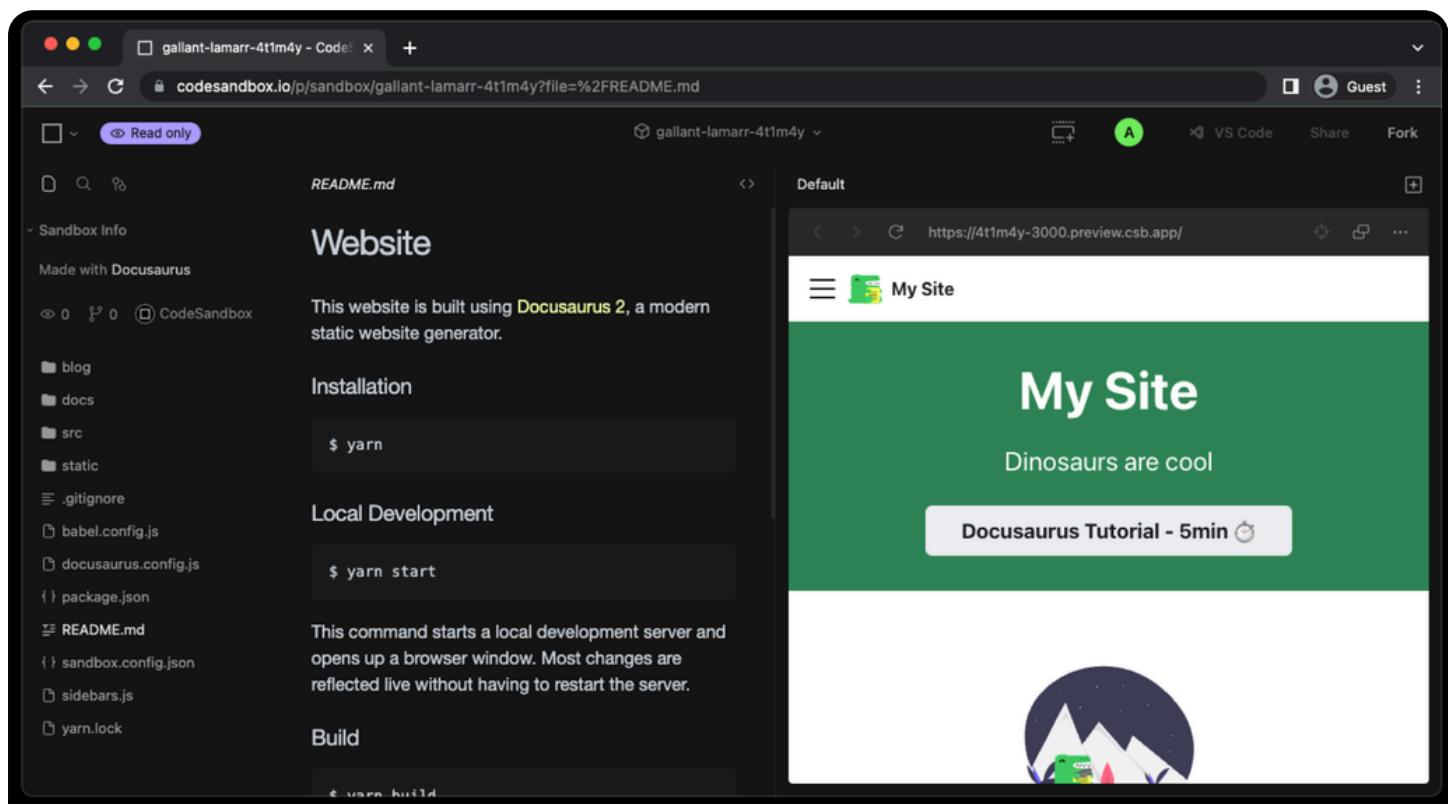
Playgrounds allow you to run Docusaurus **in your browser, without installing anything!**

They are mostly useful for:

- Testing Docusaurus
- Reporting bugs

Use [docusaurus.new](#) as an easy-to-remember shortcut.

Choose one of the available options below.



CodeSandbox

CodeSandbox is an online code editor and development environment that allows developers to create, share and collaborate on web development projects in a browser-based environment

[Try it now!](#)

[JavaScript](#)

[TypeScript](#)



StackBlitz uses a novel [WebContainers](#) technology to run Docusaurus directly in your browser.

[Try it now!](#)

JavaScript

TypeScript



For convenience, we'll remember your choice next time you visit docusaurus.new.

TypeScript Support

Docusaurus is written in TypeScript and provides first-class TypeScript support.

The minimum required version is **TypeScript 5.1**.

Initialization

Docusaurus supports writing and using TypeScript theme components. If the init template provides a TypeScript variant, you can directly [initialize a site](#) with full TypeScript support by using the `--typescript` flag.

```
npx create-docusaurus@latest my-website classic --typescript
```

Below are some guides on how to migrate an existing project to TypeScript.

Setup

Add the following packages to your project:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install --save-dev typescript @docusaurus/module-type-aliases  
@docusaurus/tsconfig @docusaurus/types
```

Then add `tsconfig.json` to your project root with the following content:

`tsconfig.json`

```
{  
  "extends": "@docusaurus/tsconfig",  
  "compilerOptions": {  
    "baseUrl": "."  
  }  
}
```

Docusaurus doesn't use this `tsconfig.json` to compile your project. It is added just for a nicer Editor experience, although you can choose to run `tsc` to type check your code for yourself or on CI.

Now you can start writing TypeScript theme components.

Typing the config file

It is possible to use a TypeScript config file in Docusaurus.

`docusaurus.config.ts`

```
import type {Config} from '@docusaurus/types';
import type * as Preset from '@docusaurus/preset-classic';

const config: Config = {
  title: 'My Site',
  favicon: 'img/favicon.ico',

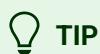
  /* Your site config here */

  presets: [
    [
      'classic',
      {
        /* Your preset config here */
      } satisfies Preset.Options,
    ],
  ],
}

themeConfig: {
  /* Your theme config here */
} satisfies Preset.ThemeConfig,
};

export default config;
```

▼ It is also possible to use [JSDoc type annotations](#) within a `.js` file:



TIP

Type annotations are very useful and help your IDE understand the type of config objects!

The best IDEs (VS Code, WebStorm, IntelliJ...) will provide a nice auto-completion experience.

Swizzling TypeScript theme components

For themes that support TypeScript theme components, you can add the `--typescript` flag to the end of the `swizzle` command to get TypeScript source code. For example, the following command will generate `index.tsx` and `styles.module.css` into `src/theme/Footer`.

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run swizzle @docusaurus/theme-classic Footer -- --typescript
```

All official Docusaurus themes support TypeScript theme components, including `theme-classic`, `theme-live-codeblock`, and `theme-search-algolia`. If you are a Docusaurus theme package author who wants to add TypeScript support, see the [Lifecycle APIs docs](#).

Creating Pages

In this section, we will learn about creating pages in Docusaurus.

The `@docusaurus/plugin-content-pages` plugin empowers you to create **one-off standalone pages** like a showcase page, playground page, or support page. You can use React components, or Markdown.

 **NOTE**

Pages do not have sidebars, only `docs` do.

 **INFO**

Check the [Pages Plugin API Reference documentation](#) for an exhaustive list of options.

Add a React page

React is used as the UI library to create pages. Every page component should export a React component, and you can leverage the expressiveness of React to build rich and interactive content.

Create a file `/src/pages/helloReact.js`:

`/src/pages/helloReact.js`

```
import React from 'react';
import Layout from '@theme/Layout';

export default function Hello() {
  return (
    <Layout title="Hello" description="Hello React Page">
      <div
        style={{
          display: 'flex',
          justifyContent: 'center',
          alignItems: 'center',
          height: '50vh',
          fontSize: '20px',
        }}>
        <p>
          Edit <code>pages/helloReact.js</code> and save to reload.
        </p>
      </div>
    </Layout>
  )
}
```

```
);  
}
```

Once you save the file, the development server will automatically reload the changes. Now open <http://localhost:3000/helloReact> and you will see the new page you just created.

Each page doesn't come with any styling. You will need to import the `Layout` component from `@theme/Layout` and wrap your contents within that component if you want the navbar and/or footer to appear.



You can also create TypeScript pages with the `.tsx` extension (`helloReact.tsx`).

Add a Markdown page

Create a file `/src/pages/helloMarkdown.md`:

```
/src/pages/helloMarkdown.md  
  
---  
title: my hello page title  
description: my hello page description  
hide_table_of_contents: true  
---  
  
# Hello  
  
How are you?
```

In the same way, a page will be created at <http://localhost:3000/helloMarkdown>.

Markdown pages are less flexible than React pages because it always uses the theme layout.

Here's an [example Markdown page](#).



You can use the full power of React in Markdown pages too, refer to the [MDX documentation](#).

Routing

If you are familiar with other static site generators like Jekyll and Next, this routing approach will feel familiar to you. Any JavaScript file you create under `/src/pages/` directory will be automatically converted to a website page, following the `/src/pages/` directory hierarchy. For example:

- `/src/pages/index.js` → `[baseUrl]`
- `/src/pages/foo.js` → `[baseUrl]/foo`
- `/src/pages/foo/test.js` → `[baseUrl]/foo/test`
- `/src/pages/foo/index.js` → `[baseUrl]/foo/`

In this component-based development era, it is encouraged to co-locate your styling, markup, and behavior together into components. Each page is a component, and if you need to customize your page design with your own styles, we recommend co-locating your styles with the page component in its own directory. For example, to create a "Support" page, you could do one of the following:

- Add a `/src/pages/support.js` file
- Create a `/src/pages/support/` directory and a `/src/pages/support/index.js` file.

The latter is preferred as it has the benefits of letting you put files related to the page within that directory. For example, a CSS module file (`styles.module.css`) with styles meant to only be used on the "Support" page.

NOTE

This is merely a recommended directory structure, and you will still need to manually import the CSS module file within your component module (`support/index.js`).

By default, any Markdown or JavaScript file starting with `_` will be ignored and no routes will be created for that file (see the `exclude` option).

```
my-website
├── src
│   └── pages
│       ├── styles.module.css
│       ├── index.js
│       ├── _ignored.js
│       ├── _ignored-folder
│       │   ├── Component1.js
│       │   └── Component2.js
│       └── support
│           ├── index.js
│           └── styles.module.css
└── .
```

WARNING

All JavaScript/TypeScript files within the `src/pages/` directory will have corresponding website paths generated for them. If you want to create reusable components into that directory, use the `exclude` option (by default, files prefixed with `_`, test files(`.test.js`), and files in `__tests__` directory are not turned into pages).

Duplicate Routes

You may accidentally create multiple pages that are meant to be accessed on the same route. When this happens, Docusaurus will warn you about duplicate routes when you run `yarn start` or `yarn build` (behavior configurable through the `onDuplicateRoutes` config), but the site will still be built successfully. The page that was created last will be accessible, but it will override other conflicting pages. To resolve this issue, you should modify or remove any conflicting routes.

Create a doc

Create a Markdown file, `greeting.md`, and place it under the `docs` directory.

```
website # root directory of your site
├── docs
│   └── greeting.md
├── src
│   └── pages
└── docusaurus.config.js
└── ...
```

```
---
```

`description`: Create a doc page with rich content.

```
---
```

Hello from Docusaurus

Are you ready to create the documentation site for your open source project?

Headers

will show up on the table of contents on the upper right

So that your users will know what this page is all about without scrolling down or even without reading too much.

Only h2 and h3 will be in the TOC by default.

You can configure the TOC heading levels either per-document or in the theme configuration.

The headers are well-spaced so that the hierarchy is clear.

- lists will help you
- present the key points
- that you want your users to remember
 - and you may nest them
 - multiple times

ⓘ NOTE

All files prefixed with an underscore (`_`) under the `docs` directory are treated as "partial" pages and will be ignored by default.

Read more about [importing partial pages](#).

Doc front matter

The [front matter](#) is used to provide additional metadata for your doc page. Front matter is optional—Docusaurus will be able to infer all necessary metadata without the front matter. For example, the [doc tags](#) feature introduced below requires using front matter. For all possible fields, see [the API documentation](#).

Doc tags

Tags are declared in the front matter and introduce another dimension of categorization in addition to the [docs sidebar](#).

It is possible to define tags inline, or to reference predefined tags declared in a [tags file](#) (optional, usually `docs/tags.yml`).

In the following example:

- `docusaurus` references a predefined tag key declared in `docs/tags.yml`
- `Releases` is an inline tag, because it does not exist in `docs/tags.yml`

```
docs/my-doc.md
```

```
---
```

```
tags:
```

```
  - Releases
```

```
  - docusaurus
```

```
---
```

```
# Title
```

```
Content
```

```
docs/tags.yml
```

```
docusaurus:  
  label: 'Docusaurus'  
  permalink: '/docusaurus'  
  description: 'Docs related to the Docusaurus framework'
```



TIP

Tags can also be declared with `tags: [Demo, Getting started]`.

Read more about all the possible [Yaml array syntaxes](#).

Organizing folder structure

How the Markdown files are arranged under the `docs` folder can have multiple impacts on Docusaurus content generation. However, most of them can be decoupled from the file structure.

Document ID

Every document has a unique `id`. By default, a document `id` is the name of the document (without the extension) relative to the root `docs` directory.

For example, the ID of `greeting.md` is `greeting`, and the ID of `guide/hello.md` is `guide/hello`.

```
website # Root directory of your site
└── docs
    ├── greeting.md
    └── guide
        └── hello.md
```

However, the **last part** of the `id` can be defined by the user in the front matter. For example, if `guide/hello.md`'s content is defined as below, its final `id` is `guide/part1`.

```
---
```

```
id: part1
```

```
---
```

```
  Lorem ipsum
```

The ID is used to refer to a document when hand-writing sidebars, or when using `docs`-related layout components or hooks.

Doc URLs

By default, a document's URL location is its file path relative to the `docs` folder, with a few exceptions. Namely, if a file is named one the following, the file name won't be included in the URL:

- Named as `index` (case-insensitive): `docs/Guides/index.md`
- Named as `README` (case-insensitive): `docs/Guides/README.mdx`
- Same name as parent folder: `docs/Guides/Guides.md`

In all cases, the default slug would only be `/Guides`, without the `/index`, `/README`, or duplicate `/Guides` segment.

NOTE

This convention is exactly the same as the category index convention. However, the `isCategoryIndex` configuration does *not* affect the document URL.

Use the `slug` front matter to change a document's URL.

For example, suppose your site structure looks like this:

```
website # Root directory of your site
└── docs
    └── guide
        └── hello.md
```

By default `hello.md` will be available at `/docs/guide/hello`. You can change its URL location to `/docs/bonjour`:

```
---
slug: /bonjour
---

Lorem ipsum
```

`slug` will be appended to the doc plugin's `routebasePath`, which is `/docs` by default. See [Docs-only mode](#) for how to remove the `/docs` part from the URL.

NOTE

It is possible to use:

- absolute slugs: `slug: /mySlug`, `slug: /...`
- relative slugs: `slug: mySlug`, `slug: ../../mySlug...`

If you want a document to be available at the root, and have a path like `https://docusaurus.io/docs/`, you can use the `slug` front matter:

```
---
```

```
id: my-home-doc
```

```
slug: /
```

```
---
```

 Lorem ipsum

Sidebars

When using [autogenerated sidebars](#), the file structure will determine the sidebar structure.

Our recommendation for file system organization is: make your file system mirror the sidebar structure (so you don't need to handwrite your `sidebars.js` file), and use the `slug` front matter to customize URLs of each document.

Sidebar items

We have introduced three types of item types in the example in the previous section: `doc`, `category`, and `link`, whose usages are fairly intuitive. We will formally introduce their APIs. There's also a fourth type: `autogenerated`, which we will explain in detail later.

- **Doc**: link to a doc page, associating it with the sidebar
- **Link**: link to any internal or external page
- **Category**: creates a dropdown of sidebar items
- **Autogenerated**: generate a sidebar slice automatically
- **HTML**: renders pure HTML in the item's position
- ***Ref**: link to a doc page, without making the item take part in navigation generation

Doc: link to a doc

Use the `doc` type to link to a doc page and assign that doc to a sidebar:

```
type SidebarItemDoc =  
  // Normal syntax  
  | {  
    type: 'doc';  
    id: string;  
    label: string; // Sidebar label text  
    className?: string; // Class name for sidebar label  
    customProps?: Record<string, unknown>; // Custom props  
  }  
  
  // Shorthand syntax  
  | string; // docId shortcut
```

Example:

sidebars.js

```
export default {  
  mySidebar: [  
    // Normal syntax:  
    {  
      type: 'doc',  
      id: 'doc1', // document ID  
      label: 'Getting started', // sidebar label
```

```
  },
  // Shorthand syntax:
  'doc2', // document ID
],
};
```

If you use the doc shorthand or `autogenerated` sidebar, you would lose the ability to customize the sidebar label through item definition. You can, however, use the `sidebar_label` Markdown front matter within that doc, which has higher precedence over the `label` key in the sidebar item. Similarly, you can use `sidebar_custom_props` to declare custom metadata for a doc page.

 **NOTE**

A `doc` item sets an implicit sidebar association. Don't assign the same doc to multiple sidebars: change the type to `ref` instead.

 **TIP**

Sidebar custom props is a useful way to propagate arbitrary doc metadata to the client side, so you can get additional information when using any doc-related hook that fetches a doc object.

Link: link to any page

Use the `link` type to link to any page (internal or external) that is not a doc.

```
type SidebarItemLink = {
  type: 'link';
  label: string;
  href: string;
  className?: string;
  description?: string;
};
```

Example:

`sidebars.js`

```
export default {
  myLinksSidebar: [
    // External link
    {
      type: 'link',
```

```

        label: 'Facebook', // The link label
        href: 'https://facebook.com', // The external URL
    },
    // Internal link
    {
        type: 'link',
        label: 'Home', // The link label
        href: '/', // The internal path
    },
],
};

```

HTML: render custom markup

Use the `html` type to render custom HTML within the item's `` tag.

This can be useful for inserting custom items such as dividers, section titles, ads, and images.

```

type SidebarItemHtml = {
    type: 'html';
    value: string;
    defaultStyle?: boolean; // Use default menu item styles
    className?: string;
};

```

Example:

`sidebars.js`

```

export default {
    myHtmlSidebar: [
        {
            type: 'html',
            value: '', // The HTML to be
            rendered
            defaultStyle: true, // Use the default menu item styling
        },
    ],
};

```

 **TIP**

The menu item is already wrapped in an `` tag, so if your custom item is simple, such as a title, just supply a string as the value and use the `className` property to style it:

sidebars.js

```
export default {
  myHtmlSidebar: [
    {
      type: 'html',
      value: 'Core concepts',
      className: 'sidebar-title',
    },
  ],
};
```

Category: create a hierarchy

Use the `category` type to create a hierarchy of sidebar items.

```
type SidebarItemCategory = {
  type: 'category';
  label: string; // Sidebar label text.
  items: SidebarItem[]; // Array of sidebar items.
  className?: string;
  description?: string;

  // Category options:
  collapsible: boolean; // Set the category to be collapsible
  collapsed: boolean; // Set the category to be initially collapsed or open by default
  link: SidebarItemCategoryLinkDoc | SidebarItemCategoryLinkGeneratedIndex;
};
```

Example:

sidebars.js

```
export default {
  docs: [
    {
      type: 'category',
      label: 'Guides',
      collapsible: true,
```

```
collapsed: false,
items: [
  'creating-pages',
  {
    type: 'category',
    label: 'Docs',
    items: ['introduction', 'sidebar', 'markdown-features', 'versioning'],
  },
],
},
],
],
};
```



TIP

Use the [shorthand syntax](#) when you don't need customizations:

sidebars.js

```
export default {
  docs: {
    Guides: [
      'creating-pages',
      {
        Docs: ['introduction', 'sidebar', 'markdown-features', 'versioning'],
      },
    ],
  },
};
```

Category links

With category links, clicking on a category can navigate you to another page.



TIP

Use category links to introduce a category of documents.

Autogenerated categories can use the [_category_.yml](#) file to declare the link.

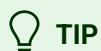
Generated index page

You can auto-generate an index page that displays all the direct children of this category. The [slug](#) allows you to customize the generated page's route, which defaults to [/category/\[categoryName\]](#).

sidebars.js

```
export default {
  docs: [
    {
      type: 'category',
      label: 'Guides',
      link: {
        type: 'generated-index',
        title: 'Docusaurus Guides',
        description: 'Learn about the most important Docusaurus concepts!',
        slug: '/category/docusaurus-guides',
        keywords: ['guides'],
        image: '/img/docusaurus.png',
      },
      items: ['pages', 'docs', 'blog', 'search'],
    },
  ],
};
```

See it in action on the [Docusaurus Guides page](#).



TIP

Use `generated-index` links as a quick way to get an introductory document.

Doc link

A category can link to an existing document.

sidebars.js

```
export default {
  docs: [
    {
      type: 'category',
      label: 'Guides',
      link: {type: 'doc', id: 'introduction'},
      items: ['pages', 'docs', 'blog', 'search'],
    },
  ],
};
```

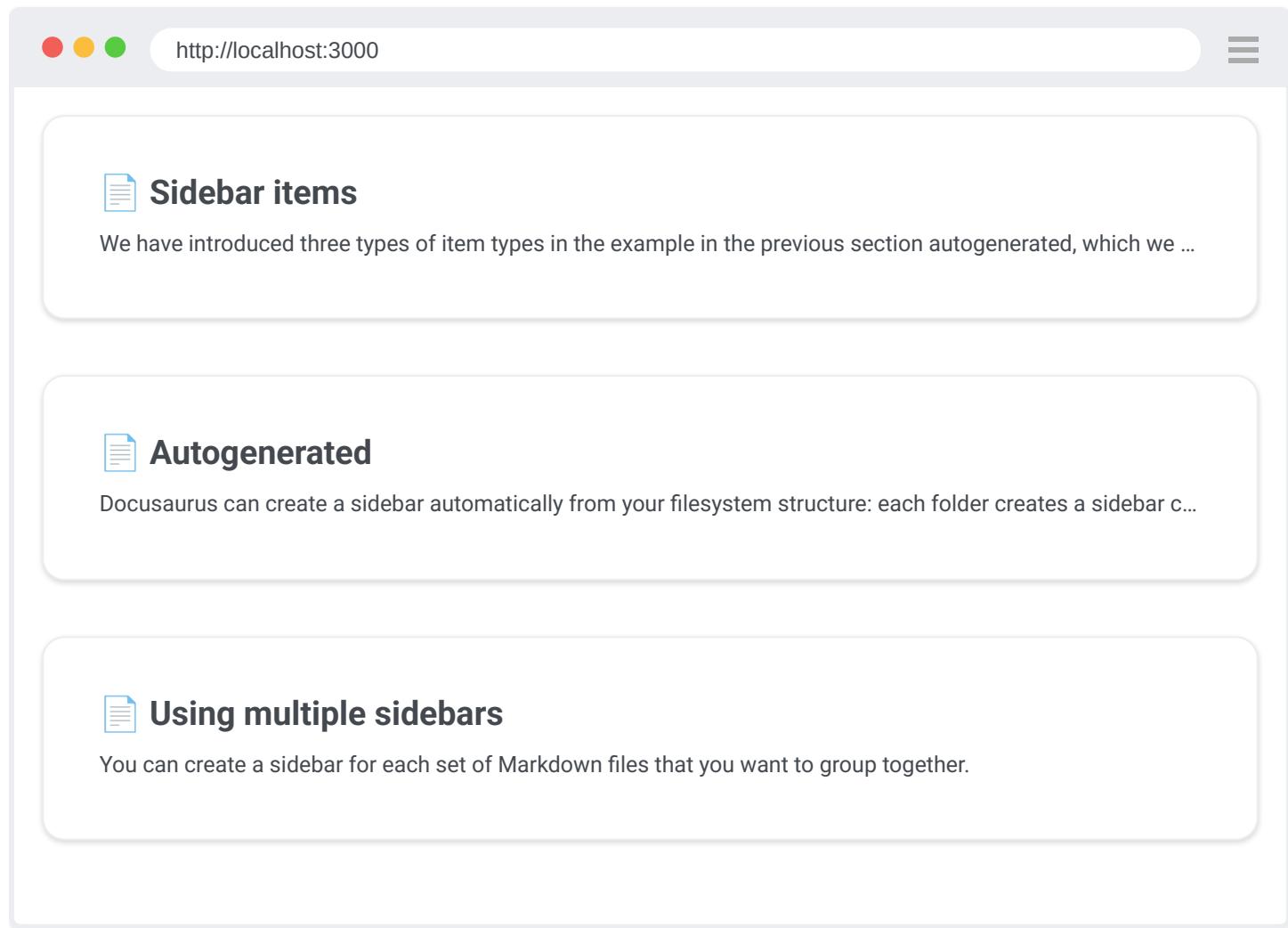
See it in action on the [i18n introduction page](#).

Embedding generated index in doc page

You can embed the generated cards list in a normal doc page as well with the `DocCardList` component. It will display all the sidebar items of the parent category of the current document.

docs/sidebar/index.md

```
import DocCardList from '@theme/DocCardList';  
  
<DocCardList />
```



The screenshot shows a browser window with the URL `http://localhost:3000`. The page displays three sidebar items in cards:

- Sidebar items**: Described as autogenerated, with a note about Docusaurus creating a sidebar from filesystem structure.
- Autogenerated**: Described as a feature where Docusaurus creates a sidebar automatically from your filesystem structure.
- Using multiple sidebars**: Described as a way to create a sidebar for each set of Markdown files you want to group together.

Collapsible categories

We support the option to expand/collapse categories. Categories are collapsible by default, but you can disable collapsing with `collapsible: false`.

sidebars.js

```
export default {  
  docs: [  
    {  
      type: 'category',
```

```
label: 'Guides',
items: [
  'creating-pages',
  {
    type: 'category',
    collapsible: false,
    label: 'Docs',
    items: ['introduction', 'sidebar', 'markdown-features', 'versioning'],
  },
],
},
],
};
```

To make all categories non-collapsible by default, set the `sidebarCollapsible` option in `plugin-content-docs` to `false`:

`docusaurus.config.js`

```
export default {
presets: [
  [
    '@docusaurus/preset-classic',
    {
      docs: {
        sidebarCollapsible: false,
      },
    },
  ],
],
};
```

 **NOTE**

The option in `sidebars.js` takes precedence over plugin configuration, so it is possible to make certain categories collapsible when `sidebarCollapsible` is set to `false` globally.

Expanded categories by default

Collapsible categories are collapsed by default. If you want them to be expanded on the first render, you can set `collapsed` to `false`:

`sidebars.js`

```
export default {
  docs: [
    Guides: [
      'creating-pages',
      {
        type: 'category',
        label: 'Docs',
        collapsed: false,
        items: ['markdown-features', 'sidebar', 'versioning'],
      },
    ],
  ],
};
```

Similar to `collapsible`, you can also set the global configuration `options.sidebarCollapsed` to `false`. Individual `collapsed` options in `sidebars.js` will still take precedence over this configuration.

docusaurus.config.js

```
export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          sidebarCollapsed: false,
        },
      },
    ],
  ],
};
```

⚠️ WARNING

When a category has `collapsed: true` but `collapsible: false` (either through `sidebars.js` or through plugin configuration), the latter takes precedence and the category is still rendered as expanded.

Using shorthands

You can express typical sidebar items without much customization more concisely with **shorthand syntaxes**. There are two parts to this: **doc shorthand** and **category shorthand**.

Doc shorthand

An item with type `doc` can be simply a string representing its ID:

Longhand Shorthand

sidebars.js

```
export default {
  sidebar: [
    {
      type: 'doc',
      id: 'myDoc',
    },
  ],
};
```

So it's possible to simplify the example above to:

sidebars.js

```
export default {
  mySidebar: [
    {
      type: 'category',
      label: 'Getting Started',
      items: [
        'doc1',
      ],
    },
    {
      type: 'category',
      label: 'Docusaurus',
      items: [
        'doc2',
        'doc3',
      ],
    },
    {
      type: 'link',
      label: 'Learn more',
      href: 'https://example.com',
    },
  ],
};
```

Category shorthand

A category item can be represented by an object whose key is its label, and the value is an array of subitems.

Longhand **Shorthand**

sidebars.js

```
export default {
  sidebar: [
    {
      type: 'category',
      label: 'Getting started',
      items: ['doc1', 'doc2'],
    },
  ],
};
```

This permits us to simplify that example to:

sidebars.js

```
export default {
  mySidebar: [
    {
      'Getting started': ['doc1'],
    },
    {
      'Docusaurus': ['doc2', 'doc3'],
    },
    {
      type: 'link',
      label: 'Learn more',
      href: 'https://example.com',
    },
  ],
};
```

Each shorthand object after this transformation will contain exactly one entry. Now consider the further simplified example below:

sidebars.js

```
export default {
  mySidebar: [
    {
      'Getting started': ['doc1'],
      Docusaurus: ['doc2', 'doc3'],
    },
    {
      type: 'link',
      label: 'Learn more',
      href: 'https://example.com',
    },
  ],
};
```

Note how the two consecutive category shorthands are compressed into one object with two entries. This syntax generates a **sidebar slice**: you shouldn't see that object as one bulk item—this object is unwrapped, with each entry becoming a separate item, and they spliced together with the rest of the items (in this case, the "Learn more" link) to form the final sidebar level. Sidebar slices are also important when discussing [autogenerated sidebars](#).

Wherever you have an array of items that is reduced to one category shorthand, you can omit that enclosing array as well.

Before **After**

sidebars.js

```
export default {
  sidebar: [
    {
      'Getting started': ['doc1'],
      Docusaurus: [
        {
          'Basic guides': ['doc2', 'doc3'],
          'Advanced guides': ['doc4', 'doc5'],
        },
      ],
    },
  ],
};
```

Autogenerated

Docusaurus can **create a sidebar automatically** from your **filesystem structure**: each folder creates a sidebar category, and each file creates a doc link.

```
type SidebarItemAutogenerated = {
  type: 'autogenerated';
  dirName: string; // Source folder to generate the sidebar slice from (relative
  to docs)
};
```

Docusaurus can generate a full sidebar from your docs folder:

sidebars.js

```
export default {
  myAutogeneratedSidebar: [
    {
      type: 'autogenerated',
      dirName: '.', // '.' means the current docs folder
    },
  ],
};
```

An `autogenerated` item is converted by Docusaurus to a **sidebar slice** (also discussed in [category shorthands](#)): a list of items of type `doc` or `category`, so you can splice **multiple autogenerated items** from multiple directories, interleaving them with regular sidebar items, in one sidebar level.

▼ A real-world example

Category index convention

Docusaurus can automatically link a category to its index document.

A category index document is a document following one of those filename conventions:

- Named as `index` (case-insensitive): `docs/Guides/index.md`
- Named as `README` (case-insensitive): `docs/Guides/README.mdx`
- Same name as parent folder: `docs/Guides/Guides.md`

This is equivalent to using a category with a [doc link](#):

sidebars.js

```
export default {
  docs: [
    {
      type: 'category',
      label: 'Guides',
      link: {type: 'doc', id: 'Guides/index'},
      items: [],
    },
  ],
};
```



TIP

Naming your introductory document `README.md` makes it show up when browsing the folder using the GitHub interface, while using `index.md` makes the behavior more in line with how HTML files are served.



TIP

If a folder only has one index page, it will be turned into a link instead of a category. This is useful for **asset collocation**:

```
some-doc
├── index.md
└── img1.png
    └── img2.png
```

▼ Customizing category index matching

Autogenerated sidebar metadata

For handwritten sidebar definitions, you would provide metadata to sidebar items through `sidebars.js`; for autogenerated, Docusaurus would read them from the item's respective file. In addition, you may want to adjust the relative position of each item because, by default, items within a sidebar slice will be generated in **alphabetical order** (using file and folder names).

Doc item metadata

The `label`, `className`, and `customProps` attributes are declared in front matter as `sidebar_label`, `sidebar_class_name`, and `sidebar_custom_props`, respectively. Position can be specified in the same way, via `sidebar_position` front matter.

```
docs/tutorials/tutorial-easy.md
```

```
---
```

```
sidebar_position: 2
sidebar_label: Easy
sidebar_class_name: green
```

```
--
```

Easy Tutorial

This is the easy tutorial!

Category item metadata

Add a `_category_.json` or `_category_.yml` file in the respective folder. You can specify any category metadata and also the `position` metadata. `label`, `className`, `position`, and `customProps` will default to the respective values of the category's linked doc, if there is one.

[JSON](#) [YAML](#)

```
docs/tutorials/_category_.json
```

```
{
  "position": 2.5,
  "label": "Tutorial",
  "collapsible": true,
  "collapsed": false,
  "className": "red",
  "link": {
    "type": "generated-index",
    "title": "Tutorial overview"
  },
  "customProps": {
    "description": "This description can be used in the swizzled DocCard"
  }
}
```

 INFO

If the `link` is explicitly specified, Docusaurus will not apply any default conventions.

The doc links can be specified relatively, e.g. if the category is generated with the `guides` directory, `"link": {"type": "doc", "id": "intro"}` will be resolved to the ID `guides/intro`, only falling back to `intro` if a doc with the former ID doesn't exist.

You can also use `link: null` to opt out of default conventions and not generate any category index page.

⚠ INFO

The position metadata is only used **within a sidebar slice**: Docusaurus does not re-order other items of your sidebar.

Using number prefixes

A simple way to order an autogenerated sidebar is to prefix docs and folders by number prefixes, which also makes them appear in the file system in the same order when sorted by file name:

```
docs
├── 01-Intro.md
├── 02-Tutorial Easy
│   ├── 01-First Part.md
│   ├── 02-Second Part.md
│   └── 03-End.md
├── 03-Tutorial Advanced
│   ├── 01-First Part.md
│   ├── 02-Second Part.md
│   ├── 03-Third Part.md
│   └── 04-End.md
└── 04-End.md
```

To make it **easier to adopt**, Docusaurus supports **multiple number prefix patterns**.

By default, Docusaurus will **remove the number prefix** from the doc id, title, label, and URL paths.

⚠ WARNING

Prefer using additional metadata.

Updating a number prefix can be annoying, as it can require **updating multiple existing Markdown links**:

- Check the [Tutorial End](./04-End.mdx);
- + Check the [Tutorial End](./05-End.mdx);

Customize the sidebar items generator

You can provide a custom `sidebarItemsGenerator` function in the docs plugin (or preset) config:

`docusaurus.config.js`

```
export default {
  plugins: [
    [
      '@docusaurus/plugin-content-docs',
      {
        async sidebarItemsGenerator({
          defaultSidebarItemsGenerator,
          numberPrefixParser,
          item,
          version,
          docs,
          categoriesMetadata,
          isCategoryIndex,
        }) {
          // Example: return an hardcoded list of static sidebar items
          return [
            {type: 'doc', id: 'doc1'},
            {type: 'doc', id: 'doc2'},
            ],
          },
        ],
      ],
    ];
};
```



TIP

Re-use and enhance the default generator instead of writing a generator from scratch: [the default generator we provide](#) is 250 lines long.

Add, update, filter, re-order the sidebar items according to your use case:

docusaurus.config.js

```
// Reverse the sidebar items ordering (including nested category items)
function reverseSidebarItems(items) {
  // Reverse items in categories
  const result = items.map((item) => {
    if (item.type === 'category') {
      return {...item, items: reverseSidebarItems(item.items)};
    }
    return item;
  });
  // Reverse items at current level
  result.reverse();
  return result;
}

export default {
  plugins: [
    [
      '@docusaurus/plugin-content-docs',
      {
        async sidebarItemsGenerator({defaultSidebarItemsGenerator, ...args}) {
          const sidebarItems = await defaultSidebarItemsGenerator(args);
          return reverseSidebarItems(sidebarItems);
        },
      },
    ],
  ],
};
```

Using multiple sidebars

You can create a sidebar for each **set of Markdown files** that you want to **group together**.



TIP

The Docusaurus site is a good example of using multiple sidebars:

- [Docs](#)
- [API](#)

Consider this example:

sidebars.js

```
export default {
  tutorialSidebar: {
    'Category A': ['doc1', 'doc2'],
  },
  apiSidebar: ['doc3', 'doc4'],
};
```

When browsing `doc1` or `doc2`, the `tutorialSidebar` will be displayed; when browsing `doc3` or `doc4`, the `apiSidebar` will be displayed.

Understanding sidebar association

Following the example above, if a `commonDoc` is included in both sidebars:

sidebars.js

```
export default {
  tutorialSidebar: {
    'Category A': ['doc1', 'doc2', 'commonDoc'],
  },
  apiSidebar: ['doc3', 'doc4', 'commonDoc'],
};
```

How does Docusaurus know which sidebar to display when browsing `commonDoc`? Answer: it doesn't, and we don't guarantee which sidebar it will pick.

When you add doc Y to sidebar X, it creates a two-way binding: sidebar X contains a link to doc Y, and when browsing doc Y, sidebar X will be displayed. But sometimes, we want to break either implicit binding:

1. *How do I generate a link to doc Y in sidebar X without making sidebar X displayed on Y?* For example, when I include doc Y in multiple sidebars as in the example above, and I want to explicitly tell Docusaurus to display one sidebar?
2. *How do I make sidebar X displayed when browsing doc Y, but sidebar X shouldn't contain the link to Y?* For example, when Y is a "doc home page" and the sidebar is purely used for navigation?

Front matter option `displayed_sidebar` will forcibly set the sidebar association. For the same example, you can still use doc shorthands without any special configuration:

sidebars.js

```
export default {  
  tutorialSidebar: {  
    'Category A': ['doc1', 'doc2'],  
  },  
  apiSidebar: ['doc3', 'doc4'],  
};
```

And then add a front matter:

commonDoc.md

```
---  
displayed_sidebar: apiSidebar  
---
```

Which explicitly tells Docusaurus to display `apiSidebar` when browsing `commonDoc`. Using the same method, you can make sidebar X which doesn't contain doc Y appear on doc Y:

home.md

```
---  
displayed_sidebar: tutorialSidebar  
---
```

Even when `tutorialSidebar` doesn't contain a link to `home`, it will still be displayed when viewing `home`.

If you set `displayed_sidebar: null`, no sidebar will be displayed whatsoever on this page, and subsequently, no pagination either.

Generating pagination

Docusaurus uses the sidebar to generate the "next" and "previous" pagination links at the bottom of each doc page. It strictly uses the sidebar that is displayed: if no sidebar is associated, it doesn't generate pagination either. However, the docs linked as "next" and "previous" are not guaranteed to display the same sidebar: they are included in this sidebar, but in their front matter, they may have a different `displayed_sidebar`.

If a sidebar is displayed by setting `displayed_sidebar` front matter, and this sidebar doesn't contain the doc itself, no pagination is displayed.

You can customize pagination with front matter `pagination_next` and `pagination_prev`. Consider this sidebar:

`sidebars.js`

```
export default {
  tutorial: [
    'introduction',
    {
      installation: ['windows', 'linux', 'macos'],
    },
    'getting-started',
  ],
};
```

The pagination next link on "windows" points to "linux", but that doesn't make sense: you would want readers to proceed to "getting started" after installation. In this case, you can set the pagination manually:

`windows.md`

```
---
: getting-started
---
# Installation on Windows
```

You can also disable displaying a pagination link with `pagination_next: null` or `pagination_prev: null`.

The pagination label by default is the sidebar label. You can use the front matter `pagination_label` to customize how this doc appears in the pagination.

The `ref` item

The `ref` type is identical to the `doc` type in every way, except that it doesn't participate in generating navigation metadata. It only registers itself as a link. When [generating pagination](#) and [displaying sidebar](#), `ref` items are completely ignored.

It is particularly useful where you wish to link to the same document from multiple sidebars. The document only belongs to one sidebar (the one where it's registered as `type: 'doc'` or from an autogenerated directory), but its link will appear in all sidebars that it's registered in.

Consider this example:

```
sidebars.js
```

```
export default {
  tutorialSidebar: {
    'Category A': [
      'doc1',
      'doc2',
      {type: 'ref', id: 'commonDoc'},
      'doc5',
    ],
  },
  apiSidebar: ['doc3', 'doc4', 'commonDoc'],
};
```

You can think of the `ref` type as the equivalent to doing the following:

- Setting `displayed_sidebar: tutorialSidebar` for `commonDoc` (`ref` is ignored in sidebar association)
- Setting `pagination_next: doc5` for `doc2` and setting `pagination_prev: doc2` for `doc5` (`ref` is ignored in pagination generation)

Versioning

You can use the versioning CLI to create a new documentation version based on the latest content in the `docs` directory. That specific set of documentation will then be preserved and accessible even as the documentation in the `docs` directory continues to evolve.

⚠️ WARNING

Think about it before starting to version your documentation - it can become difficult for contributors to help improve it!

Most of the time, you don't need versioning as it will just increase your build time, and introduce complexity to your codebase. Versioning is **best suited for websites with high-traffic and rapid changes to documentation between versions**. If your documentation rarely changes, don't add versioning to your documentation.

To better understand how versioning works and see if it suits your needs, you can read on below.

Overview

A typical versioned doc site looks like below:

```
website
├── sidebars.json      # sidebar for the current docs version
├── docs                # docs directory for the current docs version
│   ├── foo
│   │   ├── bar.md      # https://mysite.com/docs/next/foo/bar
│   │   └── hello.md     # https://mysite.com/docs/next/hello
├── versions.json       # file to indicate what versions are available
├── versioned_docs
    ├── version-1.1.0
        ├── foo
        │   ├── bar.md      # https://mysite.com/docs/next/foo/bar
        │   └── hello.md
    └── version-1.0.0
        ├── foo
        │   ├── bar.md      # https://mysite.com/docs/1.0.0/foo/bar
        │   └── hello.md
└── versioned_sidebars
    ├── version-1.1.0-sidebars.json
    └── version-1.0.0-sidebars.json
├── docusaurus.config.js
└── package.json
```

The `versions.json` file is a list of version names, ordered from newest to oldest.

The table below explains how a versioned file maps to its version and the generated URL.

Path	Version	URL
<code>versioned_docs/version-1.0.0/hello.md</code>	1.0.0	<code>/docs/1.0.0/hello</code>
<code>versioned_docs/version-1.1.0/hello.md</code>	1.1.0 (latest)	<code>/docs/hello</code>
<code>docs/hello.md</code>	current	<code>/docs/next/hello</code>



The files in the `docs` directory belong to the `current` docs version.

By default, the `current` docs version is labeled as `Next` and hosted under `/docs/next/*`, but it is entirely configurable to fit your project's release lifecycle.

Terminology

Note the terminology we use here.

Current version

The version placed in the `./docs` folder.

Latest version / last version

The version served by default for docs navbar items. Usually has path `/docs`.

Current version is defined by the **file system location**, while latest version is defined by the **the navigation behavior**. They may or may not be the same version! (And the default configuration, as shown in the table above, would treat them as different: current version at `/docs/next` and latest at `/docs`.)

Tutorials

Tagging a new version

1. First, make sure the current docs version (the `./docs` directory) is ready to be frozen.
2. Enter a new version number.

```
npm run docusaurus docs:version 1.1.0
```

When tagging a new version, the document versioning mechanism will:

- Copy the full `docs/` folder contents into a new `versioned_docs/version-[versionName]/` folder.
- Create a versioned sidebar file based from your current `sidebar` configuration (if it exists) - saved as `versioned_sidebars/version-[versionName]-sidebars.json`.
- Append the new version number to `versions.json`.

Creating new docs

1. Place the new file into the corresponding version folder.
2. Include the reference to the new file in the corresponding sidebar file according to the version number.

Current version structure

Older version structure

```
# The new file.  
docs/new.md
```

```
# Edit the corresponding sidebar file.  
sidebars.js
```



Versioned sidebar files are, like standard sidebar files, relative to the content root for the given version – so for the example above, your versioned sidebar file may look like:

```
{  
  "sidebar": [  
    {  
      "type": " autogenerated",  
      "dirName": ". "  
    }  
  ]  
}
```

or for a manual sidebar:

```
{  
  "sidebar": [  
    {  
      "type": "doc",  
      "id": "new",  
      "label": "New"  
    }  
  ]  
}
```

Updating an existing version

You can update multiple docs versions at the same time because each directory in `versioned_docs/` represents specific routes when published.

1. Edit any file.
2. Commit and push changes.
3. It will be published to the version.

Example: When you change any file in `versioned_docs/version-2.6/`, it will only affect the docs for version `2.6`.

Deleting an existing version

You can delete/remove versions as well.

1. Remove the version from `versions.json`.

Example:

```
[  
  "2.0.0",  
  "1.9.0",  
  - "1.8.0"  
]
```

2. Delete the versioned docs directory. Example: `versioned_docs/version-1.8.0`.
3. Delete the versioned sidebars file. Example: `versioned_sidebars/version-1.8.0-sidebars.json`.

Configuring versioning behavior

The "current" version is the version name for the `./docs` folder. There are different ways to manage versioning, but two very common patterns are:

- You release v1, and start immediately working on v2 (including its docs). In this case, the **current version** is v2, which is in the `./docs` source folder, and can be browsed at `example.com/docs/next`. The **latest version** is v1, which is in the `./versioned_docs/version-1` source folder, and is browsed by most of your users at `example.com/docs`.
- You release v1, and will maintain it for some time before thinking about v2. In this case, the **current version** and **latest version** will both be point to v1, since the v2 docs doesn't even exist yet!

Docusaurus defaults work great for the first use case. We will label the current version as "next" and you can even choose not to publish it.

For the 2nd use case: if you release v1 and don't plan to work on v2 anytime soon, instead of versioning v1 and having to maintain the docs in 2 folders (`./docs` + `./versioned_docs/version-1.0.0`), you may consider "pretending" that the current version is a cut version by giving it a path and a label:

`docusaurus.config.js`

```
export default {
  presets: [
    '@docusaurus/preset-classic',
    docs: {
      lastVersion: 'current',
      versions: {
        current: {
          label: '1.0.0',
          path: '1.0.0',
        },
      },
    },
  ],
};
```

The docs in `./docs` will be served at `/docs/1.0.0` instead of `/docs/next`, and `1.0.0` will become the default version we link to in the navbar dropdown, and you will only need to maintain a single `./docs` folder.

We offer these plugin options to customize versioning behavior:

- `disableVersioning`: Explicitly disable versioning even with versions. This will make the site only include the current version.
- `includeCurrentVersion`: Include the current version (the `./docs` folder) of your docs.
 - **Tip:** turn it off if the current version is a work-in-progress, not ready to be published.
- `lastVersion`: Sets which version "latest version" (the `/docs` route) refers to.
 - **Tip:** `lastVersion: 'current'` makes sense if your current version refers to a major version that's constantly patched and released. The actual route base path and label of the latest version are configurable.
- `onlyIncludeVersions`: Defines a subset of versions from `versions.json` to be deployed.
 - **Tip:** limit to 2 or 3 versions in dev and deploy previews to improve startup and build time.
- `versions`: A dictionary of version metadata. For each version, you can customize the following:
 - `label`: the label displayed in the versions dropdown and banner.
 - `path`: the route base path of this version. By default, latest version has `/` and current version has `/next`.
 - `banner`: one of `'none'`, `'unreleased'`, and `'unmaintained'`. Determines what's displayed at the top of every doc page. Any version above the latest version would be "unreleased", and any version below would be "unmaintained".
 - `badge`: show a badge with the version name at the top of a doc of that version.
 - `className`: add a custom `className` to the `<html>` element of doc pages of that version.

See [docs plugin configuration](#) for more details.

Navbar items

We offer several docs navbar items to help you quickly set up navigation without worrying about versioned routes.

- `doc`: a link to a doc.
- `docSidebar`: a link to the first item in a sidebar.
- `docsVersion`: a link to the main doc of the currently viewed version.
- `docsVersionDropdown`: a dropdown containing all the versions available.

These links would all look for an appropriate version to link to, in the following order:

1. **Active version:** the version that the user is currently browsing, if she is on a page provided by this doc plugin. If she's not on a doc page, fall back to...
2. **Preferred version:** the version that the user last viewed. If there's no history, fall back to...
3. **Latest version:** the default version that we navigate to, configured by the `lastVersion` option.

docsVersionDropdown

By default, the `docsVersionDropdown` displays a dropdown with all the available docs versions.

The `versions` attribute allows you to display a subset of the available docs versions in a given order:

docusaurus.config.js

```
export default {
  themeConfig: {
    navbar: {
      items: [
        {
          type: 'docsVersionDropdown',
          versions: ['current', '3.0', '2.0'],
        },
      ],
    },
  },
};
```

Passing a `versions` object, lets you override the display label of each version:

docusaurus.config.js

```
export default {
  themeConfig: {
    navbar: {
      items: [
        {
          type: 'docsVersionDropdown',
          versions: {
            current: {label: 'Version 4.0'},
            '3.0': {label: 'Version 3.0'},
            '2.0': {label: 'Version 2.0'},
          },
        },
      ],
    },
  },
};
```

Recommended practices

Version your documentation only when needed

For example, you are building documentation for your npm package `foo` and you are currently in version 1.0.0. You then release a patch version for a minor bug fix and it's now 1.0.1.

Should you cut a new documentation version 1.0.1? **You probably shouldn't**. 1.0.1 and 1.0.0 docs shouldn't differ according to semver because there are no new features!. Cutting a new version for it will only just create unnecessary duplicated files.

Keep the number of versions small

As a good rule of thumb, try to keep the number of your versions below 10. You will **very likely** to have a lot of obsolete versioned documentation that nobody even reads anymore. For example, `Jest` is currently in version `27.4`, and only maintains several latest documentation versions with the lowest being `25.X`. Keep it small 😊

ARCHIVE OLDER VERSIONS

If you deploy your site on a Jamstack provider (e.g. [Netlify](#)), the provider will save each production build as a snapshot under an immutable URL. You can include archived versions that will never be rebuilt as external links to these immutable URLs. The `Jest` website and the `Docusaurus` website both use such pattern to keep the number of actively built versions low.

Use absolute import within the docs

Don't use relative paths import within the docs. Because when we cut a version the paths no longer work (the nesting level is different, among other reasons). You can utilize the `@site` alias provided by `Docusaurus` that points to the `website` directory. Example:

```
- import Foo from '../src/components/Foo';
+ import Foo from '@site/src/components/Foo';
```

Link docs by file paths

Refer to other docs by relative file paths with the `.md` extension, so that `Docusaurus` can rewrite them to actual URL paths during building. Files will be linked to the correct corresponding version.

The `[@hello](hello.mdx#paginate)` document is great!

See the `[Tutorial](../getting-started/tutorial.mdx)` for more info.

Global or versioned collocated assets

You should decide if assets like images and files are per-version or shared between versions.

If your assets should be versioned, put them in the docs version, and use relative paths:

```
![img alt](./myImage.png)
```

```
[download this file](./file.pdf)
```

If your assets are global, put them in `/static` and use absolute paths:

```
![img alt](/myImage.png)
```

```
[download this file](/file.pdf)
```

Docs Multi-instance

The `@docusaurus/plugin-content-docs` plugin can support [multi-instance](#).

NOTE

This feature is only useful for [versioned documentation](#). It is recommended to be familiar with docs versioning before reading this page. If you just want [multiple sidebars](#), you can do so within one plugin.

Use-cases

Sometimes you want a Docusaurus site to host 2 distinct sets of documentation (or more).

These documentations may even have different versioning/release lifecycles.

Mobile SDKs documentation

If you build a cross-platform mobile SDK, you may have 2 documentations:

- Android SDK documentation (`v1.0`, `v1.1`)
- iOS SDK documentation (`v1.0`, `v2.0`)

In this case, you can use a distinct docs plugin instance per mobile SDK documentation.

WARNING

If each documentation instance is very large, you should rather create 2 distinct Docusaurus sites.

If someone edits the iOS documentation, is it really useful to rebuild everything, including the whole Android documentation that did not change?

Versioned and unversioned doc

Sometimes, you want some documents to be versioned, while other documents are more "global", and it feels useless to version them.

We use this pattern on the Docusaurus website itself:

- The `/docs/*` section is versioned

- The `/community/*` section is unversioned

Setup

Suppose you have 2 documentations:

- Product: some versioned doc about your product
- Community: some unversioned doc about the community around your product

In this case, you should use the same plugin twice in your site configuration.

WARNING

`@docusaurus/preset-classic` already includes a docs plugin instance for you!

When using the preset:

`docusaurus.config.js`

```
export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          // id: 'product', // omitted => default instance
          path: 'product',
          routebasePath: 'product',
          sidebarPath: './sidebarsProduct.js',
          // ... other options
        },
      },
    ],
  ],
  plugins: [
    [
      '@docusaurus/plugin-content-docs',
      {
        id: 'community',
        path: 'community',
        routebasePath: 'community',
        sidebarPath: './sidebarsCommunity.js',
        // ... other options
      },
    ],
  ],
}
```

```
],  
};
```

When not using the preset:

docusaurus.config.js

```
export default {  
  plugins: [  
    [  
      '@docusaurus/plugin-content-docs',  
      {  
        // id: 'product', // omitted => default instance  
        path: 'product',  
        routebasePath: 'product',  
        sidebarPath: './sidebarsProduct.js',  
        // ... other options  
      },  
    ],  
    [  
      '@docusaurus/plugin-content-docs',  
      {  
        id: 'community',  
        path: 'community',  
        routebasePath: 'community',  
        sidebarPath: './sidebarsCommunity.js',  
        // ... other options  
      },  
    ],  
  ],  
};
```

Don't forget to assign a unique `id` attribute to plugin instances.

NOTE

We consider that the `product` instance is the most important one, and make it the "default" instance by not assigning any ID.

Versioned paths

Each plugin instance will store versioned docs in a distinct folder.

The default plugin instance will use these paths:

- `website/versions.json`
- `website/versioned_docs`
- `website/versioned_sidebars`

The other plugin instances (with an `id` attribute) will use these paths:

- `website/[pluginId]_versions.json`
- `website/[pluginId]_versioned_docs`
- `website/[pluginId]_versioned_sidebars`

TIP

You can omit the `id` attribute (defaults to `default`) for one of the docs plugin instances.

The instance paths will be simpler, and retro-compatible with a single-instance setup.

Tagging new versions

Each plugin instance will have its own CLI command to tag a new version. They will be displayed if you run:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run docusaurus -- --help
```

To version the product/default docs plugin instance:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run docusaurus docs:version 1.0.0
```

To version the non-default/community docs plugin instance:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run docusaurus docs:version:community 1.0.0
```

Docs navbar items

Each docs-related **theme navbar items** take an optional `docsPluginId` attribute.

For example, if you want to have one version dropdown for each mobile SDK (iOS and Android), you could do:

docusaurus.config.js

```
export default {
  themeConfig: {
    navbar: {
      items: [
        {
          type: 'docsVersionDropdown',
          docsPluginId: 'ios',
        },
        {
          type: 'docsVersionDropdown',
          docsPluginId: 'android',
        },
      ],
    },
  },
};
```

Blog

The blog feature enables you to deploy a full-featured blog in no time.



INFO

Check the [Blog Plugin API Reference documentation](#) for an exhaustive list of options.

Initial setup

To set up your site's blog, start by creating a `blog` directory.

Then, add an item link to your blog within `docusaurus.config.js`:

`docusaurus.config.js`

```
export default {
  themeConfig: {
    // ...
    navbar: {
      items: [
        // ...
        {to: 'blog', label: 'Blog', position: 'left'}, // or position: 'right'
      ],
    },
  },
};
```

Adding posts

To publish in the blog, create a Markdown file within the blog directory.

For example, create a file at `website/blog/2019-09-05-hello-docusaurus.md`:

`website/blog/2019-09-05-hello-docusaurus.md`

```
---
title: Welcome Docusaurus
description: This is my first post on Docusaurus.
slug: welcome-docusaurus-v2
authors:
```

```
- name: Joel Marcey
  title: Co-creator of Docusaurus 1
  url: https://github.com/JoelMarcey
  image_url: https://github.com/JoelMarcey.png
  socials:
    x: joelmarcey
    github: JoelMarcey
- name: Sébastien Lorber
  title: Docusaurus maintainer
  url: https://sebastienlorber.com
  image_url: https://github.com/slorber.png
  socials:
    x: sebastienlorber
    github: slorber
tags: [hello, docusaurus-v2]
image: https://i.imgur.com/mErPwqL.png
hide_table_of_contents: false
---
```

Welcome to this blog. This blog is created with [\[**Docusaurus**\]](#) (<https://docusaurus.io/>).

<!-- truncate -->

This is my first post on Docusaurus.

A whole bunch of exploration to follow.

The [front matter](#) is useful to add more metadata to your blog post, for example, author information, but Docusaurus will be able to infer all necessary metadata without the front matter. For all possible fields, see [the API documentation](#).

Blog list

The blog's index page (by default, it is at `/blog`) is the *blog list page*, where all blog posts are collectively displayed.

Use the `<!--truncate-->` marker in your blog post to represent what will be shown as the summary when viewing all published blog posts. Anything above `<!--truncate-->` will be part of the summary. Note that the portion above the truncate marker must be standalone renderable Markdown. For example:

```
website/blog/my-post.md
```

```
---
```

```
title: Markdown blog truncation example
```

```
--
```

All these will be part of the blog post summary.

```
<!-- truncate -->
```

But anything from here on down will not be.

For files using the `.mdx` extension, use a [MDX](#) comment `{/* truncate */}` instead:

```
website/blog/my-post.mdx
```

```
---
```

```
title: MDX blog truncation Example
```

```
--
```

All these will be part of the blog post summary.

```
{/* truncate */}
```

But anything from here on down will not be.

By default, 10 posts are shown on each blog list page, but you can control pagination with the `postsPerPage` option in the plugin configuration. If you set `postsPerPage: 'ALL'`, pagination will be disabled and all posts will be displayed on the first page. You can also add a meta description to the blog list page for better SEO:

```
docusaurus.config.js
```

```
export default {
  // ...
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        blog: {
          blogTitle: 'Docusaurus blog!',
          blogDescription: 'A Docusaurus powered blog!',
          postsPerPage: 'ALL',
        },
      },
    ],
  ],
}
```

```
],  
};
```

Blog sidebar

The blog sidebar displays recent blog posts. The default number of items shown is 5, but you can customize with the `blogSidebarCount` option in the plugin configuration. By setting `blogSidebarCount: 0`, the sidebar will be completely disabled, with the container removed as well. This will increase the width of the main container. Specially, if you have set `blogSidebarCount: 'ALL'`, all posts will be displayed.

You can also alter the sidebar heading text with the `blogSidebarTitle` option. For example, if you have set `blogSidebarCount: 'ALL'`, instead of the default "Recent posts", you may rather make it say "All posts":

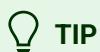
```
docusaurus.config.js
```

```
export default {  
  presets: [  
    [  
      '@docusaurus/preset-classic',  
      {  
        blog: {  
          blogSidebarTitle: 'All posts',  
          blogSidebarCount: 'ALL',  
        },  
      },  
    ],  
  ],  
};
```

Blog post date

Docusaurus will extract a `YYYY-MM-DD` date from many patterns such as `YYYY-MM-DD-my-blog-post-title.md` or `YYYY/MM/DD/my-blog-post-title.md`. This enables you to easily group blog posts by year, by month, or to use a flat structure.

▼ Supported date extraction patterns



TIP

Using a folder can be convenient to co-locate blog post images alongside the Markdown file.

This naming convention is optional, and you can also provide the date as front matter. Since the front matter follows YAML syntax where the datetime notation is supported, you can use front matter if you need more fine-grained publish dates. For example, if you have multiple posts published on the same day, you can order them according to the time of the day:

```
earlier-post.md
```

```
---
```

```
date: 2021-09-13T10:00
```

```
---
```

```
later-post.md
```

```
---
```

```
date: 2021-09-13T18:00
```

```
---
```

Blog post authors

Use the `authors` front matter field to declare blog post authors. An author should have at least a `name` or an `image_url`. Docusaurus uses information like `url`, `email`, and `title`, but any other information is allowed.

Inline authors

Blog post authors can be declared directly inside the front matter:

[Single author](#)

[Multiple authors](#)

```
my-blog-post.md
```

```
---
```

```
authors:
```

```
  name: Joel Marcey
  title: Co-creator of Docusaurus 1
  url: https://github.com/JoelMarcey
  image_url: https://github.com/JoelMarcey.png
```

```
email: jimarcey@gmail.com
```

```
socials:
```

```
x: joelmarcey
```

```
github: JoelMarcey
```

```
---
```

TIP

This option works best to get started, or for casual, irregular authors.

INFO

Prefer using the `authors` front matter, but the legacy `author_*` front matter remains supported:

```
my-blog-post.md
```

```
---
```

```
author: Joel Marcey
```

```
author_title: Co-creator of Docusaurus 1
```

```
author_url: https://github.com/JoelMarcey
```

```
author_image_url: https://github.com/JoelMarcey.png
```

```
---
```

Global authors

For regular blog post authors, it can be tedious to maintain authors' information inlined in each blog post.

It is possible to declare those authors globally in a configuration file:

```
website/blog/authors.yml
```

```
jmarcey:
```

```
  name: Joel Marcey
```

```
  title: Co-creator of Docusaurus 1
```

```
  url: https://github.com/JoelMarcey
```

```
  image_url: https://github.com/JoelMarcey.png
```

```
  email: jimarcey@gmail.com
```

```
  socials:
```

```
    x: joelmarcey
```

```
    github: JoelMarcey
```

```
slorber:
```

```
  name: Sébastien Lorber
```

```
  title: Docusaurus maintainer
```

```
url: https://sebastienlorber.com
image_url: https://github.com/slrorber.png
socials:
  x: sebastienlorber
  github: slrorber
```

TIP

Use the `authorsMapPath` plugin option to configure the path. JSON is also supported.

In blog posts front matter, you can reference the authors declared in the global configuration file:

Single author **Multiple authors**

my-blog-post.md

```
---
```

`authors: jmarcey`

```
--
```

INFO

The `authors` system is very flexible and can suit more advanced use-case:

- ▼ Mix inline authors and global authors
- ▼ Local override of global authors
- ▼ Localize the author's configuration file

An author, either declared through front matter or through the authors map, needs to have a name or an avatar, or both. If all authors of a post don't have names, Docusaurus will display their avatars compactly. See [this test post](#) for the effect.

FEED GENERATION

RSS feeds require the author's email to be set for the author to appear in the feed.

Authors pages

The authors pages feature is optional, and mainly useful for multi-author blogs.

You can activate it independently for each author by adding a `page: true` attribute to the [global author configuration](#):

```
website/blog/authors.yml
```

```
slorber:  
  name: Sébastien Lorber  
  page: true # Turns the feature on - route will be /authors/slorber  
  
jmarcey:  
  name: Joel Marcey  
  page:  
    # Turns the feature on - route will be /authors/custom-author-url  
    permalink: '/custom-author-url'
```

The blog plugin will now generate:

- a dedicated author page for each author ([example](#)) listing all the blog posts they contributed to
- an authors index page ([example](#)) listing all these authors, in the order they appear in `authors.yml`

ABOUT INLINE AUTHORS

Only [global authors](#) can activate this feature. [Inline authors](#) are not supported.

Blog post tags

Tags are declared in the front matter and introduce another dimension of categorization.

It is possible to define tags inline, or to reference predefined tags declared in a [tags file](#) (optional, usually `blog/tags.yml`).

In the following example:

- `docusaurus` references a predefined tag key declared in `blog/tags.yml`
- `Releases` is an inline tag, because it does not exist in `blog/tags.yml`

```
blog/my-post.md
```

```
---  
title: 'My blog post'  
tags:  
  - Releases  
  - docusaurus
```

Content

blog/tags.yml

```
docusaurus:  
  label: 'Docusaurus'  
  permalink: '/docusaurus'  
  description: 'Blog posts related to the Docusaurus framework'
```

Reading time

Docusaurus generates a reading time estimation for each blog post based on word count. We provide an option to customize this.

docusaurus.config.js

```
export default {  
  presets: [  
    [  
      '@docusaurus/preset-classic',  
      {  
        blog: {  
          showReadingTime: true, // When set to false, the "x min read" won't be  
          shown  
          readingTime: ({content, locale, frontMatter, defaultReadingTime}) =>  
            defaultReadingTime({  
              content,  
              locale,  
              options: {wordsPerMinute: 300},  
            }),  
        },  
      },  
    ],  
  ],  
};
```

The `readingTime` callback receives the following parameters:

- `content`: the blog content text as a string
- `frontMatter`: the front matter as a record of string keys and their values
- `locale`: the locale of the current Docusaurus site

- `defaultReadingTime`: the default built-in reading time function. It returns a number (reading time in minutes) or `undefined` (disable reading time for this page).

The default reading time is able to accept additional options:

- `wordsPerMinute` as a number (default: 300)



TIP

Use the callback for all your customization needs:

[Per-post disabling](#)

[Passing options](#)

[Using custom algorithms](#)

Disable reading time on one page:

`docusaurus.config.js`

```
export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        blog: {
          showReadingTime: true,
          readingTime: ({content, locale, frontMatter, defaultReadingTime}) =>
            frontMatter.hide_reading_time
              ? undefined
              : defaultReadingTime({content, locale}),
        },
      },
    ],
  ],
};
```

Usage:

```
---
hide_reading_time: true
---
```

This page will no longer display the reading time stats!

Feed

You can generate RSS / Atom / JSON feed by passing `feedOptions`. By default, RSS and Atom feeds are generated. To disable feed generation, set `feedOptions.type` to `null`.

```
type FeedType = 'rss' | 'atom' | 'json';

type BlogOptions = {
  feedOptions?: {
    type?: FeedType | 'all' | FeedType[] | null;
    title?: string;
    description?: string;
    copyright: string;

    language?: string; // possible values: http://www.w3.org/TR/REC-html40/struct/dirlang.html#langcodes
    limit?: number | false | null; // defaults to 20
    // XSLT permits browsers to style and render nicely the feed XML files
    xslt?: {
      | boolean
      | {
        //
        rss?: string | boolean;
        atom?: string | boolean;
      };
    };
    // Allow control over the construction of BlogFeedItems
    createFeedItems?: (params: {
      blogPosts: BlogPost[];
      siteConfig: DocusaurusConfig;
      outDir: string;
      defaultCreateFeedItems: (params: {
        blogPosts: BlogPost[];
        siteConfig: DocusaurusConfig;
        outDir: string;
      }) => Promise<BlogFeedItem[]>;
      }) => Promise<BlogFeedItem[]>;
    });
  };
};
```

Example usage:

```
docusaurus.config.js
```

```
export default {
  // ...
  presets: [
    [
```

```
'@docusaurus/preset-classic',
{
  blog: {
    feedOptions: {
      type: 'all',
      copyright: `Copyright © ${new Date().getFullYear()} Facebook, Inc.`,
      createFeedItems: async (params) => {
        const {blogPosts, defaultCreateFeedItems, ...rest} = params;
        return defaultCreateFeedItems({
          // keep only the 10 most recent blog posts in the feed
          blogPosts: blogPosts.filter((item, index) => index < 10),
          ...rest,
        });
      },
    },
  },
  [,
  ],
},
};


```

The feeds can be found at:

RSS Atom JSON

<https://example.com/blog/rss.xml>

Advanced topics

Blog-only mode

You can run your Docusaurus site without a dedicated landing page and instead have your blog's post list page as the index page. Set the `routebasePath` to be `'/'` to serve the blog through the root route `example.com/` instead of the subroute `example.com/blog/`.

`docusaurus.config.js`

```
export default {
  // ...
  presets: [
    [
      ...
    ]
  ]
};
```

```
'@docusaurus/preset-classic',
{
  docs: false, // Optional: disable the docs plugin
  blog: {
    routebasePath: '/', // Serve the blog at the site's root
    /* other blog options */
  },
},
],
],
};

};
```

WARNING

Don't forget to delete the existing homepage at `./src/pages/index.js` or else there will be two files mapping to the same route!

WARNING

If you disable the docs plugin, don't forget to delete references to the docs plugin elsewhere in your configuration file. Notably, make sure to remove the docs-related navbar items.

TIP

There's also a "Docs-only mode" for those who only want to use the docs. Read [Docs-only mode](#) for detailed instructions or a more elaborate explanation of `routebasePath`.

Multiple blogs

By default, the classic theme assumes only one blog per website and hence includes only one instance of the blog plugin. If you would like to have multiple blogs on a single website, it's possible too! You can add another blog by specifying another blog plugin in the `plugins` option for `docusaurus.config.js`.

Set the `routebasePath` to the URL route that you want your second blog to be accessed on. Note that the `routebasePath` here has to be different from the first blog or else there could be a collision of paths! Also, set `path` to the path to the directory containing your second blog's entries.

As documented for [multi-instance plugins](#), you need to assign a unique ID to the plugins.

`docusaurus.config.js`

```
export default {
  // ...
  plugins: [
    [
      {
        id: 'secondBlog',
        plugin: '@docusaurus/plugin-content-blog',
        config: {
          routebasePath: '/second-blog',
          path: 'src/second-blog'
        }
      }
    ]
  ]
};
```

```
'@docusaurus/plugin-content-blog',
{
  /**
   * Required for any multi-instance plugin
   */
  id: 'second-blog',
  /**
   * URL route for the blog section of your site.
   * *DO NOT* include a trailing slash.
   */
  routebasePath: 'my-second-blog',
  /**
   * Path to data on filesystem relative to site dir.
   */
  path: './my-second-blog',
},
],
],
};

};
```

As an example, we host a second blog [here](#).

MDX and React

Docusaurus has built-in support for [MDX](#), which allows you to write JSX within your Markdown files and render them as React components.

Check out the [MDX docs](#) to see what fancy stuff you can do with MDX.

DEBUGGING MDX

The MDX format is quite strict, and you may get compilation errors.

Use the [MDX playground](#) to debug them and make sure your syntax is valid.

INFO

Prettier, the most popular formatter, [supports only the legacy MDX v1](#). If you get an unintentional formatting result, you may want to add `{/* prettier-ignore */}` before the problematic area, or add `*.mdx` to your `.prettierignore`, until Prettier has proper support for MDX v3. [One of the main authors of MDX recommends remark-cli with remark-mdx](#).

Exporting components

To define any custom component within an MDX file, you have to export it: only paragraphs that start with `export` will be parsed as components instead of prose.

```
export const Highlight = ({children, color}) => (
  <span
    style={{
      backgroundColor: color,
      borderRadius: '2px',
      color: '#fff',
      padding: '0.2rem',
    }}>
    {children}
  </span>
);

<Highlight color="#25c2a0">Docusaurus green</Highlight> and <Highlight
color="#1877F2">Facebook blue</Highlight> are my favorite colors.

I can write **Markdown** alongside my _JSX_!
```

Notice how it renders both the markup from your React component and the Markdown syntax:

Docusaurus green and Facebook blue are my favorite colors.

I can write **Markdown** alongside my JSX!

⚠️ MDX IS JSX

Since all doc files are parsed using MDX, anything that looks like HTML is actually JSX. Therefore, if you need to inline-style a component, follow JSX flavor and provide style objects.

```
/* Instead of this: */
<span style="background-color: red">Foo</span>
/* Use this: */
<span style={{backgroundColor: 'red'}}>Foo</span>
```

Importing components

You can also import your own components defined in other files or third-party components installed via npm.

```
<!-- Docusaurus theme component -->
import TOCInline from '@theme/TOCInline';
<!-- External component -->
import Button from '@mui/material/Button';
<!-- Custom component -->
import BrowserWindow from '@site/src/components/BrowserWindow';
```

💡 TIP

The `@site` alias points to your website's directory, usually where the `docusaurus.config.js` file is. Using an alias instead of relative paths (`'../../src/components/BrowserWindow'`) saves you from updating import paths when moving files around, or when [versioning docs](#) and [translating](#).

While declaring components within Markdown is very convenient for simple cases, it becomes hard to maintain because of limited editor support, risks of parsing errors, and low reusability. Use a separate `.js` file when your component involves complex JS logic:

```
import React from 'react';

export default function Highlight({children, color}) {
  return (
    <span
      style={{
        backgroundColor: color,
        borderRadius: '2px',
        color: '#fff',
        padding: '0.2rem',
      }}>
      {children}
    </span>
  );
}
```

```
import Highlight from '@site/src/components/Highlight';

<Highlight color="#25c2a0">Docusaurus green</Highlight>
```



TIP

If you use the same component across a lot of files, you don't need to import it everywhere—consider adding it to the global scope. [See below](#)

MDX component scope

Apart from [importing a component](#) and [exporting a component](#), a third way to use a component in MDX is to **register it to the global scope**, which will make it automatically available in every MDX file, without any import statements.

For example, given this MDX file:

```
- a
- list!
```

And some `<Highlight>custom markup</Highlight>...`

It will be compiled to a React component containing `ul`, `li`, `p`, and `Highlight` elements. `Highlight` is not a native html element: you need to provide your own React component implementation for it.

In Docusaurus, the MDX component scope is provided by the `@theme/MDXComponents` file. It's not a React component, *per se*, unlike most other exports under the `@theme/` alias: it is a record from tag names like `Highlight` to their React component implementations.

If you `swizzle` this component, you will find all tags that have been implemented, and you can further customize our implementation by swizzling the respective sub-component, like `@theme/MDXComponents/Code` (which is used to render `Markdown code blocks`).

If you want to register extra tag names (like the `<Highlight>` tag above), you should consider `wrapping` `@theme/MDXComponents`, so you don't have to maintain all the existing mappings. Since the `swizzle` CLI doesn't allow wrapping non-component files yet, you should manually create the wrapper:

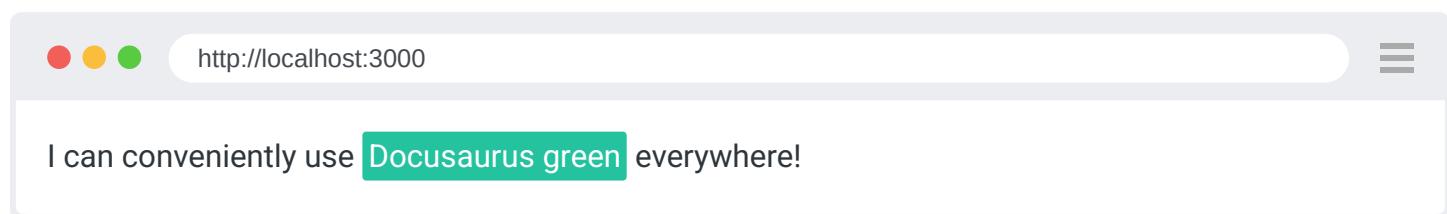
```
src/theme/MDXComponents.js
```

```
import React from 'react';
// Import the original mapper
import MDXComponents from '@theme-original/MDXComponents';
import Highlight from '@site/src/components/Highlight';

export default {
  // Re-use the default mapping
  ...MDXComponents,
  // Map the "<Highlight>" tag to our Highlight component
  // `Highlight` will receive all props that were passed to `<Highlight>` in MDX
  Highlight,
};
```

And now, you can freely use `<Highlight>` in every page, without writing the import statement:

I can conveniently use `<Highlight color="#25c2a0">Docusaurus green</Highlight>` everywhere!



⚠️ WARNING

We use **upper-case** tag names like `Highlight` on purpose.

From MDX v3+ onward (Docusaurus v3+), lower-case tag names are always rendered as native html elements, and will not use any component mapping you provide.

⚠️ WARNING

This feature is powered by an `MDXProvider`. If you are importing Markdown in a React page, you have to supply this provider yourself through the `MDXContent` theme component.

src/pages/index.js

```
import React from 'react';
import FeatureDisplay from './_featureDisplay.mdx';
import MDXContent from '@theme/MDXContent';

export default function LandingPage() {
  return (
    <div>
      <MDXContent>
        <FeatureDisplay />
      </MDXContent>
    </div>
  );
}
```

If you don't wrap your imported MDX with `MDXContent`, the global scope will not be available.

Markdown and JSX interoperability

Docusaurus v3 is using `MDX v3`.

The `MDX syntax` is mostly compatible with `CommonMark`, but is much stricter because your `.mdx` files can use JSX and are compiled into real React components (check the [playground](#)).

Some valid CommonMark features won't work with MDX ([more info](#)), notably:

- Indented code blocks: use triple backticks instead
- Autolinks (`<http://localhost:3000>`): use regular link syntax instead
(`http://localhost:3000`)
- HTML syntax (`<p style="color: red;">`): use JSX instead (`<p style={{color: 'red'}}>`)
- Unescaped `{` and `<`: escape them with `\` instead (`\{` and `\<`)

🔥 EXPERIMENTAL COMMONMARK SUPPORT

Docusaurus v3 makes it possible to opt-in for a less strict, standard [CommonMark](#) support with the following options:

- The `mdx.format: md` front matter
- The `.md` file extension combined with the `siteConfig.markdown.format: "detect"` configuration

This feature is **experimental** and currently has a few [limitations](#).

Importing code snippets

You can not only import a file containing a component definition, but also import any code file as raw text, and then insert it in a code block, thanks to [Webpack raw-loader](#). In order to use `raw-loader`, you first need to install it in your project:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

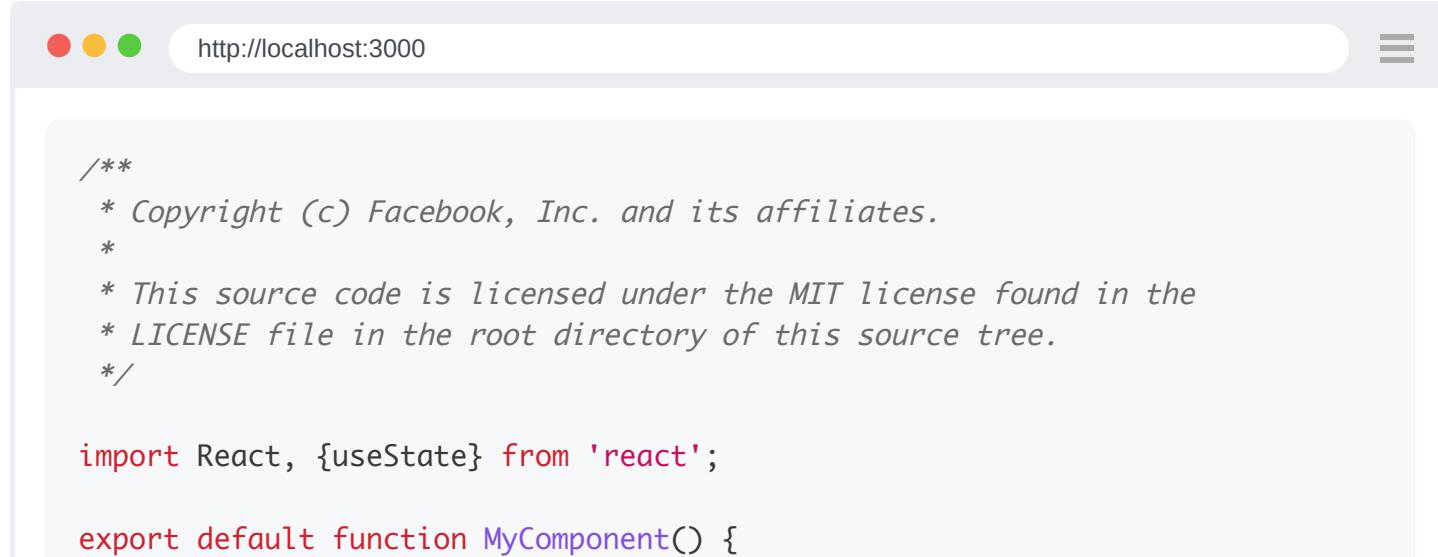
```
npm install --save raw-loader
```

Now you can import code snippets from another file as it is:

myMarkdownFile.mdx

```
import CodeBlock from '@theme/CodeBlock';
import MyComponentSource from '!!raw-loader!./myComponent';

<CodeBlock language="jsx">{MyComponentSource}</CodeBlock>
```



The screenshot shows a browser window with the URL `http://localhost:3000` in the address bar. The page content displays a code block with the following code:

```
/**
 * Copyright (c) Facebook, Inc. and its affiliates.
 *
 * This source code is licensed under the MIT license found in the
 * LICENSE file in the root directory of this source tree.
 */

import React, {useState} from 'react';

export default function MyComponent() {
```

```
const [bool, setBool] = useState(false);
return (
  <div>
    <p>MyComponent rendered !</p>
    <p>bool={bool ? 'true' : 'false'}</p>
    <p>
      <button onClick={() => setBool((b) => !b)}>toggle bool</button>
    </p>
  </div>
);
}
```

See [using code blocks in JSX](#) for more details of the `<CodeBlock>` component.

 NOTE

You have to use `<CodeBlock>` rather than the Markdown triple-backtick `````, because the latter will ship out any of its content as-is, but you want to interpolate the imported text here.

 WARNING

This feature is experimental and might be subject to breaking API changes in the future.

Importing Markdown

You can use Markdown files as components and import them elsewhere, either in Markdown files or in React pages. Each MDX file default-exports its page content as a React component. In the `import` statement, you can default-import this component with any name, but it must be capitalized following React's naming rules.

By convention, using the `_ filename prefix` will not create any doc page and means the Markdown file is a "partial", to be imported by other files.

`_markdown-partial-example.mdx`

```
<span>Hello {props.name}</span>
```

This is text some content from `'_markdown-partial-example.mdx'`.

`someOtherDoc.mdx`

```
import PartialExample from './_markdown-partial-example.mdx';
<PartialExample name="Sebastien" />
```

Hello Sebastien
This is text some content from `_markdown-partial-example.md`.

This way, you can reuse content among multiple pages and avoid duplicating materials.

Available exports

Within the MDX page, the following variables are available as globals:

- `frontMatter`: the front matter as a record of string keys and values;
- `toc`: the table of contents, as a tree of headings. See also [Inline TOC](#) for a more concrete use-case.
- `contentTitle`: the Markdown title, which is the first `h1` heading in the Markdown text. It's `undefined` if there isn't one (e.g. title specified in the front matter).

```
import TOCInline from '@theme/TOCInline';
import CodeBlock from '@theme/CodeBlock';
```

The table `of` contents `for this` page, `serialized`:

```
<CodeBlock className="language-json">{JSON.stringify(toc, null, 2)}</CodeBlock>
```

The front matter `of this page`:

```
<ul>
  {Object.entries(frontMatter).map(([key, value]) => <li key={key}><b>{key}</b>:<br/>{value}</li>)}
</ul>
```

```
<p>The title of this page is: <b>{contentTitle}</b></p>
```

The table of contents for this page, serialized:

```
[
  {
    "value": "Exporting components",
    "id": "exporting-components",
    "level": 3
  },
  {
    "value": "Importing components",
    "id": "importing-components",
    "level": 3
  },
  {
    "value": "MDX component scope",
    "id": "mdx-component-scope",
    "level": 3
  },
  {
    "value": "Markdown and JSX interoperability",
    "id": "markdown-and-jsx-interoperability",
    "level": 3
  },
  {
    "value": "Importing code snippets",
    "id": "importing-code-snippets",
    "level": 2
  },
  {
    "value": "Importing Markdown",
    "id": "importing-markdown",
    "level": 2
  },
  {
    "value": "Available exports",
    "id": "available-exports",
    "level": 2
  }
]
```

The front matter of this page:

- **id:** react
- **description:** Using the power of React in Docusaurus Markdown documents, thanks to MDX
- **slug:** /markdown-features/react

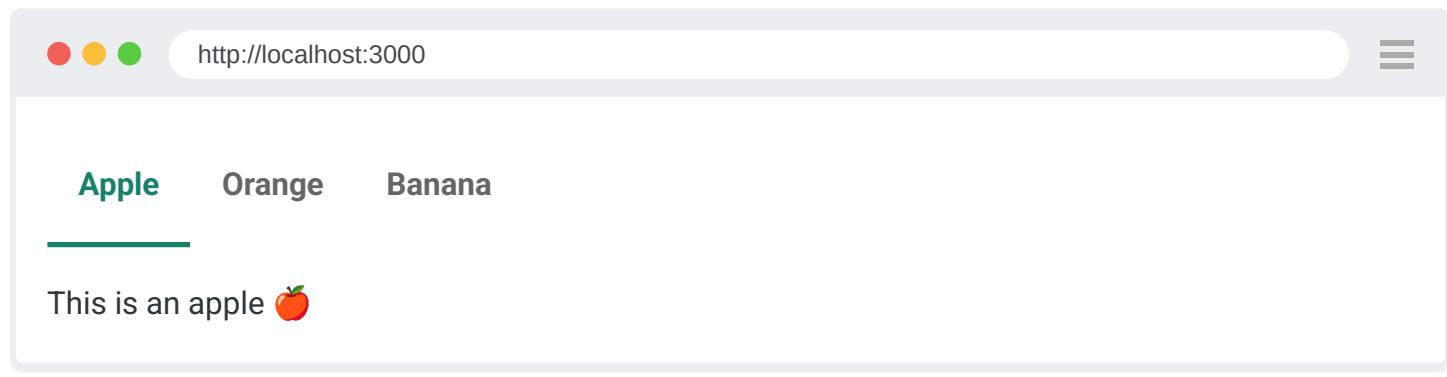
The title of this page is: **MDX and React**

Tabs

Docusaurus provides the `<Tabs>` component that you can use in Markdown thanks to [MDX](#):

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

<Tabs>
  <TabItem value="apple" label="Apple" default>
    This is an apple 🍎
  </TabItem>
  <TabItem value="orange" label="Orange">
    This is an orange 🍊
  </TabItem>
  <TabItem value="banana" label="Banana">
    This is a banana 🍌
  </TabItem>
</Tabs>
```



It is also possible to provide `values` and `defaultValue` props to `Tabs`:

```
<Tabs
  defaultValue="apple"
  values={[
    {label: 'Apple', value: 'apple'},
    {label: 'Orange', value: 'orange'},
    {label: 'Banana', value: 'banana'},
  ]}>
  <TabItem value="apple">This is an apple 🍎</TabItem>
  <TabItem value="orange">This is an orange 🍊</TabItem>
  <TabItem value="banana">This is a banana 🍌</TabItem>
</Tabs>
```

Apple Orange Banana

This is an apple 

▼ `Tabs` props take precedence over the `TabItem` props:



TIP

By default, all tabs are rendered eagerly during the build process, and search engines can index hidden tabs.

It is possible to only render the default tab with `<Tabs lazy />`.

Displaying a default tab

The first tab is displayed by default, and to override this behavior, you can specify a default tab by adding `default` to one of the tab items. You can also set the `defaultValue` prop of the `Tabs` component to the label value of your choice. For example, in the example above, either setting `default` for the `value="apple"` tab or setting `defaultValue="apple"` for the tabs forces the "Apple" tab to be open by default.

Docusaurus will throw an error if a `defaultValue` is provided for the `Tabs` but it refers to a non-existing value. If you want none of the tabs to be shown by default, use `defaultValue={null}`.

Syncing tab choices

You may want choices of the same kind of tabs to sync with each other. For example, you might want to provide different instructions for users on Windows vs users on macOS, and you want to change all OS-specific instructions tabs in one click. To achieve that, you can give all related tabs the same `groupId` prop. Note that doing this will persist the choice in `localStorage` and all `<Tab>` instances with the same `groupId` will update automatically when the value of one of them is changed. Note that group IDs are globally namespaced.

```
<Tabs groupId="operating-systems">
  <TabItem value="win" label="Windows">Use Ctrl + C to copy.</TabItem>
  <TabItem value="mac" label="macOS">Use Command + C to copy.</TabItem>
```

```
</Tabs>
```

```
<Tabs groupId="operating-systems">
  <TabItem value="win" label="Windows">Use Ctrl + V to paste.</TabItem>
  <TabItem value="mac" label="macOS">Use Command + V to paste.</TabItem>
</Tabs>
```

http://localhost:3000

Windows macOS

Use Ctrl + C to copy.

Windows macOS

Use Ctrl + V to paste.

For all tab groups that have the same `groupId`, the possible values do not need to be the same. If one tab group is chosen a value that does not exist in another tab group with the same `groupId`, the tab group with the missing value won't change its tab. You can see that from the following example. Try to select Linux, and the above tab groups don't change.

```
<Tabs groupId="operating-systems">
  <TabItem value="win" label="Windows">
    I am Windows.
  </TabItem>
  <TabItem value="mac" label="macOS">
    I am macOS.
  </TabItem>
  <TabItem value="linux" label="Linux">
    I am Linux.
  </TabItem>
</Tabs>
```

http://localhost:3000

Windows macOS Linux

I am Windows.

I am Windows.

Tab choices with different group IDs will not interfere with each other:

```
<Tabs groupId="operating-systems">
  <TabItem value="win" label="Windows">Windows in windows.</TabItem>
  <TabItem value="mac" label="macOS">macOS is macOS.</TabItem>
</Tabs>
```

```
<Tabs groupId="non-mac-operating-systems">
  <TabItem value="win" label="Windows">Windows is windows.</TabItem>
  <TabItem value="unix" label="Unix">Unix is unix.</TabItem>
</Tabs>
```

The screenshot shows a browser window with the URL <http://localhost:3000>. There are two tabs visible at the top: "Windows" and "macOS". The "Windows" tab is active, showing the content "Windows in windows.". Below it, another tab labeled "Unix" is shown, with the content "Windows is windows." This demonstrates that tabs with different group IDs do not interfere with each other.

Customizing tabs

You might want to customize the appearance of a certain set of tabs. You can pass the string in `className` prop, and the specified CSS class will be added to the `Tabs` component:

```
<Tabs className="unique-tabs">
  <TabItem value="Apple">This is an apple 🍎</TabItem>
  <TabItem value="Orange">This is an orange 🍊</TabItem>
  <TabItem value="Banana">This is a banana 🍌</TabItem>
</Tabs>
```

The screenshot shows a browser window with the URL <http://localhost:3000>. It displays three tabs with custom icons: an apple, an orange, and a banana. The tabs are part of a `Tabs` component with the `unique-tabs` class applied, demonstrating how to customize the appearance of tabs.

Apple

Orange

Banana

This is an apple 

Customizing tab headings

You can also customize each tab heading independently by using the `attributes` field. The extra props can be passed to the headings either through the `values` prop in `Tabs`, or props of each `TabItem`—in the same way as you declare `label`.

some-doc.mdx

```
import styles from './styles.module.css';

<Tabs>
  <TabItem value="apple" label="Apple" attributes={{className: styles.red}}>
    This is an apple 
  </TabItem>
  <TabItem value="orange" label="Orange" attributes={{className: styles.orange}}>
    This is an orange 
  </TabItem>
  <TabItem value="banana" label="Banana" attributes={{className: styles.yellow}}>
    This is a banana 
  </TabItem>
</Tabs>
```

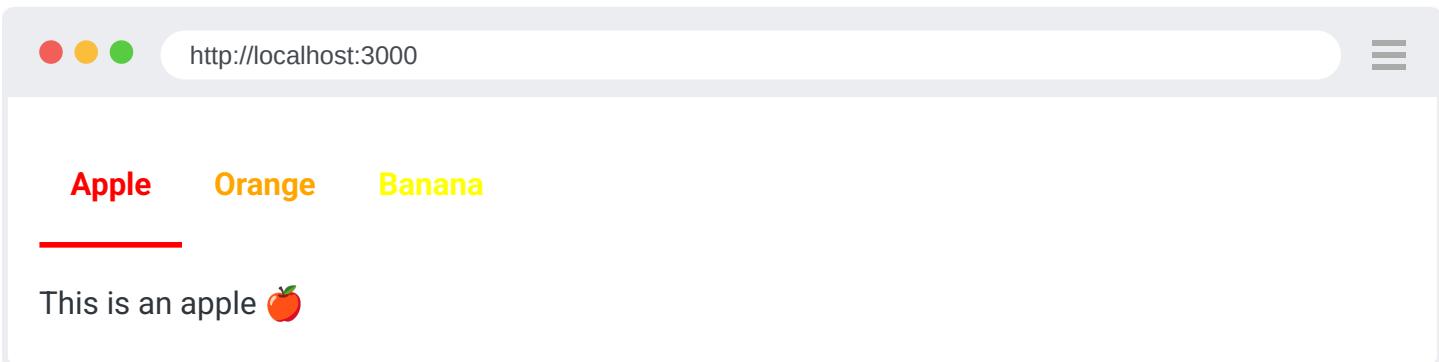
styles.module.css

```
.red {
  color: red;
}
.red[aria-selected='true'] {
  border-bottom-color: red;
}

.orange {
  color: orange;
}
.orange[aria-selected='true'] {
  border-bottom-color: orange;
}

.yellow {
```

```
color: yellow;  
}  
.yellow[aria-selected='true'] {  
  border-bottom-color: yellow;  
}
```



TIP

`className` would be merged with other default class names. You may also use a custom `data-value` field (`{'data-value': 'apple'}`) paired with CSS attribute selectors:

styles.module.css

```
li[role='tab'][data-value='apple'] {  
  color: red;  
}
```

Query string

It is possible to persist the selected tab into the url search parameters. This enables you to share a link to a page which pre-selects the tab - linking from your Android app to documentation with the Android tabs pre-selected. This feature does not provide an anchor link - the browser will not scroll to the tab.

Use the `queryString` prop to enable this feature and define the search param name to use.

```
<Tabs queryString="current-os">  
  <TabItem value="android" label="Android">  
    Android  
  </TabItem>  
  <TabItem value="ios" label="iOS">  
    iOS  
  </TabItem>  
</Tabs>
```

http://localhost:3000

Android iOS

Android

As soon as a tab is clicked, a search parameter is added at the end of the url: `?current-os=android` or `?current-os=ios`.

TIP

`queryString` can be used together with `groupId`.

For convenience, when the `queryString` prop is `true`, the `groupId` value will be used as a fallback.

```
<Tabs groupId="current-os" queryString>
  <TabItem value="android" label="Android">
    Android
  </TabItem>
  <TabItem value="ios" label="iOS">
    iOS
  </TabItem>
</Tabs>
```

http://localhost:3000

Android iOS

Android

When the page loads, the tab query string choice will be restored in priority over the `groupId` choice (using `localStorage`).

Code blocks

Code blocks within documentation are super-powered 💪.

Code title

You can add a title to the code block by adding a `title` key after the language (leave a space between them).

```
```jsx title="/src/components/HelloCodeTitle.js"
function HelloCodeTitle(props) {
 return <h1>Hello, {props.name}</h1>;
}
````
```

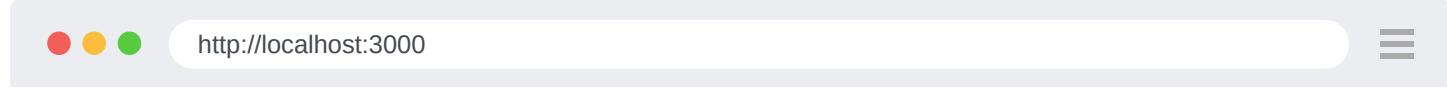


Syntax highlighting

Code blocks are text blocks wrapped around by strings of 3 backticks. You may check out [this reference](#) for the specifications of MDX.

```
```js
console.log('Every repo must come with a mascot.');
````
```

Use the matching language meta string for your code block, and Docusaurus will pick up syntax highlighting automatically, powered by [Prism React Renderer](#).



```
console.log('Every repo must come with a mascot.');
```

Theming

By default, the Prism [syntax highlighting theme](#) we use is [Palenight](#). You can change this to another theme by passing `theme` field in `prism` as `themeConfig` in your `docusaurus.config.js`.

For example, if you prefer to use the [dracula](#) highlighting theme:

`docusaurus.config.js`

```
import {themes as prismThemes} from 'prism-react-renderer';

export default {
  themeConfig: {
    prism: {
      theme: prismThemes.dracula,
    },
  },
};
```

Because a Prism theme is just a JS object, you can also write your own theme if you are not satisfied with the default. Docusaurus enhances the `github` and `vsDark` themes to provide richer highlight, and you can check our implementations for the [light](#) and [dark](#) code block themes.

Supported Languages

By default, Docusaurus comes with a subset of [commonly used languages](#).

WARNING

Some popular languages like Java, C#, or PHP are not enabled by default.

To add syntax highlighting for any of the other [Prism-supported languages](#), define it in an array of additional languages.

NOTE

Each additional language has to be a valid Prism component name. For example, Prism would map the *language* `cs` to `csharp`, but only `prism-csharp.js` exists as a *component*, so you need

to use `additionalLanguages: ['csharp']`. You can look into `node_modules/prismjs/components` to find all components (languages) available.

For example, if you want to add highlighting for the PowerShell language:

docusaurus.config.js

```
export default {
  // ...
  themeConfig: {
    prism: {
      additionalLanguages: ['powershell'],
    },
    // ...
  },
};
```

After adding `additionalLanguages`, restart Docusaurus.

If you want to add highlighting for languages not yet supported by Prism, you can swizzle `prism-include-languages`:

npm **Yarn** **pnpm** **Bun**

```
npm run swizzle @docusaurus/theme-classic prism-include-languages
```

It will produce `prism-include-languages.js` in your `src/theme` folder. You can add highlighting support for custom languages by editing `prism-include-languages.js`:

src/theme/prism-include-languages.js

```
const prismIncludeLanguages = (Prism) => {
  // ...

  additionalLanguages.forEach((lang) => {
    require(`prismjs/components/prism-${lang}`);
  });

  require('/path/to/your/prism-language-definition');

  // ...
};
```

You can refer to [Prism's official language definitions](#) when you are writing your own language definitions.

When adding a custom language definition, you do not need to add the language to the `additionalLanguages` config array, since Docusaurus only looks up the `additionalLanguages` strings in languages that Prism provides. Adding the language import in `prism-include-languages.js` is sufficient.

Line highlighting

Highlighting with comments

You can use comments with `highlight-next-line`, `highlight-start`, and `highlight-end` to select which lines are highlighted.

```
```js
function HighlightSomeText(highlight) {
 if (highlight) {
 // highlight-next-line
 return 'This text is highlighted!';
 }

 return 'Nothing highlighted';
}

function HighlightMoreText(highlight) {
 // highlight-start
 if (highlight) {
 return 'This range is highlighted!';
 }
 // highlight-end

 return 'Nothing highlighted';
}
```

```



```

    return 'Nothing highlighted';
}

function HighlightMoreText(highlight) {
  if (highlight) {
    return 'This range is highlighted!';
  }

  return 'Nothing highlighted';
}

```

Supported commenting syntax:

| Style | Syntax |
|------------|--|
| C-style | <code>/* ... */</code> and <code>// ...</code> |
| JSX-style | <code>{/* ... */}</code> |
| Bash-style | <code># ...</code> |
| HTML-style | <code><!-- ... --></code> |

We will do our best to infer which set of comment styles to use based on the language, and default to allowing *all* comment styles. If there's a comment style that is not currently supported, we are open to adding them! Pull requests welcome. Note that different comment styles have no semantic difference, only their content does.

You can set your own background color for highlighted code line in your `src/css/custom.css` which will better fit to your selected syntax highlighting theme. The color given below works for the default highlighting theme (Palenight), so if you are using another theme, you will have to tweak the color accordingly.

`/src/css/custom.css`

```

:root {
  --docusaurus-highlighted-code-line-bg: #4f77b0;
}

/* If you have a different syntax highlighting theme for dark mode. */
[data-theme='dark'] {
  /* Color which works with dark mode syntax highlighting theme */
}

```

```
--docusaurus-highlighted-code-line-bg: rgb(100, 100, 100);  
}
```

If you also need to style the highlighted code line in some other way, you can target on `theme-code-block-highlighted-line` CSS class.

Highlighting with metadata string

You can also specify highlighted line ranges within the language meta string (leave a space after the language). To highlight multiple lines, separate the line numbers by commas or use the range syntax to select a chunk of lines. This feature uses the `parse-number-range` library and you can find [more syntax](#) on their project details.

```
```jsx {1,4-6,11}  
import React from 'react';

function MyComponent(props) {
 if (props.isBar) {
 return <div>Bar</div>;
 }

 return <div>Foo</div>;
}

export default MyComponent;
```
```



A screenshot of a web browser window titled "http://localhost:3000". The page displays a code editor with the following JSX code:

```
import React from 'react';  
  
function MyComponent(props) {  
  if (props.isBar) {  
    return <div>Bar</div>;  
  }  
  
  return <div>Foo</div>;  
}  
  
export default MyComponent;
```

The code is styled with syntax highlighting. The first line, `import React from 'react';`, is in red. The function definition and its body are in purple. The condition in the if statement is in red. The return statement is in green. The final export statement is in red. The browser interface includes standard window controls (red, yellow, green buttons) and a URL bar.

 PREFER COMMENTS

Prefer highlighting with comments where you can. By inlining highlight in the code, you don't have to manually count the lines if your code block becomes long. If you add/remove lines, you also don't have to offset your line ranges.

```
- ``js` {3}
+ ``js` {4}
  function HighlightSomeText(highlight) {
    if (highlight) {
+      console.log('Highlighted text found');
      return 'This text is highlighted!';
    }

    return 'Nothing highlighted';
}
``
```

Below, we will introduce how the magic comment system can be extended to define custom directives and their functionalities. The magic comments would only be parsed if a highlight metastring is not present.

Custom magic comments

// highlight-next-line and // highlight-start etc. are called "magic comments", because they will be parsed and removed, and their purposes are to add metadata to the next line, or the section that the pair of start- and end-comments enclose.

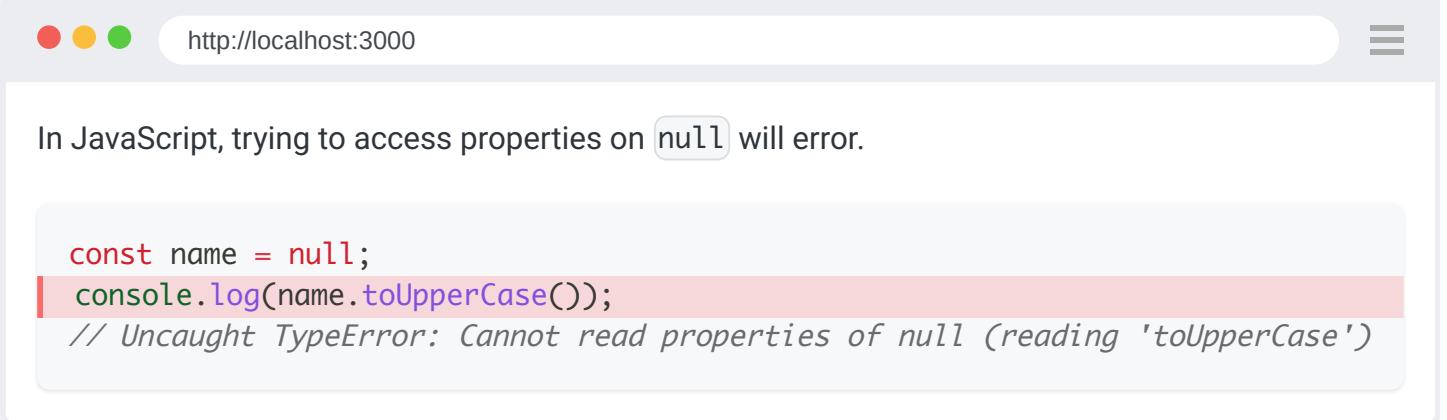
You can declare custom magic comments through theme config. For example, you can register another magic comment that adds a code-block-error-line class name:

[docusaurus.config.js](#) [src/css/custom.css](#) [myDoc.md](#)

```
export default {
  themeConfig: {
    prism: {
      magicComments: [
        // Remember to extend the default highlight class name as well!
        {
          className: 'theme-code-block-highlighted-line',
          line: 'highlight-next-line',
          block: {start: 'highlight-start', end: 'highlight-end'},
        },
        {
          className: 'code-block-error-line',
```

```
        line: 'This will error',
    },
],
},
},
};


```



In JavaScript, trying to access properties on `null` will error.

```
const name = null;
console.log(name.toUpperCase());
// Uncaught TypeError: Cannot read properties of null (reading 'toUpperCase')
```

If you use number ranges in metastring (the `{1,3-4}` syntax), Docusaurus will apply the **first magicComments entry's class name**. This, by default, is `theme-code-block-highlighted-line`, but if you change the `magicComments` config and use a different entry as the first one, the meaning of the metastring range will change as well.

You can disable the default line highlighting comments with `magicComments: []`. If there's no magic comment config, but Docusaurus encounters a code block containing a metastring range, it will error because there will be no class name to apply—the highlighting class name, after all, is just a magic comment entry.

Every magic comment entry will contain three keys: `className` (required), `line`, which applies to the directly next line, or `block` (containing `start` and `end`), which applies to the entire block enclosed by the two comments.

Using CSS to target the class can already do a lot, but you can unlock the full potential of this feature through [swizzling](#).

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run swizzle @docusaurus/theme-classic CodeBlock/Line
```

The `Line` component will receive the list of class names, based on which you can conditionally render different markup.

Line numbering

You can enable line numbering for your code block by using `showLineNumbers` key within the language meta string (don't forget to add space directly before the key).

```
```jsx showLineNumbers
import React from 'react';

export default function MyComponent(props) {
 return <div>Foo</div>;
}
```

```

A screenshot of a browser window with a light gray header bar. On the left are three colored dots (red, yellow, green). In the center is the URL <http://localhost:3000>. On the right is a menu icon. The main content area shows a code block with line numbers:

```
1 import React from 'react';
2
3 export default function MyComponent(props) {
4   return <div>Foo</div>;
5 }
```

By default, the counter starts at line number 1. It's possible to pass a custom counter start value to split large code blocks for readability:

```
```jsx showLineNumbers=3
export default function MyComponent(props) {
 return <div>Foo</div>;
}
```

```

A screenshot of a browser window with a light gray header bar. On the left are three colored dots (red, yellow, green). In the center is the URL <http://localhost:3000>. On the right is a menu icon. The main content area shows a code block with line numbers starting at 3:

```
3 export default function MyComponent(props) {
4   return <div>Foo</div>;
5 }
```

Interactive code editor

You can create an interactive coding editor with the [@docusaurus/theme-live-codeblock](#) plugin. First, add the plugin to your package.

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install --save @docusaurus/theme-live-codeblock
```

You will also need to add the plugin to your [docusaurus.config.js](#).

```
export default {
  // ...
  themes: ['@docusaurus/theme-live-codeblock'],
  // ...
};
```

To use the plugin, create a code block with `live` attached to the language meta string.

```
```jsx live
function Clock(props) {
 const [date, setDate] = useState(new Date());
 useEffect(() => {
 const timerID = setInterval(() => tick(), 1000);

 return function cleanup() {
 clearInterval(timerID);
 };
 });

 function tick() {
 setDate(new Date());
 }

 return (
 <div>
 <h2>It is {date.toLocaleTimeString()}</h2>
 </div>
);
}
```

```

The code block will be rendered as an interactive editor. Changes to the code will reflect on the result panel live.

The screenshot shows a browser window with the URL <http://localhost:3000>. The title bar says "LIVE EDITOR". The code editor contains the following code:

```
function Clock(props) {
  const [date, setDate] = useState(new Date());
  useEffect(() => {
    const timerID = setInterval(() => tick(), 1000);

    return function cleanup() {
      clearInterval(timerID);
    };
  });

  function tick() {
    setDate(new Date());
  }

  return (
    <div>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  );
}
```

The result panel below the editor shows the output: "It is 10:35:30 PM."

Imports

REACT-LIVE AND IMPORTS

It is not possible to import components directly from the react-live code editor, you have to define available imports upfront.

By default, all React imports are available. If you need more imports available, swizzle the react-live scope:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm run swizzle @docusaurus/theme-live-codeblock ReactLiveScope -- --eject
```

src/theme/ReactLiveScope/index.js

```
import React from 'react';

const ButtonExample = (props) => (
  <button
    {...props}
    style={{
      backgroundColor: 'white',
      color: 'black',
      border: 'solid red',
      borderRadius: 20,
      padding: 10,
      cursor: 'pointer',
      ...props.style,
    }}
  />
);

// Add react-live imports you need here
const ReactLiveScope = {
  React,
  ...React,
  ButtonExample,
};

export default ReactLiveScope;
```

The `ButtonExample` component is now available to use:

The screenshot shows a browser window with a dark theme. At the top, there are three circular icons (red, yellow, green) and the URL `http://localhost:3000`. Below the address bar, there's a header bar with the text "LIVE EDITOR". The main content area is divided into two sections: "LIVE EDITOR" on the left and "RESULT" on the right. In the "LIVE EDITOR" section, there is a code editor containing the following JavaScript code:

```
function MyPlayground(props) {
  return (
    <div>
      <ButtonExample onClick={() => alert('hey!')}>Click me</ButtonExample>
    </div>
  );
}
```

In the "RESULT" section, the output of the code is displayed as a single button with the text "Click me".

Click me

Imperative Rendering (`noInline`)

The `noInline` option should be used to avoid errors when your code spans multiple components or variables.

```
```jsx live noInline
const project = 'Docusaurus';

const Greeting = () => <p>Hello {project}!</p>

render(<Greeting />);
```
```

Unlike an ordinary interactive code block, when using `noInline` React Live won't wrap your code in an inline function to render it.

You will need to explicitly call `render()` at the end of your code to display the output.

The screenshot shows the React Live interface. At the top, there's a browser-like header with three colored dots (red, yellow, green) and the URL `http://localhost:3000`. Below this is a dark grey bar labeled "LIVE EDITOR". The main area contains the following code:

```
const project = "Docusaurus";

const Greeting = () => (
  <p>Hello {project}!</p>
);

render(
  <Greeting />
);
```

Below the editor is a light grey bar labeled "RESULT". Underneath it, the text "Hello Docusaurus!" is displayed, indicating the rendered output of the code.

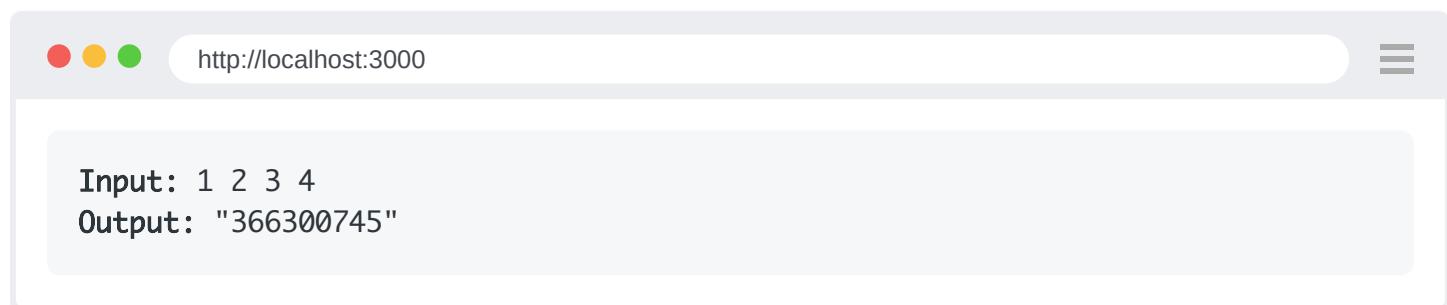
Using JSX markup in code blocks

Code block in Markdown always preserves its content as plain text, meaning you can't do something like:

```
type EditUrlFunction = (params: {  
  // This doesn't turn into a link (for good reason!)  
  version: <a href="/docs/versioning">Version</a>;  
  versionDocsDirPath: string;  
  docPath: string;  
  permalink: string;  
  locale: string;  
}) => string | undefined;
```

If you want to embed HTML markup such as anchor links or bold type, you can use the `<pre>` tag, `<code>` tag, or `<CodeBlock>` component.

```
<pre>  
  <b>Input: </b>1 2 3 4{'\n'}  
  <b>Output: </b>"366300745"{'\n'}  
</pre>
```



⚠️ MDX IS WHITESPACE INSENSITIVE

MDX is in line with JSX behavior: line break characters, even when inside `<pre>`, are turned into spaces. You have to explicitly write the new line character for it to be printed out.

⚠️ WARNING

Syntax highlighting only works on plain strings. Docusaurus will not attempt to parse code block content containing JSX children.

Multi-language support code blocks

With MDX, you can easily create interactive components within your documentation, for example, to display code in multiple programming languages and switch between them using a tabs component.

Instead of implementing a dedicated component for multi-language support code blocks, we've implemented a general-purpose `<Tabs>` component in the classic theme so that you can use it for other non-code scenarios as well.

The following example is how you can have multi-language code tabs in your docs. Note that the empty lines above and below each language block are **intentional**. This is a [current limitation of MDX](#): you have to leave empty lines around Markdown syntax for the MDX parser to know that it's Markdown syntax and not JSX.

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

<Tabs>
<TabItem value="js" label="JavaScript">

```js
function helloWorld() {
 console.log('Hello, world!');
}
```

</TabItem>
<TabItem value="py" label="Python">

```py
def hello_world():
 print("Hello, world!")
```

</TabItem>
<TabItem value="java" label="Java">

```java
class HelloWorld {
 public static void main(String args[]) {
 System.out.println("Hello, World");
 }
}
```

</TabItem>
</Tabs>
```

And you will get the following:



JavaScript Python Java

```
function helloWorld() {
  console.log('Hello, world!');
}
```

If you have multiple of these multi-language code tabs, and you want to sync the selection across the tab instances, refer to the [Syncing tab choices section](#).

Docusaurus npm2yarn remark plugin

Displaying CLI commands in both npm and Yarn is a very common need, for example:

npm Yarn pnpm Bun

```
npm install @docusaurus/remark-plugin-npm2yarn
```

Docusaurus provides such a utility out of the box, freeing you from using the `Tabs` component every time. To enable this feature, first install the `@docusaurus/remark-plugin-npm2yarn` package as above, and then in `docusaurus.config.js`, for the plugins where you need this feature (doc, blog, pages, etc.), register it in the `remarkPlugins` option. (See [Docs configuration](#) for more details on configuration format)

`docusaurus.config.js`

```
export default {
  // ...
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          remarkPlugins: [
            [require('@docusaurus/remark-plugin-npm2yarn'), {sync: true}],
          ],
        },
        pages: {
          remarkPlugins: [
            [require('@docusaurus/remark-plugin-npm2yarn'), {sync: true}],
          ],
        },
      },
    ],
  ],
}
```

```

    remarkPlugins: [require('@docusaurus/remark-plugin-npm2yarn')],  

  },  

  blog: {  

    remarkPlugins: [  

      [  

        require('@docusaurus/remark-plugin-npm2yarn'),  

        {converters: ['pnpm']},  

        ],  

        ],  

        // ...  

      },  

      ],  

      ],  

    };  

}

```

And then use it by adding the `npm2yarn` key to the code block:

```

```bash npm2yarn
npm install @docusaurus/remark-plugin-npm2yarn
```

```

Configuration

| Option | Type | Default | Description |
|-------------------------|----------------------|--|---|
| <code>sync</code> | <code>boolean</code> | <code>false</code> | Whether to sync the selected converter across all code blocks. |
| <code>converters</code> | <code>array</code> | <code>'yarn'</code> ,
<code>'pnpm'</code> | The list of converters to use. The order of the converters is important, as the first converter will be used as the default choice. |

Usage in JSX

Outside of Markdown, you can use the `@theme/CodeBlock` component to get the same output.

```

import CodeBlock from '@theme/CodeBlock';

export default function MyReactPage() {
  return (
    <div>

```

```
<CodeBlock
  language="jsx"
  title="/src/components/HelloCodeTitle.js"
  showLineNumbers>
  {`function HelloCodeTitle(props) {
  return <h1>Hello, {props.name}</h1>;
}`}
</CodeBlock>
</div>
);
}
```



The props accepted are `language`, `title` and `showLineNumbers`, in the same way as you write Markdown code blocks.

Although discouraged, you can also pass in a `metastring` prop like `metastring='{1-2} title="/src/components/HelloCodeTitle.js" showLineNumbers'`, which is how Markdown code blocks are handled under the hood. However, we recommend you [use comments for highlighting lines](#).

As [previously stated](#), syntax highlighting is only applied when the children is a simple string.

Admonitions

In addition to the basic Markdown syntax, we have a special admonitions syntax by wrapping text with a set of 3 colons, followed by a label denoting its type.

Example:

```
:::note
```

Some **content** with _Markdown_ `syntax`. Check [\[this `api`\]\(#\)](#).

```
:::
```

```
:::tip
```

Some **content** with _Markdown_ `syntax`. Check [\[this `api`\]\(#\)](#).

```
:::
```

```
:::info
```

Some **content** with _Markdown_ `syntax`. Check [\[this `api`\]\(#\)](#).

```
:::
```

```
:::warning
```

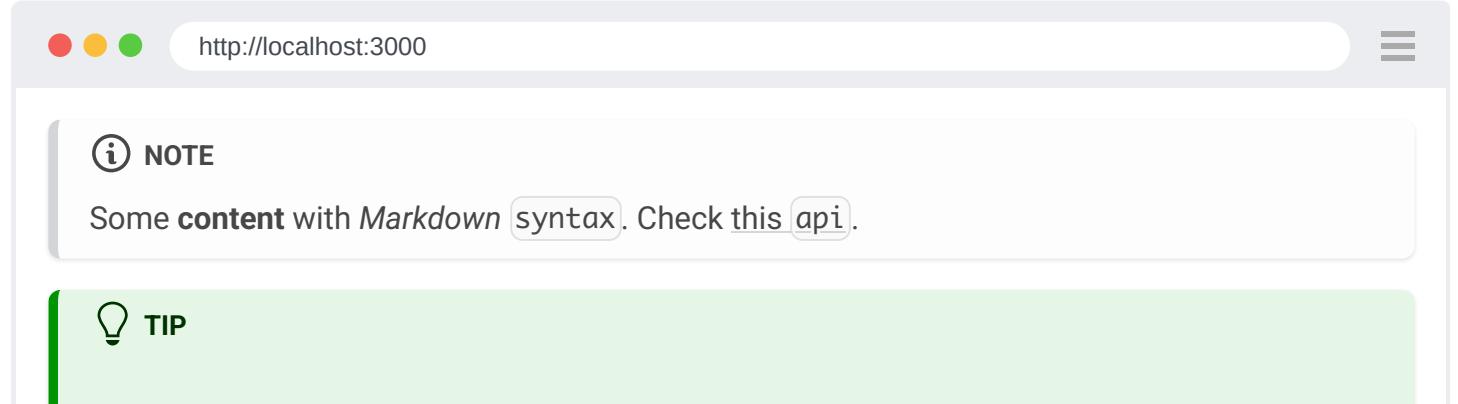
Some **content** with _Markdown_ `syntax`. Check [\[this `api`\]\(#\)](#).

```
:::
```

```
:::danger
```

Some **content** with _Markdown_ `syntax`. Check [\[this `api`\]\(#\)](#).

```
:::
```



A screenshot of a web browser window. The address bar shows "http://localhost:3000". The main content area contains a "NOTE" box with an info icon. Inside the box, the text "Some content with Markdown syntax. Check this api." is displayed. Below this is a "TIP" box with a lightbulb icon.

 NOTE

Some **content** with *Markdown* `syntax`. Check [this `api`](#).

 TIP

Some **content** with *Markdown* [syntax](#). Check this [api](#).

ⓘ INFO

Some **content** with *Markdown* [syntax](#). Check this [api](#).

⚠ WARNING

Some **content** with *Markdown* [syntax](#). Check this [api](#).

🔥 DANGER

Some **content** with *Markdown* [syntax](#). Check this [api](#).

Usage with Prettier

If you use [Prettier](#) to format your Markdown files, Prettier might auto-format your code to invalid admonition syntax. To avoid this problem, add empty lines around the starting and ending directives. This is also why the examples we show here all have empty lines around the content.

```
<!-- Prettier doesn't change this -->
:::note
```

```
Hello world
```

```
:::
```

```
<!-- Prettier changes this -->
```

```
:::note
Hello world
:::
```

```
<!-- to this -->
```

```
::: note Hello world:::
```

Specifying title

You may also specify an optional title.

```
:::note[Your Title **with** some _Markdown_ `syntax`!]
```

```
Some **content** with some _Markdown_ `syntax`.
```

:::



Nested admonitions

Admonitions can be nested. Use more colons :: for each parent admonition level.

```
::::::info[Parent]
```

Parent content

```
::::::danger[Child]
```

Child content

```
:::::tip[Deep Child]
```

Deep child content

```
:::
```

```
:::::
```

```
::::::
```

```
http://localhost:3000
```

PARENT

Parent content

CHILD

Child content

DEEP CHILD

Admonitions with MDX

You can use MDX inside admonitions too!

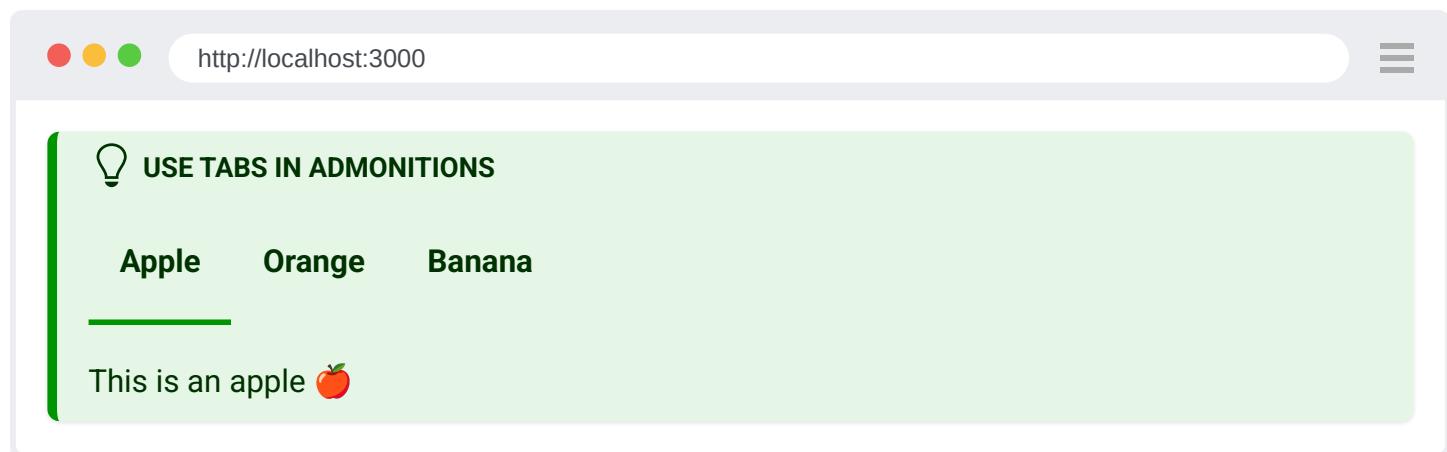
```
import Tabs from '@theme/Tabs';

import TabItem from '@theme/TabItem';

:::tip[Use tabs in admonitions]

<Tabs>
  <TabItem value="apple" label="Apple">This is an apple 🍎</TabItem>
  <TabItem value="orange" label="Orange">This is an orange 🍊</TabItem>
  <TabItem value="banana" label="Banana">This is a banana 🍌</TabItem>
</Tabs>

:::
```



Usage in JSX

Outside of Markdown, you can use the `@theme/Admonition` component to get the same output.

MyReactPage.jsx

```
import Admonition from '@theme/Admonition';

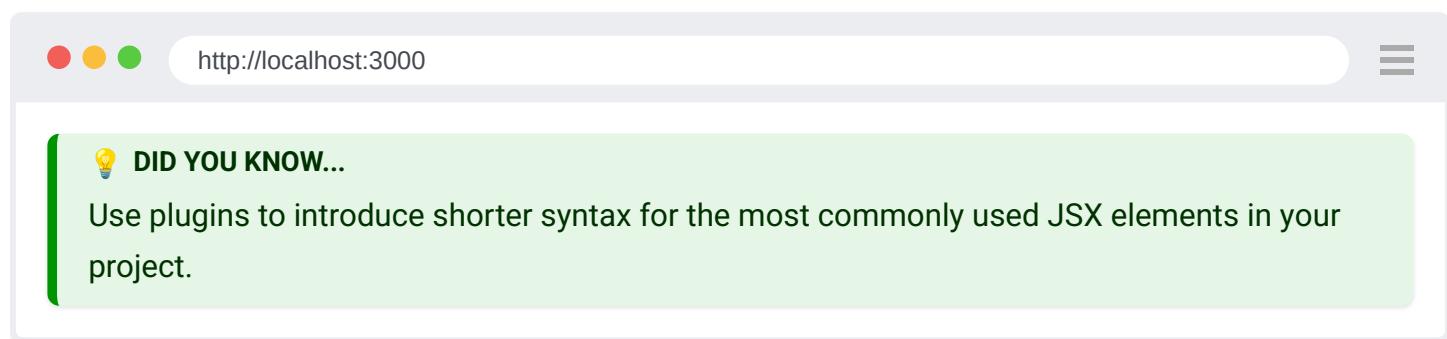
export default function MyReactPage() {
  return (
    <Admonition>
```

```
<div>
  <Admonition type="info">
    <p>Some information</p>
  </Admonition>
</div>
);
}
```

The types that are accepted are the same as above: `note`, `tip`, `danger`, `info`, `warning`. Optionally, you can specify an icon by passing a JSX element or a string, or a title:

MyReactPage.jsx

```
<Admonition type="tip" icon="💡" title="Did you know...">
  Use plugins to introduce shorter syntax for the most commonly used JSX
  elements in your project.
</Admonition>
```



Customizing admonitions

There are two kinds of customizations possible with admonitions: `parsing` and `rendering`.

Customizing rendering behavior

You can customize how each individual admonition type is rendered through `swizzling`. You can often achieve your goal through a simple wrapper. For example, in the follow example, we swap out the icon for `info` admonitions only.

src/theme/Admonition.js

```
import React from 'react';
import Admonition from '@theme-original/Admonition';
import MyCustomNoteIcon from '@site/static/img/info.svg';
```

```
export default function AdmonitionWrapper(props) {
  if (props.type !== 'info') {
    return <Admonition title="My Custom Admonition Title" {...props} />;
  }
  return <Admonition icon={<MyCustomNoteIcon />} {...props} />;
}
```

Customizing parsing behavior

Admonitions are implemented with a [Remark plugin](#). The plugin is designed to be configurable. To customize the Remark plugin for a specific content plugin (docs, blog, pages), pass the options through the `admonitions` key.

`docusaurus.config.js`

```
export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          admonitions: {
            keywords: ['note', 'tip', 'info', 'warning', 'danger'],
            extendDefaults: true,
          },
        },
      },
    ],
  ],
};
```

The plugin accepts the following options:

- `keywords`: An array of keywords that can be used as the type for the admonition.
- `extendDefaults`: Should the provided options (such as `keywords`) be merged into the existing defaults. Defaults to `true`.

The `keyword` will be passed as the `type` prop of the `Admonition` component.

Custom admonition type components

By default, the theme doesn't know what do to with custom admonition keywords such as `:::my-custom-admonition`. It is your responsibility to map each admonition keyword to a React component so that the theme knows how to render them.

If you registered a new admonition type `my-custom-admonition` via the following config:

docusaurus.config.js

```
export default {
  // ...
  presets: [
    [
      'classic',
      {
        // ...
        docs: {
          admonitions: {
            keywords: ['my-custom-admonition'],
            extendDefaults: true,
          },
        },
      },
    ],
  ],
};
```

You can provide the corresponding React component for `:::my-custom-admonition` by creating the following file (unfortunately, since it's not a React component file, it's not swizzlable):

src/theme/Admonition/Types.js

```
import React from 'react';
import DefaultAdmonitionTypes from '@theme-original/Admonition/Types';

function MyCustomAdmonition(props) {
  return (
    <div style={{border: 'solid red', padding: 10}}>
      <h5 style={{color: 'blue', fontSize: 30}}>{props.title}</h5>
      <div>{props.children}</div>
    </div>
  );
}

const AdmonitionTypes = {
  ...DefaultAdmonitionTypes,
  // Add all your custom admonition types here...
  // You can also override the default ones if you want
  'my-custom-admonition': MyCustomAdmonition,
};
```

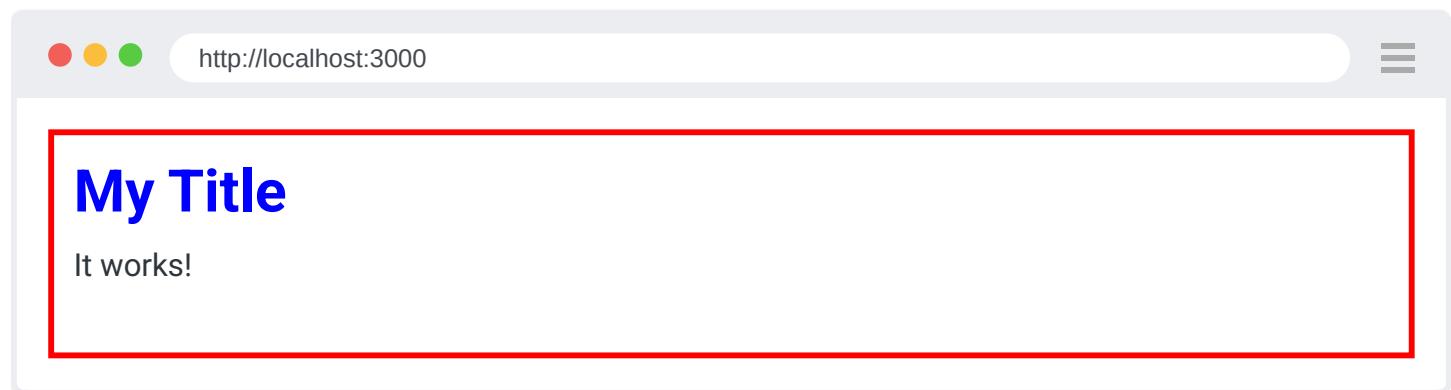
```
export default AdmonitionTypes;
```

Now you can use your new admonition keyword in a Markdown file, and it will be parsed and rendered with your custom logic:

```
:::my-custom-admonition[My Title]
```

It works!

```
:::
```



Headings and Table of contents

Markdown headings

You can use regular Markdown headings.

```
## Level 2 title  
### Level 3 title  
#### Level 4 title
```

Each Markdown heading will appear as a table of contents entry.

Heading IDs

Each heading has an ID that can be automatically generated or explicitly specified. Heading IDs allow you to link to a specific document heading in Markdown or JSX:

```
[link](#heading-id)
```

```
<Link to="#heading-id">link</Link>
```

By default, Docusaurus will generate heading IDs for you, based on the heading text. For example, `### Hello World` will have ID `hello-world`.

Generated IDs have **some limitations**:

- The ID might not look good
- You might want to **change or translate** the text without updating the existing ID

A special Markdown syntax lets you set an **explicit heading id**:

```
### Hello World {#my-explicit-id}
```



TIP

Use the `write-heading-ids` CLI command to add explicit IDs to all your Markdown documents.

AVOID COLLIDING IDS

Generated heading IDs will be guaranteed to be unique on each page, but if you use custom IDs, make sure each one appears exactly once on each page, or there will be two DOM elements with the same ID, which is invalid HTML semantics, and will lead to one heading being unlinkable.

Table of contents heading level

Each Markdown document displays a table of contents on the top-right corner. By default, this table only shows h2 and h3 headings, which should be sufficient for an overview of the page structure. In case you need to change the range of headings displayed, you can customize the minimum and maximum heading level – either per page or globally.

To set the heading level for a particular page, use the `toc_min_heading_level` and `toc_max_heading_level` front matter.

myDoc.md

```
---
# Display h2 to h5 headings
toc_min_heading_level: 2
toc_max_heading_level: 5
---
```

To set the heading level for *all* pages, use the `themeConfig.tableOfContents` option.

docusaurus.config.js

```
export default {
  themeConfig: {
    tableOfContents: {
      minHeadingLevel: 2,
      maxHeadingLevel: 5,
    },
  },
};
```

If you've set the options globally, you can still override them locally via front matter.

NOTE

The `themeConfig` option would apply to all TOC on the site, including `inline TOC`, but front matter options only affect the top-right TOC. You need to use the `minHeadingLevel` and `maxHeadingLevel` props to customize each `<TOCInline />` component.

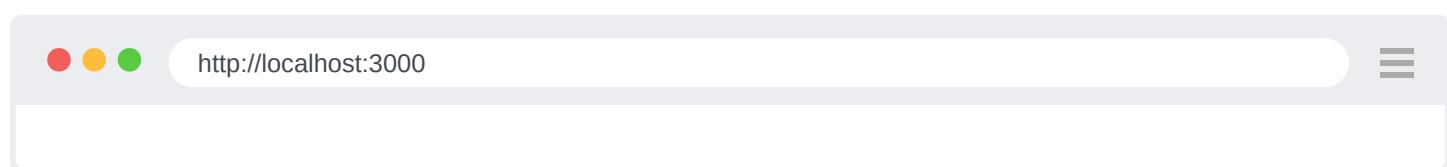
Inline table of contents

It is also possible to display an inline table of contents directly inside a Markdown document, thanks to MDX.

The `toc` variable is available in any MDX document and contains all the headings of an MDX document. By default, only `h2` and `h3` headings are displayed in the TOC. You can change which heading levels are visible by setting `minHeadingLevel` or `maxHeadingLevel` for individual `TOCInline` components.

```
import TOCInline from '@theme/TOCInline';

<TOCInline toc={toc} />
```



The `toc` global is just a list of heading items:

```
declare const toc: [
  value: string;
  id: string;
  level: number;
][];
```

Note that the `toc` global is a flat array, so you can easily cut out unwanted nodes or insert extra nodes, and create a new TOC tree.

```
import TOCInline from '@theme/TOCInline';

<TOCInline
  // Only show h2 and h4 headings
  toc={toc.filter((node) => node.level === 2 || node.level === 4)}
  minHeadingLevel={2}
  // Show h4 headings in addition to the default h2 and h3 headings
  maxHeadingLevel={4}
/>
```

Customizing table of contents generation

The table-of-contents is generated by parsing the Markdown source with a [Remark plugin](#). There are known edge-cases where it generates false-positives and false-negatives.

Markdown headings within hideable areas will still show up in the TOC. For example, headings within `Tabs` and `details` will not be excluded.

Non-Markdown headings will not show up in the TOC. This can be used to your advantage to tackle the aforementioned issue.

```
<details>
<summary>Some details containing headings</summary>
<h2 id="#heading-id">I'm a heading that will not show up in the TOC</h2>

Some content...

</details>
```

The ability to ergonomically insert extra headings or ignore certain headings is a work-in-progress. If this feature is important to you, please report your use-case in [this issue](#).

WARNING

Below is just some dummy content to have more table of contents items available on the current page.

Example Section 1

Lorem ipsum

Example Subsection 1 a

Lorem ipsum

Example subsubsection 1 a I

Example subsubsection 1 a II

Example subsubsection 1 a III

Example Subsection 1 b

 Lorem ipsum

Example subsubsection 1 b I

Example subsubsection 1 b II

Example subsubsection 1 b III

Example Subsection 1 c

 Lorem ipsum

Example subsubsection 1 c I

Example subsubsection 1 c II

Example subsubsection 1 c III

Example Section 2

 Lorem ipsum

Example Subsection 2 a

 Lorem ipsum

Example subsubsection 2 a I

Example subsubsection 2 a II

Example subsubsection 2 a III

Example Subsection 2 b

 Lorem ipsum

Example subsubsection 2 b I

Example subsubsection 2 b II

Example subsubsection 2 b III

Example Subsection 2 c

 Lorem ipsum

Example subsubsection 2 c I

Example subsubsection 2 c II

Example subsubsection 2 c III

Example Section 3

 Lorem ipsum

Example Subsection 3 a

 Lorem ipsum

Example subsubsection 3 a I

Example subsubsection 3 a II

Example subsubsection 3 a III

Example Subsection 3 b

 Lorem ipsum

Example subsubsection 3 b I

Example subsubsection 3 b II

Example subsubsection 3 b III

Example Subsection 3 c

 Lorem ipsum

Example subsubsection 3 c I

Example subsubsection 3 c II

Example subsubsection 3 c III

Assets

Sometimes you want to link to assets (e.g. docx files, images...) directly from Markdown files, and it is convenient to co-locate the asset next to the Markdown file using it.

Let's imagine the following file structure:

```
# Your doc  
/website/docs/myFeature.mdx  
  
# Some assets you want to use  
/website/docs/assets/docusaurus-asset-example-banner.png  
/website/docs/assets/docusaurus-asset-example.docx
```

Images

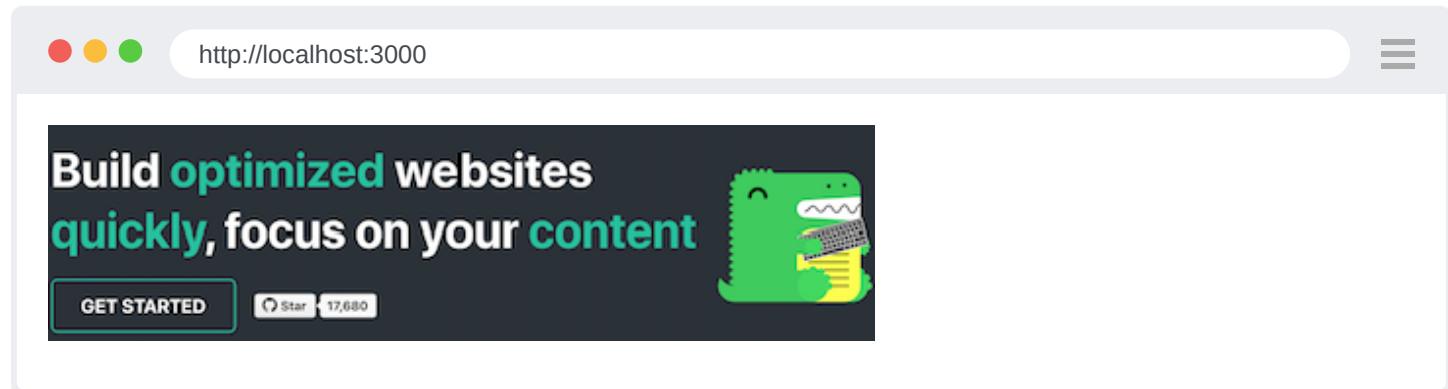
You can display images in three different ways: Markdown syntax, CJS require, or ES imports syntax.

[Markdown syntax](#) [CommonJS require](#) [Import statement](#)

Display images using simple Markdown syntax:

```
![Example banner](./assets/docusaurus-asset-example-banner.png)
```

All of the above result in displaying the image:



NOTE

If you are using `@docusaurus/plugin-ideal-image`, you need to use the dedicated image component, as documented.

Files

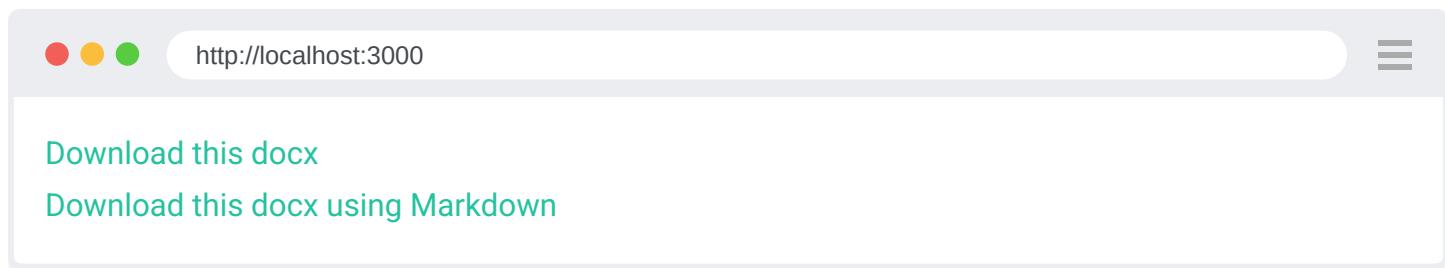
In the same way, you can link to existing assets by `require`'ing them and using the returned URL in `videos`, `a` anchor links, etc.

My Markdown page

```
<a target="_blank" href={require('./assets/docusaurus-asset-example.docx').default}> Download this docx </a>
```

or

```
[Download this docx using Markdown](./assets/docusaurus-asset-example.docx)
```



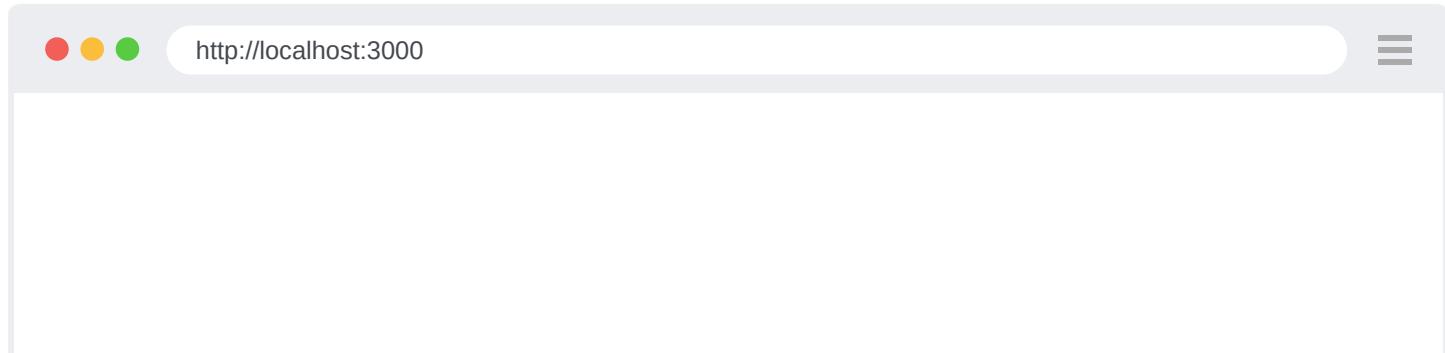
! MARKDOWN LINKS ARE ALWAYS FILE PATHS

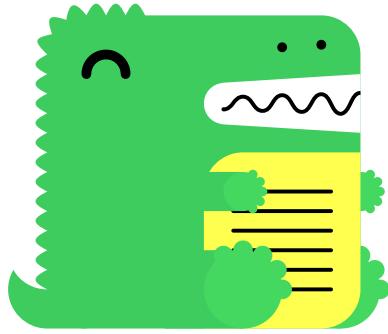
If you use the Markdown image or link syntax, all asset paths will be resolved as file paths by Docusaurus and automatically converted to `require()` calls. You don't need to use `require()` in Markdown unless you use the JSX syntax, which you do have to handle yourself.

Inline SVGs

Docusaurus supports inlining SVGs out of the box.

```
import DocusaurusSvg from './docusaurus.svg';  
  
<DocusaurusSvg />;
```

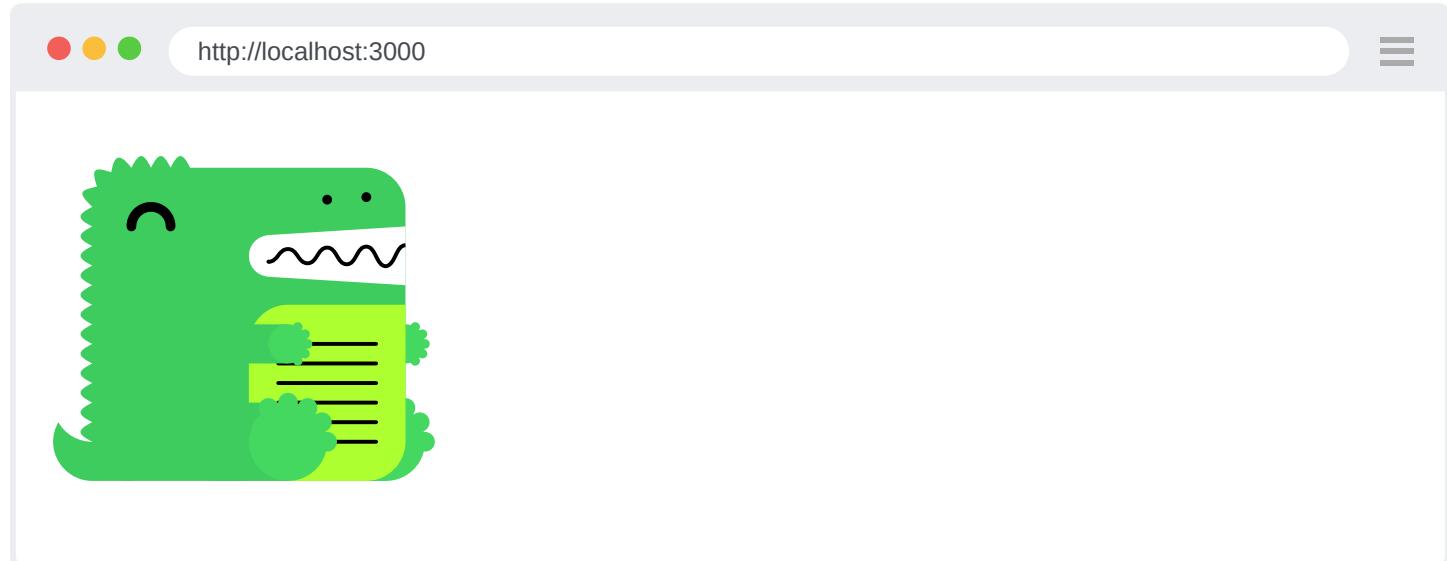




This can be useful if you want to alter the part of the SVG image via CSS. For example, you can change one of the SVG colors based on the current theme.

```
import DocusaurusSvg from './docusaurus.svg';  
  
<DocusaurusSvg className="themedDocusaurus" />;
```

```
[data-theme='light'] .themedDocusaurus [fill='#FFFF50'] {  
  fill: greenyellow;  
}  
  
[data-theme='dark'] .themedDocusaurus [fill='#FFFF50'] {  
  fill: seagreen;  
}
```

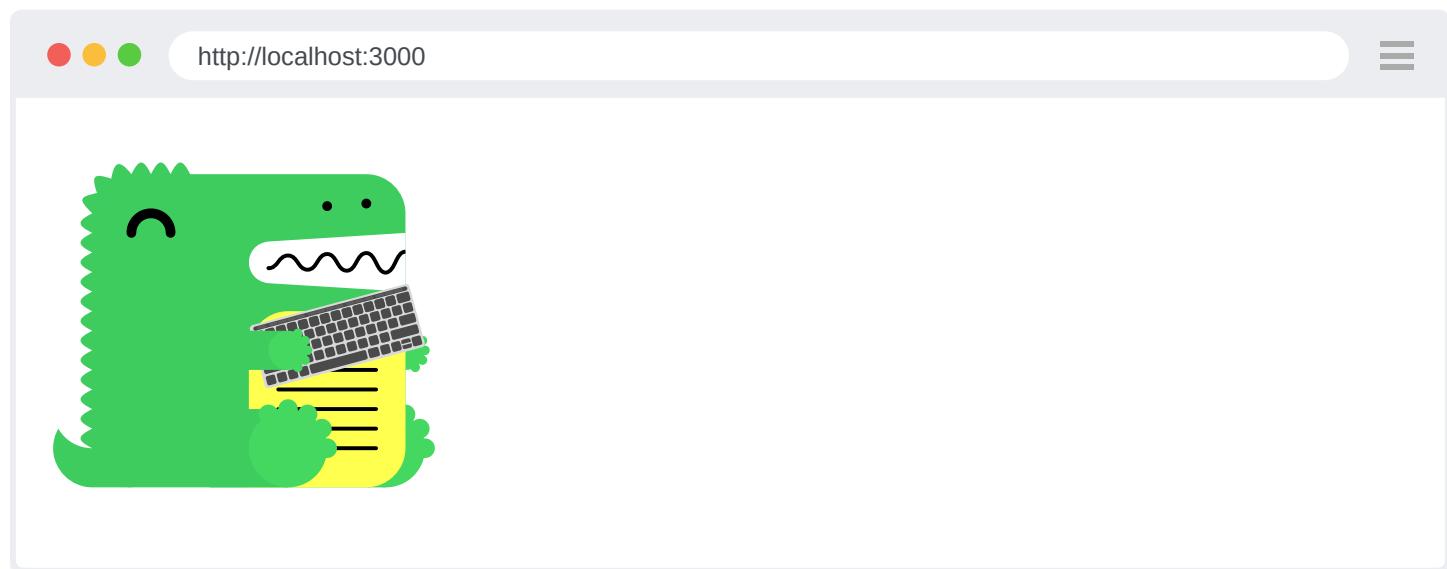


Themed Images

Docusaurus supports themed images: the `ThemedImage` component (included in the themes) allows you to switch the image source based on the current theme.

```
import useBaseUrl from '@docusaurus/useBaseUrl';
import ThemedImage from '@theme/ThemedImage';

<ThemedImage
  alt="Docusaurus themed image"
  sources={{
    light: useBaseUrl('/img/docusaurus_light.svg'),
    dark: useBaseUrl('/img/docusaurus_dark.svg'),
  }}
/>;
```



GitHub-style themed images

GitHub uses its own [image theming approach](#) with path fragments, which you can easily implement yourself.

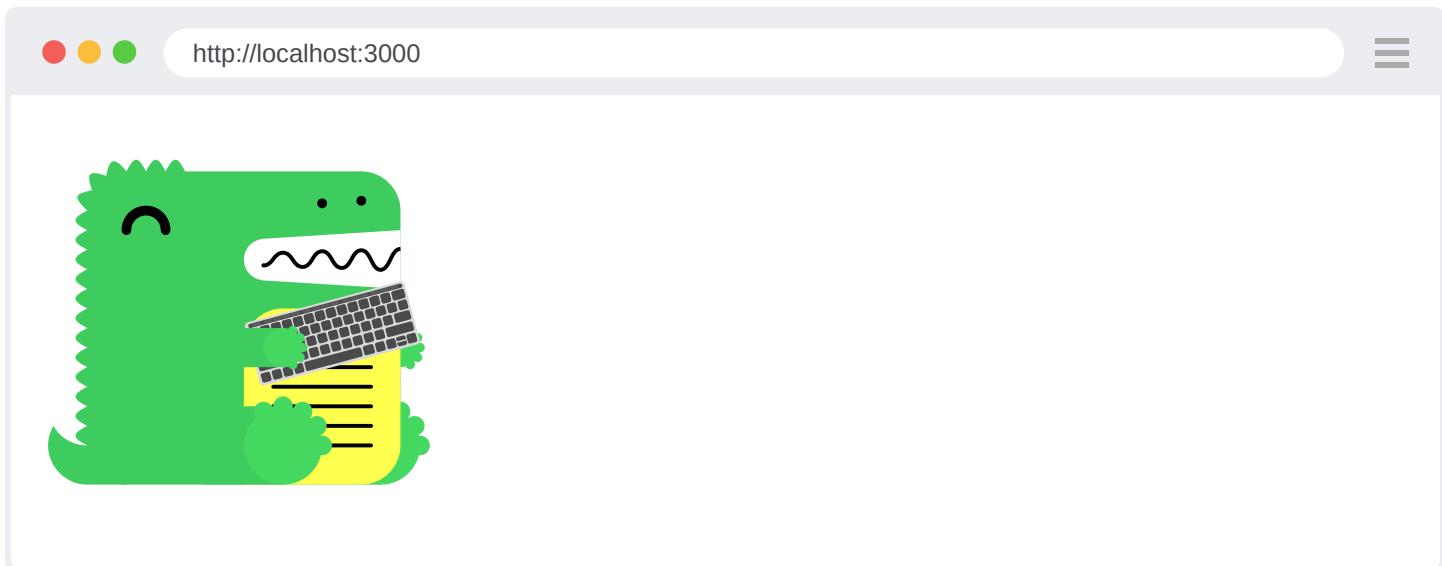
To toggle the visibility of an image using the path fragment (for GitHub, it's `#gh-dark-mode-only` and `#gh-light-mode-only`), add the following to your custom CSS (you can also use your own suffix if you don't want to be coupled to GitHub):

```
src/css/custom.css
```

```
[data-theme='light'] img[src$('#gh-dark-mode-only')],
[data-theme='dark'] img[src$('#gh-light-mode-only')] {
  display: none;
}
```

! [Docusaurus themed image](/img/docusaurus_keytar.svg#gh-light-mode-only)!

[Docusaurus themed image](/img/docusaurus_speed.svg#gh-dark-mode-only)



Static assets

If a Markdown link or image has an absolute path, the path will be seen as a file path and will be resolved from the static directories. For example, if you have configured `static directories` to be `['public', 'static']`, then for the following image:

my-doc.md

! [An image from the static](/img/docusaurus.png)

Docusaurus will try to look for it in both `static/img/docusaurus.png` and `public/img/docusaurus.png`. The link will then be converted to a `require()` call instead of staying as a URL. This is desirable in two regards:

1. You don't have to worry about the base URL, which Docusaurus will take care of when serving the asset;
2. The image enters Webpack's build pipeline and its name will be appended by a hash, which enables browsers to aggressively cache the image and improves your site's performance.

If you intend to write URLs, you can use the `pathname://` protocol to disable automatic asset linking.

! [banner](pathname:///img/docusaurus-asset-example-banner.png)

This link will be generated as ``, without any processing or file existence checking.

Markdown links

There are two ways of adding a link to another page: through a **URL path** and a **file path**.

- [URL path to another document](./installation)
- [file path to another document](./installation.mdx)

URL paths are unprocessed by Docusaurus, and you can see them as directly rendering to ``, i.e. it will be resolved according to the page's URL location, rather than its file-system location.

If you want to reference another Markdown file **included by the same plugin**, you could use the relative path of the document you want to link to. Docusaurus' Markdown loader will convert the file path to the target file's URL path (and hence remove the `.md` extension).

For example, if you are in `docs/folder/doc1.md` and you want to reference `docs/folder/doc2.md`, `docs/folder/subfolder/doc3.md` and `docs/otherFolder/doc4.md`:

`docs/folder/doc1.md`

I am referencing a [document](doc2.mdx).

Reference to another [document in a subfolder](subfolder/doc3.mdx).

[Relative document](../otherFolder/doc4.mdx) referencing works as well.

Relative file paths are resolved against the current file's directory. Absolute file paths, on the other hand, are resolved relative to the **content root**, usually `docs/`, `blog/`, or **localized ones** like `i18n/zh-Hans/plugin-content-docs/current`.

Absolute file paths can also be relative to the site directory. However, beware that links that begin with `/docs/` or `/blog/` are **not portable** as you would need to manually update them if you create new doc versions or localize them.

You can write [links](/otherFolder/doc4.mdx) relative to the content root (`^/docs/^`).

You can also write [links](/docs/otherFolder/doc4.mdx) relative to the site directory, but it's not recommended.

Using relative *file* paths (with `.md` extensions) instead of relative *URL* links provides the following benefits:

- Links will keep working on the GitHub interface and many Markdown editors
- You can customize the files' slugs without having to update all the links
- Moving files around the folders can be tracked by your editor, and some editors may automatically update file links
- A [versioned doc](#) will link to another doc of the exact same version
- Relative URL links are very likely to break if you update the [trailingSlash config](#)

 **WARNING**

Markdown file references only work when the source and target files are processed by the same plugin instance. This is a technical limitation of our Markdown processing architecture and will be fixed in the future. If you are linking files between plugins (e.g. linking to a doc page from a blog post), you have to use URL links.

MDX Plugins

Sometimes, you may want to extend or tweak your Markdown syntax. For example:

- How do I embed youtube videos using the image syntax (`![]` (`https://youtu.be/yKNxeF4KMsY`))?
- How do I style links that are on their own lines differently, e.g., as a social card?
- How do I make every page start with a copyright notice?

And the answer is: create an MDX plugin! MDX has a built-in [plugin system](#) that can be used to customize how the Markdown files will be parsed and transformed to JSX. There are three typical use-cases of MDX plugins:

- Using existing [remark plugins](#) or [rehype plugins](#);
- Creating remark/rehype plugins to transform the elements generated by existing MDX syntax;
- Creating remark/rehype plugins to introduce new syntaxes to MDX.

If you play with the [MDX playground](#), you would notice that the MDX transpilation has two intermediate steps: Markdown AST (MDAST), and Hypertext AST (HAST), before arriving at the final JSX output. MDX plugins also come in two forms:

- **Remark**: processes the Markdown AST.
- **Rehype**: processes the Hypertext AST.



Use plugins to introduce shorter syntax for the most commonly used JSX elements in your project. The [admonition](#) syntax that we offer is also generated by a Remark plugin, and you could do the same for your own use case.

Default plugins

Docusaurus injects [some default Remark plugins](#) during Markdown processing. These plugins would:

- Generate the table of contents;
- Add anchor links to each heading;
- Transform images and links to `require()` calls.
- ...

These are all typical use-cases of Remark plugins, which can also be a source of inspiration if you want to implement your own plugin.

Installing plugins

An MDX plugin is usually an npm package, so you install them like other npm packages using npm. Take the [math plugins](#) as an example.

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install --save remark-math@5 rehype-katex@6
```

▼ How are `remark-math` and `rehype-katex` different?

⚠ WARNING

Many official Remark/Rehype plugins are [ES Modules only](#), a JavaScript module system, which Docusaurus supports. We recommend using an [ES Modules](#) config file to make it easier to import such packages.

Next, import your plugins and add them to the plugin options through plugin or preset config in `docusaurus.config.js`:

`docusaurus.config.js`

```
import remarkMath from 'remark-math';
import rehypeKatex from 'rehype-katex';

export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          path: 'docs',
          remarkPlugins: [remarkMath],
          rehypePlugins: [rehypeKatex],
        },
      },
    ],
  ],
}
```

```
],  
};
```

▼ Using a [CommonJS](#) config file?

Configuring plugins

Some plugins can be configured and accept their own options. In that case, use the `[plugin, pluginOptions]` syntax, like this:

`docusaurus.config.js`

```
import rehypeKatex from 'rehype-katex';

export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          rehypePlugins: [
            [rehypeKatex, {strict: false}],
            ],
          },
        },
      ],
    ],
  };
};
```

You should check your plugin's documentation for the options it supports.

Creating new rehype/remark plugins

If there isn't an existing package that satisfies your customization need, you can create your own MDX plugin.

NOTE

The writeup below is **not** meant to be a comprehensive guide to creating a plugin, but just an illustration of how to make it work with Docusaurus. Visit the [Remark](#) or [Rehype](#) documentation for a more in-depth explanation of how they work.

For example, let's make a plugin that visits every `h2` heading and adds a `Section X.` prefix. First, create your plugin source file anywhere—you can even publish it as a separate npm package and install it like explained above. We would put ours at `src/remark/section-prefix.js`. A remark/rehype plugin is just a function that receives the `options` and returns a `transformer` that operates on the AST.

```
import {visit} from 'unist-util-visit';

const plugin = (options) => {
  const transformer = async (ast) => {
    let number = 1;
    visit(ast, 'heading', (node) => {
      if (node.depth === 2 && node.children.length > 0) {
        node.children.unshift({
          type: 'text',
          value: `Section ${number}. `,
        });
        number++;
      }
    });
  };
  return transformer;
};

export default plugin;
```

You can now import your plugin in `docusaurus.config.js` and use it just like an installed plugin!

`docusaurus.config.js`

```
import sectionPrefix from './src/remark/section-prefix';

export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          remarkPlugins: [sectionPrefix],
        },
      },
    ],
  ],
};
```

TIP

The `transformer` has a second parameter `vfile` which is useful if you need to access the current Markdown file's path.

```
const plugin = (options) => {
  const transformer = async (ast, vfile) => {
    ast.children.unshift({
      type: 'text',
      value: `The current file path is ${vfile.path}`,
    });
  };
  return transformer;
};
```

Our `transformImage` plugin uses this parameter, for example, to transform relative image references to `require()` calls.

NOTE

The default plugins of Docusaurus would operate before the custom remark plugins, and that means the images or links have been converted to JSX with `require()` calls already. For example, in the example above, the table of contents generated is still the same even when all `h2` headings are now prefixed by `Section X.`, because the TOC-generating plugin is called before our custom plugin. If you need to process the MDAST before the default plugins do, use the `beforeDefaultRemarkPlugins` and `beforeDefaultRehypePlugins`.

docusaurus.config.js

```
export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          beforeDefaultRemarkPlugins: [sectionPrefix],
        },
      },
    ],
  ],
};
```

This would make the table of contents generated contain the `Section X.` prefix as well.

Math Equations

Mathematical equations can be rendered using [KaTeX](#).

See [below](#) how to activate them.

Usage

Please read the [KaTeX](#) documentation for more details.

Inline

Write inline math equations by wrapping LaTeX equations between `$`:

```
Let  $f: [a, b] \rightarrow \mathbb{R}$  be Riemann integrable. Let  $F: [a, b] \rightarrow \mathbb{R}$  be  $F(x) = \int_a^x f(t) dt$ . Then  $F$  is continuous, and at all  $x$  such that  $f$  is continuous at  $x$ ,  $F$  is differentiable at  $x$  with  $F'(x) = f(x)$ .
```

A screenshot of a web browser window. The address bar shows "http://localhost:3000". The main content area displays the text: "Let $f: [a, b] \rightarrow \mathbb{R}$ be Riemann integrable. Let $F: [a, b] \rightarrow \mathbb{R}$ be $F(x) = \int_a^x f(t) dt$. Then F is continuous, and at all x such that f is continuous at x , F is differentiable at x with $F'(x) = f(x)$ ".

Blocks

For equation block or display mode, use line breaks and `$$`:

```
$$
I = \int_0^{2\pi} \sin(x) dx
$$
```

A screenshot of a web browser window. The address bar shows "http://localhost:3000". The main content area displays the equation:
$$I = \int_0^{2\pi} \sin(x) dx$$

Enabling math equations

Enable KaTeX:

1. Install the `remark-math` and `rehype-katex` plugins:

[npm](#) [Yarn](#) [pnpm](#) [Bun](#)

```
npm install --save remark-math@6 rehype-katex@7
```



WARNING

Make sure to use `remark-math` 6 and `rehype-katex` 7 for Docusaurus v3 (using MDX v3). We can't guarantee other versions will work.

2. These 2 plugins are **only available as ES Modules**. We recommended to use an **ES Modules** config file:

ES module `docusaurus.config.js`

```
import remarkMath from 'remark-math';
import rehypeKatex from 'rehype-katex';

export default {
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          path: 'docs',
          remarkPlugins: [remarkMath],
          rehypePlugins: [rehypeKatex],
        },
      },
    ],
  ],
};
```

▼ Using a [CommonJS](#) config file?

3. Include the KaTeX CSS in your config under `stylesheets`:

```
export default {
  //...
  stylesheets: [
    {
      href: 'https://cdn.jsdelivr.net/npm/katex@0.13.24/dist/katex.min.css',
      type: 'text/css',
      integrity:
        'sha384-  
odtC+0UGzzFL/6PNoE8rX/SPcQDXBJ+uRepguP4QkPCm2LBxH3FA3y+fKSiJ+AmM',
      crossorigin: 'anonymous',
    },
  ],
};
```

▼ See a config file example

Self-hosting KaTeX assets

Loading stylesheets, fonts, and JavaScript libraries from CDN sources is a good practice for popular libraries and assets, since it reduces the amount of assets you have to host. In case you prefer to self-host the `katex.min.css` (along with required KaTeX fonts), you can download the latest version from [KaTeX GitHub releases](#), extract and copy `katex.min.css` and `fonts` directory (only `.woff2` font types should be enough) to your site's `static` directory, and in `docusaurus.config.js`, replace the stylesheet's `href` from the CDN URL to your local path (say, `/katex/katex.min.css`).

`docusaurus.config.js`

```
export default {
  stylesheets: [
    {
      href: '/katex/katex.min.css',
      type: 'text/css',
    },
  ],
};
```

Diagrams

Diagrams can be rendered using **Mermaid** in a code block.

Installation

npm **Yarn** **pnpm** **Bun**

```
npm install --save @docusaurus/theme-mermaid
```

Enable Mermaid functionality by adding plugin `@docusaurus/theme-mermaid` and setting `markdown.mermaid` to `true` in your `docusaurus.config.js`.

docsaurus.config.js

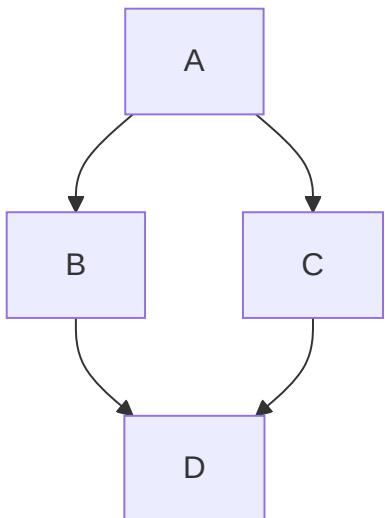
```
export default {
  markdown: {
    mermaid: true,
  },
  themes: ['@docusaurus/theme-mermaid'],
};
```

Usage

Add a code block with language `mermaid`:

Example Mermaid diagram

```
```mermaid\ngraph TD;\n    A-->B;\n    A-->C;\n    B-->D;\n    C-->D;\n```
```



See the [Mermaid syntax documentation](#) for more information on the Mermaid syntax.

## Theming

The diagram dark and light themes can be changed by setting `mermaid.theme` values in the `themeConfig` in your `docusaurus.config.js`. You can set themes for both light and dark mode.

### `docusaurus.config.js`

```

export default {
 themeConfig: {
 mermaid: {
 theme: {light: 'neutral', dark: 'forest'},
 },
 },
};

```

See the [Mermaid theme documentation](#) for more information on theming Mermaid diagrams.

## Mermaid Config

Options in `mermaid.options` will be passed directly to `mermaid.initialize`:

### `docusaurus.config.js`

```

export default {
 themeConfig: {
 mermaid: {
 options: {

```

```
 maxTextSize: 50,
 },
 },
 },
};
```

See the [Mermaid config documentation](#) and the [Mermaid config types](#) for the available config options.

## Dynamic Mermaid Component

To generate dynamic diagrams, you can use the `Mermaid` component:

### Example of dynamic Mermaid component

```
import Mermaid from '@theme/Mermaid';

<Mermaid
 value={`graph TD;
 A-->B;
 A-->C;
 B-->D;
 C-->D;`}
/>
```

# Head metadata

## Customizing head metadata

Docusaurus automatically sets useful page metadata in `<html>`, `<head>` and `<body>` for you. It is possible to add extra metadata (or override existing ones) with the `<head>` tag in Markdown files:

```
markdown-features-head-metadata.mdx
```

```

```

```
id: head-metadata
title: Head Metadata

```

```
<head>
 <html className="some-extra-html-class" />
 <body className="other-extra-body-class" />
 <title>Head Metadata customized title!</title>
 <meta charset="utf-8" />
 <meta name="twitter:card" content="summary" />
 <link rel="canonical" href="https://docusaurus.io/docs/markdown-features/head-metadata" />
</head>
```

```
Head Metadata
```

```
My text
```

This `<head>` declaration has been added to the current Markdown doc as a demo. Open your browser DevTools and check how this page's metadata has been affected.

### NOTE

This feature is built on top of the Docusaurus `<Head>` component. Refer to [react-helmet](#) for exhaustive documentation.

### YOU DON'T NEED THIS FOR REGULAR SEO

Content plugins (e.g. docs and blog) provide front matter options like `description`, `keywords`, and `image`, which will be automatically applied to both `description` and `og:description`, while you would have to manually declare two metadata tags when using the `<head>` tag.

# Markdown page description

The Markdown pages' description metadata may be used in more places than the head metadata. For example, the docs plugin's [generated category index](#) uses the description metadata for the doc cards.

By default, the description is the first content-ful line, with some efforts to convert it to plain text. For example, the following file...

```
Title
```

```
Main content... May contain some [links](./file.mdx) or **emphasis**.
```

...will have the default description "Main content... May contain some links or emphasis". However, **it's not designed to be fully functional**. Where it fails to produce reasonable descriptions, you can explicitly provide one through front matter:

```

```

```
description: This description will override the default.
```

```
--
```

```
Title
```

```
Main content... May contain some [links](./file.mdx) or **emphasis**.
```

# Styling and Layout



TIP

This section is focused on styling through stylesheets. For more advanced customizations (DOM structure, React code...), refer to the [swizzling guide](#).

A Docusaurus site is a single-page React application. You can style it the way you style React apps.

There are a few approaches/frameworks which will work, depending on your preferences and the type of website you are trying to build. Websites that are highly interactive and behave more like web apps will benefit from more modern styling approaches that co-locate styles with the components.

Component styling can also be particularly useful when you wish to customize or swizzle a component.

## Global styles

This is the most traditional way of styling that most developers (including non-front-end developers) would be familiar with. It works fine for small websites that do not have much customization.

If you're using `@docusaurus/preset-classic`, you can create your own CSS files (e.g. `/src/css/custom.css`) and import them globally by passing them as an option of the classic theme.

`docusaurus.config.js`

```
export default {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 theme: {
 customCss: ['./src/css/custom.css'],
 },
 },
],
],
};
```

Any CSS you write within that file will be available globally and can be referenced directly using string literals.

```
/src/css/custom.css
```

```
.purple-text {
 color: rebeccapurple;
}
```

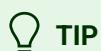
```
function MyComponent() {
 return (
 <main>
 <h1 className="purple-text">Purple Heading!</h1>
 </main>
);
}
```

If you want to add CSS to any element, you can open the DevTools in your browser to inspect its class names. Class names come in several kinds:

- **Theme class names.** These class names are listed exhaustively in [the next subsection](#). They don't have any default properties. You should always prioritize targeting those stable class names in your custom CSS.
- **Infima class names.** These class names are found in the classic theme and usually follow the [BEM convention](#) of `block__element--modifier`. They are usually stable but are still considered implementation details, so you should generally avoid targeting them. However, you can [modify Infima CSS variables](#).
- **CSS module class names.** These class names end with a hash which may change over time (`codeBlockContainer_RIuc`). They are considered implementation details and you should almost always avoid targeting them in your custom CSS. If you must, you can use an [attribute selector](#) (`[class*='codeBlockContainer']`) that ignores the hash.

## Theme Class Names

We provide some stable CSS class names for robust and maintainable global layout styling. These names are theme-agnostic and meant to be targeted by custom CSS.



If you can't find a way to create a robust CSS selector, please [report your customization use-case](#) and we will consider adding new class names.

▼ Exhaustive list of stable class names

# Styling your site with Infima

@docusaurus/preset-classic uses [Infima](#) as the underlying styling framework. Infima provides a flexible layout and common UI components styling suitable for content-centric websites (blogs, documentation, landing pages). For more details, check out the [Infima website](#).

When you scaffold your Docusaurus project with [create-docusaurus](#), the website will be generated with basic Infima stylesheets and default styling. You can override Infima CSS variables globally.

```
/src/css/custom.css
```

```
:root {
 --ifm-color-primary: #25c2a0;
 --ifm-code-font-size: 95%;
}
```

Infima uses 7 shades of each color. We recommend using [ColorBox](#) to find the different shades of colors for your chosen primary color.

Alternatively, use the following tool to generate the different shades for your website and copy the variables into [/src/css/custom.css](#).



TIP

Aim for at least [WCAG-AA contrast ratio](#) for the primary color to ensure readability. Use the Docusaurus website itself to preview how your color palette would look like. You can use alternative palettes in dark mode because one color doesn't usually work in both light and dark mode.

Primary Color:



[Edit dark mode](#)

[Reset](#)

Background:

CSS Variable Name	Hex	Adjustment	Contrast Rating
--ifm-color-primary-lightest	<input type="color" value="#3cad6e"/>	<input type="text" value="-30"/>	<span>Fail</span>
--ifm-color-primary-lighter	<input type="color" value="#359962"/>	<input type="text" value="-15"/>	<span>Fail</span>
--ifm-color-primary-light	<input type="color" value="#33925d"/>	<input type="text" value="-10"/>	<span>Fail</span>

CSS Variable Name	Hex	Adjustment	Contrast Rating
--ifm-color-primary	 #2e8555	0	AA 🌟
--ifm-color-primary-dark	 #29784c	10	AA 🌟
--ifm-color-primary-darker	 #277148	15	AA 🌟
--ifm-color-primary-darkest	 #205d3b	30	AAA 🎉

Replace the variables in `src/css/custom.css` with these new variables.

`/src/css/custom.css`

```
:root {
 --ifm-color-primary: #2e8555;
 --ifm-color-primary-dark: #29784c;
 --ifm-color-primary-darker: #277148;
 --ifm-color-primary-darkest: #205d3b;
 --ifm-color-primary-light: #33925d;
 --ifm-color-primary-lighter: #359962;
 --ifm-color-primary-lightest: #3cad6e;
}
```

## Dark Mode

In light mode, the `<html>` element has a `data-theme="light"` attribute; in dark mode, it's `data-theme="dark"`. Therefore, you can scope your CSS to dark-mode-only by targeting `html` with a specific attribute.

```
/* Overriding root Infima variables */
[data-theme='dark'] {
 --ifm-color-primary: #4e89e8;
}

/* Styling one class specially in dark mode */
[data-theme='dark'] .purple-text {
 color: plum;
}
```

 **TIP**

It is possible to initialize the Docusaurus theme directly from a `docusaurus-theme` query string parameter.

Examples:

A screenshot of a web browser window with a dark theme. The address bar shows `/docs/?docusaurus-theme=dark`. The page content is the 'Introduction' section of a Docusaurus site. It features a large title 'Introduction' and two bullet points: one with a lightning bolt icon and another with a wrench icon. Both points mention the benefits of using Docusaurus.

- ⚡ Docusaurus will help you ship a **beautiful documentation site in no time.**
- 🔧 Building a custom tech stack is expensive. Instead, **focus on your content** and just write Markdown files.

A screenshot of a web browser window with a light theme. The address bar shows `/docs/configuration?docusaurus-theme=light`. The page content is the 'Configuration' section of a Docusaurus site. It features a large title 'Configuration' and a callout box with an 'INFO' icon. The text inside the box advises checking the `docusaurus.config.js` API reference for options.

INFO

Check the `docusaurus.config.js` API reference for an exhaustive list of options.

## Data Attributes

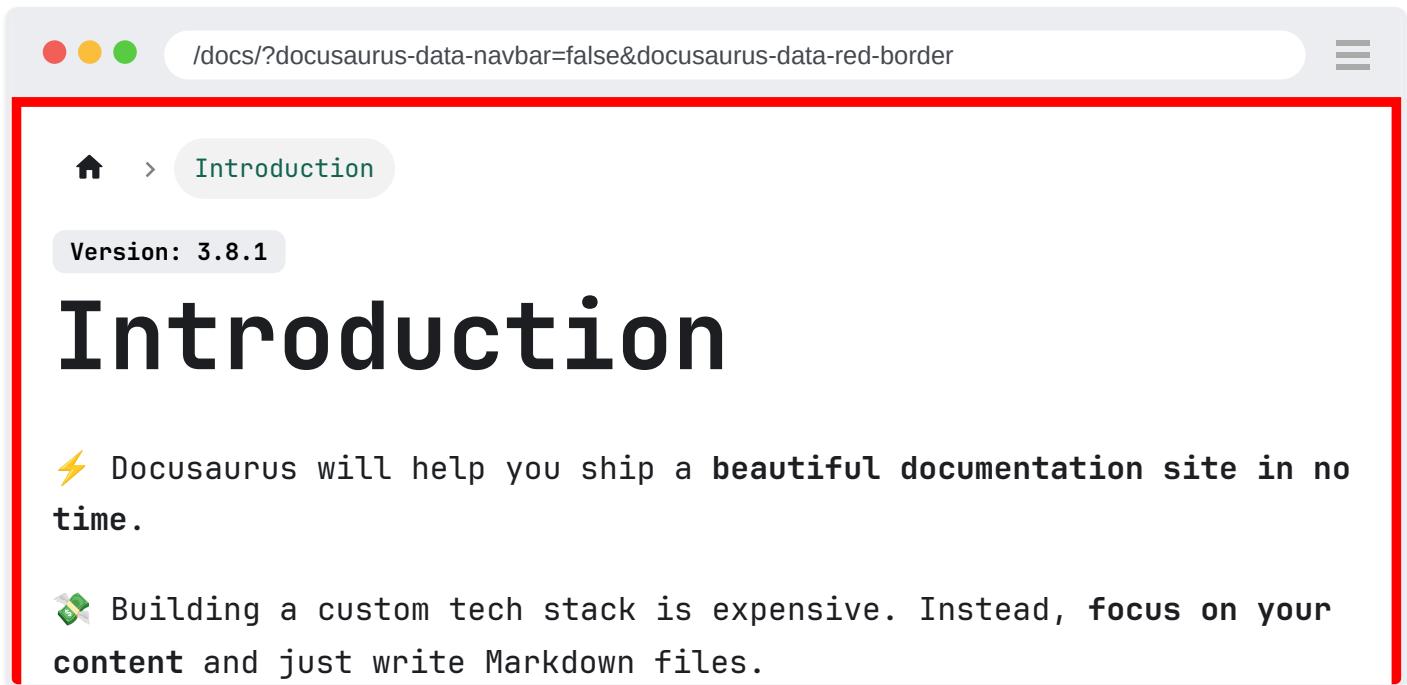
It is possible to inject `<html>` data attributes with query string parameters following the `docusaurus-data-<key>` pattern. This gives you the flexibility to style a page differently based on the query string.

For example, let's render one of our pages with a red border and no navbar:

```
/src/css/custom.css
```

```
html[data-navbar='false'] .navbar {
 display: none;
}

html[data-red-border] div#__docusaurus {
 border: red solid thick;
}
```



### IFRAME MODE

If you plan to embed some Docusaurus pages on another site through an iframe, it can be useful to create an alternative display mode and use iframe urls such as

<https://mysite.com/docs/myDoc?docusaurus-data-mode=iframe>. It is your responsibility to provide the additional styles and decide which UI elements you want to keep or hide.

## Mobile View

Docusaurus uses `996px` as the cutoff between mobile screen width and desktop. If you want your layout to be different in the mobile view, you can use media queries.

```
.banner {
 padding: 4rem;
}
```

```
/** In mobile view, reduce the padding */
@media screen and (max-width: 996px) {
 .heroBanner {
 padding: 2rem;
 }
}
```

### CUSTOMIZING THE BREAKPOINT

Some React components, such as the header and the sidebar, implement different JavaScript logic when in mobile view. If you change the breakpoint value in your custom CSS, you probably also want to update the invocations of the `useWindowSize` hook by swizzling the components it's used in and passing an explicit option argument.

## CSS modules

To style your components using [CSS Modules](#), name your stylesheet files with the `.module.css` suffix (e.g. `welcome.module.css`). Webpack will load such CSS files as CSS modules and you have to reference the class names as properties of the imported CSS module (as opposed to using plain strings). This is similar to the convention used in [Create React App](#).

`styles.module.css`

```
.main {
 padding: 12px;
}

.heading {
 font-weight: bold;
}

.contents {
 color: #ccc;
}
```

```
import styles from './styles.module.css';

function MyComponent() {
 return (
 <main className={styles.main}>
 <h1 className={styles.heading}>Hello!</h1>
 <article className={styles.contents}>Lorem Ipsum</article>
 </main>
)
}
```

```
);
}
```

The class names will be processed by webpack into a globally unique class name during build.

## CSS-in-JS

### ⚠️ WARNING

CSS-in-JS support is a work in progress, so libs like MUI may have display quirks. [Welcoming PRs](#).

## Sass/SCSS

To use Sass/SCSS as your CSS preprocessor, install the unofficial Docusaurus plugin [docusaurus-plugin-sass](#). This plugin works for both global styles and the CSS modules approach:

1. Install [docusaurus-plugin-sass](#):

[npm](#)    [Yarn](#)    [pnpm](#)    [Bun](#)

```
npm install --save docusaurus-plugin-sass sass
```

2. Include the plugin in your [docusaurus.config.js](#) file:

[docusaurus.config.js](#)

```
export default {
 // ...
 plugins: ['docusaurus-plugin-sass'],
 // ...
};
```

3. Write and import your stylesheets in Sass/SCSS as normal.

## Global styles using Sass/SCSS

You can now set the [customCss](#) property of [@docusaurus/preset-classic](#) to point to your Sass/SCSS file:

## docusaurus.config.js

```
export default {
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 // ...
 theme: {
 customCss: ['./src/css/custom.scss'],
 },
 // ...
 },
],
],
};

};
```

## Modules using Sass/SCSS

Name your stylesheet files with the `.module.scss` suffix (e.g. `welcome.module.scss`) instead of `.css`. Webpack will use `sass-loader` to preprocess your stylesheets and load them as CSS modules.

### styles.module.scss

```
.main {
 padding: 12px;

 article {
 color: #ccc;
 }
}
```

```
import styles from './styles.module.scss';

function MyComponent() {
 return (
 <main className={styles.main}>
 <article>Lorem Ipsum</article>
 </main>
);
}
```

## TypeScript support

To enable TypeScript support for Sass/SCSS modules, the TypeScript configuration should be updated to add the `docusaurus-plugin-sass` type definitions. This can be done in the `tsconfig.json` file:

```
{
 "extends": "@docusaurus/tsconfig",
 "compilerOptions": {
 ...
+ "types": ["docusaurus-plugin-sass"]
 }
}
```

# Swizzling

In this section, we will introduce how customization of layout is done in Docusaurus.

Déjà vu...?

This section is similar to [Styling and Layout](#), but this time, we will customize React components themselves, rather than what they look like. We will talk about a central concept in Docusaurus: **swizzling**, which allows **deeper site customizations**.

In practice, swizzling permits to **swap a theme component with your own implementation**, and it comes in 2 patterns:

- **Ejecting**: creates a **copy** of the original theme component, which you can fully **customize**
- **Wrapping**: creates a **wrapper** around the original theme component, which you can **enhance**

▼ Why is it called swizzling?

## Swizzling Process

### Overview

Docusaurus provides a convenient **interactive CLI** to swizzle components. You generally only need to remember the following command:

**npm**    **Yarn**    **pnpm**    **Bun**

**npm run swizzle**

It will generate a new component in your `src/theme` directory, which should look like this example:

**Ejecting**    **Wrapping**

`src/theme/SomeComponent.js`

```
import React from 'react';
```

```
export default function SomeComponent(props) {
 // You can fully customize this implementation
 // including changing the JSX, CSS and React hooks
 return (
 <div className="some-class">
 <h1>Some Component</h1>
 <p>Some component implementation details</p>
 </div>
);
}
```

To get an overview of all the themes and components available to swizzle, run:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle -- --list
```

Use `--help` to see all available CLI options, or refer to the reference [swizzle CLI documentation](#).

### REMOVING UNNEEDED SWIZZLED COMPONENTS

If you decide that a previously swizzled component is no longer necessary, you can simply remove its file(s) from the `src/theme` directory. After removing the component, make sure to restart your development server to ensure the changes are properly reflected.

### NOTE

After swizzling a component, **restart your dev server** in order for Docusaurus to know about the new component.

### PREFER STAYING ON THE SAFE SIDE

Be sure to understand [which components are safe to swizzle](#). Some components are **internal implementation details** of a theme.

### INFO

`docusaurus swizzle` is only an automated way to help you swizzle the component. You can also create the `src/theme/SomeComponent.js` file manually, and Docusaurus will [resolve it](#). There's no internal magic behind this command!

## Ejecting

Ejecting a theme component is the process of **creating a copy** of the original theme component, which you can **fully customize and override**.

To eject a theme component, use the swizzle CLI interactively, or with the `--eject` option:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle [theme name] [component name] -- --eject
```

An example:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle @docusaurus/theme-classic Footer -- --eject
```

This will copy the current `<Footer />` component's implementation to your site's `src/theme` directory. Docusaurus will now use this `<Footer>` component copy instead of the original one. You are now free to completely re-implement the `<Footer>` component.

`src/theme/Footer/index.js`

```
import React from 'react';

export default function Footer(props) {
 return (
 <footer>
 <h1>This is my custom site footer</h1>
 <p>And it is very different from the original</p>
 </footer>
);
}
```

### WARNING

Ejecting an unsafe component can sometimes lead to copying a large amount of internal code, which you now have to maintain yourself. It can make Docusaurus upgrades more difficult, as you will need to migrate your customizations if the props received or internal theme APIs used have changed.

**Prefer wrapping whenever possible:** the amount of code to maintain is smaller.

### RE-SWIZZLING

To keep ejected components up-to-date after a Docusaurus upgrade, re-run the eject command and compare the changes with `git diff`. You are also recommended to write a brief comment at the top of the file explaining what changes you have made, so that you could more easily re-apply your changes after re-ejection.

## Wrapping

Wrapping a theme component is the process of **creating a wrapper** around the original theme component, which you can **enhance**.

To wrap a theme component, use the swizzle CLI interactively, or with the `--wrap` option:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle [theme name] [component name] -- --wrap
```

An example:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle @docusaurus/theme-classic Footer -- --wrap
```

This will create a wrapper in your site's `src/theme` directory. Docusaurus will now use the `<FooterWrapper>` component instead of the original one. You can now add customizations around the original component.

`src/theme/Footer/index.js`

```
import React from 'react';
import Footer from '@theme-original/Footer';

export default function FooterWrapper(props) {
 return (
 <>
 <section>
```

```
<h2>Extra section</h2>
 <p>This is an extra section that appears above the original footer</p>
</section>
<Footer {...props} />
</>
);
}
```

▼ What is this `@theme-original` thing?



TIP

Wrapping a theme is a great way to **add extra components around existing one** without ejecting it. For example, you can easily add a custom comment system under each blog post:

src/theme/BlogPostItem.js

```
import React from 'react';
import BlogPostItem from '@theme-original/BlogPostItem';
import MyCustomCommentSystem from '@site/src/MyCustomCommentSystem';

export default function BlogPostItemWrapper(props) {
 return (
 <>
 <BlogPostItem {...props} />
 <MyCustomCommentSystem />
 </>
);
}
```

## What is safe to swizzle?

With great power comes great responsibility

Some theme components are **internal implementation details** of a theme. Docusaurus allows you to swizzle them, but it **might be risky**.

▼ Why is it risky?

For each theme component, the swizzle CLI will indicate **3 different levels of safety** declared by theme authors:

- **Safe**: this component is safe to be swizzled, its public API is considered stable, and no breaking changes should happen within a theme **major version**

- **Unsafe**: this component is a theme implementation detail, not safe to be swizzled, and breaking changes might happen within a theme **minor version**
- **Forbidden**: the swizzle CLI will prevent you from swizzling this component, because it is not designed to be swizzled at all

#### NOTE

Some components might be safe to wrap, but not safe to eject.

#### INFO

Don't be too **afraid to swizzle unsafe components**: just keep in mind that **breaking changes** might happen, and you might need to upgrade your customizations manually on minor version upgrades.

#### REPORT YOUR USE-CASE

If you have a **strong use-case for swizzling an unsafe component**, please [report it here](#) and we will work together to find a solution to make it safe.

## Which component should I swizzle?

It is not always clear which component you should swizzle exactly to achieve the desired result. `@docusaurus/theme-classic`, which provides most of the theme components, has about [100 components!](#)

#### TIP

To print an overview of all the `@docusaurus/theme-classic` components:

`npm`    `Yarn`    `pnpm`    `Bun`

```
npm run swizzle @docusaurus/theme-classic -- --list
```

You can follow these steps to locate the appropriate component to swizzle:

1. **Component description**. Some components provide a short description, which is a good way to find the right one.
2. **Component name**. Official theme components are semantically named, so you should be able to infer its function from the name. The swizzle CLI allows you to enter part of a component name to

narrow down the available choices. For example, if you run `yarn swizzle @docusaurus/theme-classic`, and enter `Doc`, only the docs-related components will be listed.

3. **Start with a higher-level component.** Components form a tree with some components importing others. Every route will be associated with one top-level component that the route will render (most of them listed in [Routing in content plugins](#)). For example, all blog post pages have `@theme/BlogPostPage` as the topmost component. You can start with swizzling this component, and then go down the component tree to locate the component that renders just what you are targeting. Don't forget to unswizzle the rest by deleting the files after you've found the correct one, so you don't maintain too many components.

4. **Read the theme source code** and use search wisely.

#### JUST ASK!

If you still have no idea which component to swizzle to achieve the desired effect, you can reach out for help in one of our [support channels](#).

We also want to understand better your fanciest customization use-cases, so please [report them](#).

## Do I need to swizzle?

Swizzling ultimately means you have to maintain some additional React code that interact with Docusaurus internal APIs. If you can, think about the following alternatives when customizing your site:

1. **Use CSS.** CSS rules and selectors can often help you achieve a decent degree of customization. Refer to [styling and layout](#) for more details.
2. **Use translations.** It may sound surprising, but translations are ultimately just a way to customize the text labels. For example, if your site's default language is `en`, you can still run `yarn write-translations -l en` and edit the `code.json` emitted. Refer to the [i18n tutorial](#) for more details.

#### TIP

**The smaller, the better.** If swizzling is inevitable, prefer to swizzle only the relevant part and maintain as little code on your own as possible. Swizzling a small component often means less risk of **breaking changes** during upgrade.

[Wrapping](#) is also a far safer alternative to [ejecting](#).

## Wrapping your site with `<Root>`

The `<Root>` component is rendered at the **very top** of the React tree, above the theme `<Layout>`, and **never unmounts**. It is the perfect place to add stateful logic that should not be re-initialized across navigations (user authentication status, shopping cart state...).

Swizzle it **manually** by creating a file at `src/theme/Root.js`:

```
src/theme/Root.js
```

```
import React from 'react';

// Default implementation, that you can customize
export default function Root({children}) {
 return <>{children}</>;
}
```



Use this component to render React Context providers.

# Static Assets

Static assets are the non-code files that are directly copied to the build output. They include images, stylesheets, favicons, fonts, etc.

By default, you are suggested to put these assets in the `static` folder. Every file you put into **that directory will be copied** into the root of the generated `build` folder with the directory hierarchy preserved. E.g. if you add a file named `sun.jpg` to the static folder, it will be copied to `build/sun.jpg`.

This means that:

- for site `baseUrl: '/'`, the image `/static/img/docusaurus.png` will be served at `/img/docusaurus.png`.
- for site `baseUrl: '/subpath/'`, the image `/static/img/docusaurus.png` will be served at `/subpath/img/docusaurus.png`.

You can customize the static directory sources in `docusaurus.config.js`. For example, we can add `public` as another possible path:

`docusaurus.config.js`

```
export default {
 title: 'My site',
 staticDirectories: ['public', 'static'],
 // ...
};
```

Now, all files in `public` as well as `static` will be copied to the build output.

## Referencing your static asset

### In JSX

In JSX, you can reference assets from the `static` folder in your code using absolute URLs, but this is not ideal because changing the site `baseUrl` will **break those links**. For the image `` served at `https://example.com/test`, the browser will try to resolve it from the URL root, i.e. as `https://example.com/img/docusaurus.png`, which will fail because it's actually served at `https://example.com/test/img/docusaurus.png`.

You can `import()` or `require()` the static asset (recommended), or use the `useBaseUrl` utility function: both prepend the `baseUrl` to paths for you.

Examples:

#### MyComponent.js

```
import DocusaurusImageUrl from '@site/static/img/docusaurus.png';

;
```

#### MyComponent.js

```

```

#### MyComponent.js

```
import useBaseUrl from '@docusaurus/useBaseUrl';

;
```

You can also import SVG files: they will be transformed into React components.

#### MyComponent.js

```
import DocusaurusLogoWithKeytar from '@site/static/img/docusaurus_keytar.svg';

<DocusaurusLogoWithKeytar title="Docusaurus Logo" className="logo" />;
```

## In Markdown

In Markdown, you can stick to using absolute paths when writing links or images **in Markdown syntax** because Docusaurus handles them as `require` calls instead of URLs when parsing the Markdown. See [Markdown static assets](#).

You write a link like this: [\[Download this document\]\(/files/note.docx\)](#)

Docusaurus changes that to: `<a href={require('static/files/note.docx')}>Download this document</a>`

## A USE MARKDOWN SYNTAX

Docusaurus will only parse links that are in Markdown syntax. If your asset references are using the JSX tag `<a>` / `<img>`, nothing will be done.

## In CSS

In CSS, the `url()` function is commonly used to reference assets like fonts and images. To reference a static asset, use absolute paths:

```
@font-face {
 font-family: 'Caroline';
 src: url('/font/Caroline.otf');
}
```

The `static/font/Caroline.otf` asset will be loaded by the bundler.

## A IMPORTANT TAKEAWAY

One important takeaway: **never hardcode your base URL!** The base URL is considered an implementation detail and should be easily changeable. All paths, even when they look like URL slugs, are actually file paths.

If you find the URL slug mental model more understandable, here's a rule of thumb:

- Pretend you have a base URL like `/test/` when writing JSX so you don't use an absolute URL path like `src="/img/thumbnail.png"` but instead `require` the asset.
- Pretend it's `/` when writing Markdown or CSS so you always use absolute paths without the base URL.

## Caveats

Keep in mind that:

- By default, none of the files in the `static` folder will be post-processed, hashed, or minified.
  - However, as we've demonstrated above, we are usually able to convert them to `require` calls for you so they do get processed. This is good for aggressive caching and better user experience.
- Missing files referenced via hard-coded absolute paths will not be detected at compilation time and will result in a 404 error.

- By default, GitHub Pages runs published files through **Jekyll**. Since Jekyll will discard any files that begin with `_`, it is recommended that you disable Jekyll by adding an empty file named `.nojekyll` file to your `static` directory if you are using GitHub pages for hosting.

# Search

There are a few options you can use to add search to your website:

-  [Algolia DocSearch \(official\)](#)
-  [Typesense DocSearch](#)
-  [Local Search](#)
-  Your own [SearchBar component](#)

## INFO

 Docusaurus provides **first-class support** for [Algolia DocSearch](#).

 Other options are **maintained by the community**: please report bugs to their respective repositories.

## Using Algolia DocSearch

Docusaurus has **official support** for [Algolia DocSearch](#).

The service is **free** for any developer documentation or technical blog: just make sure to read the [checklist](#) and [apply to the DocSearch program](#).

DocSearch crawls your website once a week (the schedule is configurable from the web interface) and aggregates all the content in an Algolia index. This content is then queried directly from your front-end using the Algolia API.

If your website is [not eligible](#) for the free, hosted version of DocSearch, or if your website sits behind a firewall and is not public, then you can [run your own](#) DocSearch crawler.

## NOTE

By default, the Docusaurus preset generates a [sitemap.xml](#) that the Algolia crawler can use.

## FROM THE OLD DOCSEARCH?

You can read more about migration from the legacy DocSearch infra in [our blog post](#) or the [DocSearch migration docs](#).

## Index Configuration

After your application has been approved and deployed, you will receive an email with all the details for you to add DocSearch to your project. Editing and managing your crawls can be done via [the web interface](#). Indices are readily available after deployment, so manual configuration usually isn't necessary.

### USE THE RECOMMENDED CRAWLER CONFIG

It is highly recommended to use our official [Docusaurus v3 crawler configuration](#). We cannot support you if you choose a different crawler configuration.

### WHEN UPDATING YOUR CRAWLER CONFIG

The crawler configuration contains a `initialIndexSettings`, which will only be used to initialize your Algolia index if it does not exist yet.

If you update your `initialIndexSettings` crawler setting, it is possible to update the index manually through the interface, but [the Algolia team recommends to delete your index and then restart a crawl](#) to fully reinitialize it with the new settings.

## Connecting Algolia

Docusaurus' own `@docusaurus/preset-classic` supports Algolia DocSearch integration. If you use the classic preset, no additional installation is needed.

### Installation steps when not using `@docusaurus/preset-classic`

Then, add an `algolia` field in your `themeConfig`. [Apply for DocSearch](#) to get your Algolia index and API key.

#### `docusaurus.config.js`

```
export default {
 // ...
 themeConfig: {
 // ...
 algolia: {
 // The application ID provided by Algolia
 appId: 'YOUR_APP_ID',
 // Public API key: it is safe to commit it
 apiKey: 'YOUR_SEARCH_API_KEY',
 indexName: 'YOUR_INDEX_NAME',
 // Optional: see doc section below
 }
 }
}
```

```

contextualSearch: true,

// Optional: Specify domains where the navigation should occur through
// window.location instead on history.push. Useful when our Algolia config crawls
// multiple documentation sites and we want to navigate with window.location.href to
// them.
externalUrlRegex: 'external\\.com|domain\\.com',

// Optional: Replace parts of the item URLs from Algolia. Useful when using
// the same search index for multiple deployments using a different baseUrl. You can
// use regexp or string in the `from` param. For example: localhost:3000 vs
// myCompany.com/docs
replaceSearchResultPathname: {
 from: '/docs/', // or as RegExp: /\docs\//
 to: '/',
},

// Optional: Algolia search parameters
searchParameters: {},

// Optional: path for search page that enabled by default (`false` to
// disable it)
searchPagePath: 'search',

// Optional: whether the insights feature is enabled or not on Docsearch
//(`false` by default)
insights: false,

//... other Algolia params
},
},
};

```

### ! INFO

The `searchParameters` option used to be named `algoliaOptions` in Docusaurus v1.

Refer to its [official DocSearch documentation](#) for possible values.

### ! WARNING

The search feature will not work reliably until Algolia crawls your site.

If search doesn't work after any significant change, please use the Algolia dashboard to **trigger a new crawl**.

## Contextual search

Contextual search is **enabled by default**.

It ensures that search results are **relevant to the current language and version**.

docusaurus.config.js

```
export default {
 // ...
 themeConfig: {
 // ...
 algolia: {
 contextualSearch: true,
 },
 },
};
```

Let's consider you have 2 docs versions (**v1** and **v2**) and 2 languages (`en` and `fr`).

When browsing v2 docs, it would be odd to return search results for the v1 documentation. Sometimes v1 and v2 docs are quite similar, and you would end up with duplicate search results for the same query (one result per version).

Similarly, when browsing the French site, it would be odd to return search results for the English docs.

To solve this problem, the contextual search feature understands that you are browsing a specific docs version and language, and will create the search query filters dynamically.

- on `/en/docs/v1/myDoc`, search results will only include **English** results for the **v1** docs (+ other unversioned pages)
- on `/fr/docs/v2/myDoc`, search results will only include **French** results for the **v2** docs (+ other unversioned pages)

### !(INFO)

When using `contextualSearch: true` (default), the contextual facet filters will be merged with the ones provided with `algolia.searchParameters.facetFilters`.

For specific needs, you can disable `contextualSearch` and define your own `facetFilters`:

docusaurus.config.js

```
export default {
 // ...
```

```
themeConfig: {
 // ...
 algolia: {
 contextualSearch: false,
 searchParameters: {
 facetFilters: ['language:en', ['filter1', 'filter2'], 'filter3'],
 },
 },
};
```

Refer to the relevant [Algolia faceting documentation](#).

### ⚠ CONTEXTUAL SEARCH DOESN'T WORK?

If you only get search results when Contextual Search is disabled, this is very likely because of an [index configuration issue](#).

## Styling your Algolia search

By default, DocSearch comes with a fine-tuned theme that was designed for accessibility, making sure that colors and contrasts respect standards.

Still, you can reuse the [Infima CSS variables](#) from Docusaurus to style DocSearch by editing the `/src/css/custom.css` file.

`/src/css/custom.css`

```
[data-theme='light'] .DocSearch {
 /* --docsearch-primary-color: var(--ifm-color-primary); */
 /* --docsearch-text-color: var(--ifm-font-color-base); */
 --docsearch-muted-color: var(--ifm-color-secondary-darkest);
 --docsearch-container-background: rgba(94, 100, 112, 0.7);
 /* Modal */
 --docsearch-modal-background: var(--ifm-color-secondary-lighter);
 /* Search box */
 --docsearch-searchbox-background: var(--ifm-color-secondary);
 --docsearch-searchbox-focus-background: var(--ifm-color-white);
 /* Hit */
 --docsearch-hit-color: var(--ifm-font-color-base);
 --docsearch-hit-active-color: var(--ifm-color-white);
 --docsearch-hit-background: var(--ifm-color-white);
 /* Footer */
 --docsearch-footer-background: var(--ifm-color-white);
}
```

```
[data-theme='dark'] .DocSearch {
 --docsearch-text-color: var(--ifm-font-color-base);
 --docsearch-muted-color: var(--ifm-color-secondary-darkest);
 --docsearch-container-background: rgba(47, 55, 69, 0.7);
 /* Modal */
 --docsearch-modal-background: var(--ifm-background-color);
 /* Search box */
 --docsearch-searchbox-background: var(--ifm-background-color);
 --docsearch-searchbox-focus-background: var(--ifm-color-black);
 /* Hit */
 --docsearch-hit-color: var(--ifm-font-color-base);
 --docsearch-hit-active-color: var(--ifm-color-white);
 --docsearch-hit-background: var(--ifm-color-emphasis-100);
 /* Footer */
 --docsearch-footer-background: var(--ifm-background-surface-color);
 --docsearch-key-gradient: linear-gradient(
 -26.5deg,
 var(--ifm-color-emphasis-200) 0%,
 var(--ifm-color-emphasis-100) 100%
);
}
```

## Customizing the Algolia search behavior

Algolia DocSearch supports a [list of options](#) that you can pass to the `algolia` field in the `docusaurus.config.js` file.

### `docusaurus.config.js`

```
export default {
 themeConfig: {
 // ...
 algolia: {
 apiKey: 'YOUR_API_KEY',
 indexName: 'YOUR_INDEX_NAME',
 // Options...
 },
 },
};
```

## Editing the Algolia search component

If you prefer to edit the Algolia search React component, [swizzle](#) the `SearchBar` component in `@docusaurus/theme-search-algolia`:

```
npm run swizzle @docusaurus/theme-search-algolia SearchBar
```

## Troubleshooting

Here are the most common issues Docusaurus users face when using Algolia DocSearch.

### No Search Results

Seeing no search results is usually related to an **index configuration problem**.

- ▼ How to check if I have a config problem?

#### USE THE RECOMMENDED CONFIGURATION

We recommend a specific crawler configuration for a good reason. We cannot support you if you choose to use a different configuration.

You can fix index configuration problems by following those steps:

1. Use the [recommend crawler configuration](#)
2. Delete your index through the UI
3. Trigger a new crawl through the UI
4. Check your index is recreated with the appropriate faceting fields: `docusaurus_tag`, `language`, `lang`, `version`, `type`
5. See that you now get search results, even with [Contextual Search](#) enabled

## Support

The Algolia DocSearch team can help you figure out search problems on your site.

You can reach out to Algolia via [their support page](#) or on [Discord](#).

Docusaurus also has an `#algolia` channel on [Discord](#).

## Using Typesense DocSearch

Typesense DocSearch works similar to Algolia DocSearch, except that your website is indexed into a Typesense search cluster.

Typesense is an [open source](#) instant-search engine that you can either:

- [Self-Host](#) on your own servers or
- Use the Managed [Typesense Cloud](#) service.

Similar to Algolia DocSearch, there are two components:

- [typesense-docsearch-scraper](#) - which scrapes your website and indexes the data in your Typesense cluster.
- [docusaurus-theme-search-typesense](#) - a search bar UI component to add to your website.

Read a step-by-step walk-through of how to [run typesense-docsearch-scraper here](#) and how to [install the Search Bar in your Docusaurus Site here](#).

## Using Local Search

You can use a local search plugin for websites where the search index is small and can be downloaded to your users' browsers when they visit your website.

You'll find a list of community-supported [local search plugins listed here](#).

## Using your own search

To use your own search, swizzle the `earchBar` component in `@docusaurus/theme-classic`

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run swizzle @docusaurus/theme-classic SearchBar
```

This will create an `src/theme/SearchBar` file in your project folder. Restart your dev server and edit the component, you will see that Docusaurus uses your own `earchBar` component now.

**Notes:** You can alternatively [swizzle from Algolia SearchBar](#) and create your own search component from there.

# Browser support

Docusaurus allows sites to define the list of supported browsers through a [browserslist configuration](#).

## Purpose

Websites need to balance between backward compatibility and bundle size. As old browsers do not support modern APIs or syntax, more code is needed to implement the same functionality.

For example, you may use the [optional chaining syntax](#):

```
const value = obj?.prop?.val;
```

...which unfortunately is only recognized by browser versions released after 2020. To be compatible with earlier browser versions, when building your site for production, our JS loader will transpile your code to a more verbose syntax:

```
var _obj, _obj$prop;

const value =
(_obj = obj) === null || _obj === void 0
? void 0
: (_obj$prop = _obj.prop) === null || _obj$prop === void 0
? void 0
: _obj$prop.val;
```

However, this penalizes all other users with increased site load time because the 29-character line now becomes 168 characters—a 6-fold increase! (In practice, it will be better because the names used will be shorter.) As a tradeoff, the JS loader only transpiles the syntax to the degree that's supported by all browser versions defined in the browser list.

The browser list by default is provided through the `package.json` file as a root `browserslist` field.

### WARNING

On old browsers, the compiled output will use unsupported (too recent) JS syntax, causing React to fail to initialize and end up with a static website with only HTML/CSS and no JS.

## Default values

Websites initialized with the default classic template has the following in `package.json`:

### package.json

```
{
 "name": "docusaurus",
 // ...
 "browserslist": {
 "production": [">0.5%", "not dead", "not op_mini all"],
 "development": [
 "last 1 chrome version",
 "last 1 firefox version",
 "last 1 safari version"
]
 }
 // ...
}
```

Explained in natural language, the browsers supported in production are those:

- With more than 0.5% of market share; *and*
- Has official support or updates in the past 24 months (the opposite of "dead"); *and*
- Is not Opera Mini.

And browsers used in development are:

- The latest version of Chrome or Firefox or Safari.

You can "evaluate" any config with the `browserslist` CLI to obtain the actual list:

```
npx browserslist --env="production"
```

The output is all browsers supported in production. Below is the output in January 2022:

```
and_chr 96
and_uc 12.12
chrome 96
chrome 95
chrome 94
edge 96
firefox 95
firefox 94
ie 11
ios_saf 15.2
```

```
ios_saf 15.0-15.1
ios_saf 14.5-14.8
ios_saf 14.0-14.4
ios_saf 12.2-12.5
opera 82
opera 81
safari 15.1
safari 14.1
safari 13.1
```

## Read more

You may wish to visit the [browserslist documentation](#) for more specifications, especially the accepted [query values](#) and [best practices](#).

# Search engine optimization (SEO)

Docusaurus supports search engine optimization in a variety of ways.

## Global metadata

Provide global meta attributes for the entire site through the [site configuration](#). The metadata will all be rendered in the HTML `<head>` using the key-value pairs as the prop name and value. The `metadata` attribute is a convenient shortcut to declare `<meta>` tags, but it is also possible to inject arbitrary tags in `<head>` with the `headTags` attribute.

```
docusaurus.config.js
```

```
export default {
 themeConfig: {
 // Declare some <meta> tags
 metadata: [
 {name: 'keywords', content: 'cooking, blog'},
 {name: 'twitter:card', content: 'summary_large_image'},
],
 },
 headTags: [
 // Declare a <link> preconnect tag
 {
 tagName: 'link',
 attributes: {
 rel: 'preconnect',
 href: 'https://example.com',
 },
 },
 // Declare some json-ld structured data
 {
 tagName: 'script',
 attributes: {
 type: 'application/ld+json',
 },
 innerHTML: JSON.stringify({
 '@context': 'https://schema.org/',
 '@type': 'Organization',
 name: 'Meta Open Source',
 url: 'https://opensource.fb.com/',
 logo: 'https://opensource.fb.com/img/logos/Meta-Open-Source.svg',
 }),
 },
],
}
```

```
],
};
```

Docusaurus adds some metadata out-of-the-box. For example, if you have configured [i18n](#), you will get a alternate link.

To read more about types of meta tags, visit [the MDN docs](#).

## Single page metadata

Similar to [global metadata](#), Docusaurus also allows for the addition of meta-information to individual pages. Follow [this guide](#) for configuring the `<head>` tag. In short:

```
my-markdown-page.mdx
```

### # A cooking guide

```
<head>
 <meta name="keywords" content="cooking, blog" />
 <meta name="twitter:card" content="summary_large_image" />
 <link rel="preconnect" href="https://example.com" />
 <script type="application/ld+json">
 {JSON.stringify({
 '@context': 'https://schema.org/',
 '@type': 'Organization',
 name: 'Meta Open Source',
 url: 'https://opensource.fb.com/',
 logo: 'https://opensource.fb.com/img/logos/Meta-Open-Source.svg',
 })}
 </script>
</head>
```

```
Some content...
```

Docusaurus automatically adds `description`, `title`, canonical URL links, and other useful metadata to each Markdown page. They are configurable through [front matter](#):

```

title: Title for search engines; can be different from the actual heading
description: A short description of this page
image: a thumbnail image to be shown in social media cards
keywords: [keywords, describing, the main topics]

```

When creating your React page, adding these fields in `Layout` would also improve SEO.

### TIP

Prefer to use `front matter` for fields like `description` and `keywords`: Docusaurus will automatically apply this to both `description` and `og:description`, while you would have to manually declare two metadata tags when using the `<head>` tag.

### INFO

The official plugins all support the following `front matter`: `title`, `description`, `keywords` and `image`. Refer to their respective API documentation for additional `front matter` support:

- [Docs front matter](#)
- [Blog front matter](#)
- [Pages front matter](#)

For JSX pages, you can use the Docusaurus `<Head>` component.

my-react-page.jsx

```
import React from 'react';
import Layout from '@theme/Layout';
import Head from '@docusaurus/Head';

export default function page() {
 return (
 <Layout title="Page" description="A React page demo">
 <Head>
 <meta property="og:image" content="image.png" />
 <meta name="twitter:card" content="summary_large_image" />
 <link rel="preconnect" href="https://example.com" />
 <script type="application/ld+json">
 {JSON.stringify({
 '@context': 'https://schema.org/',
 '@type': 'Organization',
 name: 'Meta Open Source',
 url: 'https://opensource.fb.com/',
 logo: 'https://opensource.fb.com/img/logos/Meta-Open-Source.svg',
 })}
 </script>
 </Head>
 {/* ... */}
 </Layout>
);
}
```



TIP

For convenience, the default theme `<Layout>` component accept `title` and `description` as props.

## Static HTML generation

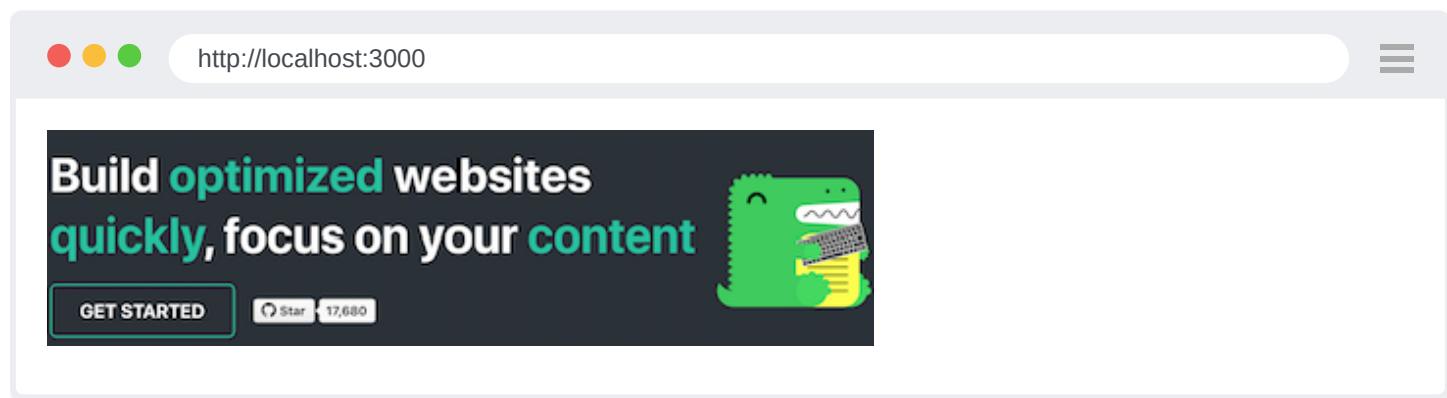
Docusaurus is a static site generator—HTML files are statically generated for every URL route, which helps search engines discover your content more easily.

## Image meta description

The alt tag for an image tells the search engine what the image is about, and is used when the image can't be visually seen, e.g. when using a screen reader, or when the image is broken. Alt tags are commonly supported in Markdown.

You may also add a title for your image—this doesn't impact SEO much but is displayed as a tooltip when hovering above the image, usually used to provide hints.

```
![Docusaurus banner](./assets/docusaurus-asset-example-banner.png 'Image title')
```



## Rich search information

Docusaurus blogs support [rich search results](#) out-of-the-box to get maximum search engine experience. The information is created depending on your meta information in blog/global configuration. In order to get the benefits of the rich search information, fill in the information about the post's publish date, authors, and image, etc. Read more about the meta-information [here](#).

# Robots file

A `robots.txt` file regulates search engines' behavior about which should be displayed and which shouldn't. You can provide it as [static asset](#). The following would allow access to all sub-pages from all requests:

```
static/robots.txt
```

```
User-agent: *
Disallow:
```

Read more about the robots file in [the Google documentation](#).

## ⚠️ WARNING

**Important:** the `robots.txt` file does **not** prevent HTML pages from being indexed.

To prevent your whole Docusaurus site from being indexed, use the [noIndex](#) site config. Some [hosting providers](#) may also let you configure a `X-Robots-Tag: noindex` HTTP header (GitHub Pages does not support this).

To prevent a single page from being indexed, use `<meta name="robots" content="noindex">` as [page metadata](#). Read more about the [robots meta tag](#).

# Sitemap file

Docusaurus provides the [@docusaurus/plugin-sitemap](#) plugin, which is shipped with [preset-classic](#) by default. It autogenerated a `sitemap.xml` file which will be available at [https://example.com/\[baseUrl\]/sitemap.xml](https://example.com/[baseUrl]/sitemap.xml) after the production build. This sitemap metadata helps search engine crawlers crawl your site more accurately.

## 💡 TIP

The sitemap plugin automatically filters pages containing a `noindex` [robots meta directive](#).

For example, [/examples/noIndex](#) is not included in the [Docusaurus sitemap.xml file](#) because it contains the following [page metadata](#):

```
<head>
 <meta name="robots" content="noindex,nofollow" />
</head>
```

# Human readable links

Docusaurus uses your file names as links, but you can always change that using slugs, see this [tutorial](#) for more details.

# Structured content

Search engines rely on the HTML markup such as `<h2>`, `<table>`, etc., to understand the structure of your webpage. When Docusaurus renders your pages, semantic markup, e.g. `<aside>`, `<nav>`, `<main>`, are used to divide the different sections of the page, helping the search engine to locate parts like sidebar, navbar, and the main page content.

Most [CommonMark](#) syntaxes have their corresponding HTML tags. By using Markdown consistently in your project, you will make it easier for search engines to understand your page content.

# Using Plugins

The Docusaurus core doesn't provide any feature of its own. All features are delegated to individual plugins: the [docs](#) feature provided by the [docs plugin](#); the [blog](#) feature provided by the [blog plugin](#); or individual [pages](#) provided by the [pages plugin](#). If there are no plugins installed, the site won't contain any routes.

You may not need to install common plugins one-by-one though: they can be distributed as a bundle in a [preset](#). For most users, plugins are configured through the preset configuration.

We maintain a [list of official plugins](#), but the community has also created some [unofficial plugins](#). If you want to add any feature: autogenerated doc pages, executing custom scripts, integrating other services... be sure to check out the list: someone may have implemented it for you!

If you are feeling energetic, you can also read [the plugin guide](#) or [plugin method references](#) for how to make a plugin yourself.

## Installing a plugin

A plugin is usually an npm package, so you install them like other npm packages using npm.

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm install --save docusaurus-plugin-name
```

Then you add it in your site's `docusaurus.config.js`'s `plugins` option:

`docusaurus.config.js`

```
export default {
 // ...
 plugins: ['@docusaurus/plugin-content-pages'],
};
```

Docusaurus can also load plugins from your local directory, with something like the following:

`docusaurus.config.js`

```
export default {
 // ...
 plugins: ['./src/plugins/docusaurus-local-plugin'],
};
```

Paths should be absolute or relative to the config file.

## Configuring plugins

For the most basic usage of plugins, you can provide just the plugin name or the path to the plugin.

However, plugins can have options specified by wrapping the name and an options object in a two-member tuple inside your config. This style is usually called "Babel Style".

docusaurus.config.js

```
export default {
 // ...
 plugins: [
 [
 '@docusaurus/plugin-xxx',
 {
 /* options */
 },
],
],
};
```

Example:

docusaurus.config.js

```
export default {
 plugins: [
 // Basic usage.
 '@docusaurus/plugin-debug',

 // With options object (babel style)
 [
 '@docusaurus/plugin-sitemap',
 {
 changefreq: 'weekly',
 },
],
],
};
```

```
],
],
};
```

## Multi-instance plugins and plugin IDs

All Docusaurus content plugins can support multiple plugin instances. For example, it may be useful to have [multiple docs plugin instances](#) or [multiple blogs](#). It is required to assign a unique ID to each plugin instance, and by default, the plugin ID is `default`.

`docusaurus.config.js`

```
export default {
 plugins: [
 [
 '@docusaurus/plugin-content-docs',
 {
 id: 'docs-1',
 // other options
 },
],
 [
 '@docusaurus/plugin-content-docs',
 {
 id: 'docs-2',
 // other options
 },
],
],
};
```

### NOTE

At most one plugin instance can be the "default plugin instance", by omitting the `id` attribute, or using `id: 'default'`.

## Using themes

Themes are loaded in the exact same way as plugins. From the consumer perspective, the `themes` and `plugins` entries are interchangeable when installing and configuring a plugin. The only nuance is that themes are loaded after plugins, and it's possible for [a theme to override a plugin's default theme components](#).



TIP

The `themes` and `plugins` options lead to different shorthand resolutions, so if you want to take advantage of shorthands, be sure to use the right entry!

`docusaurus.config.js`

```
export default {
 // ...
 themes: ['@docusaurus/theme-classic', '@docusaurus/theme-live-codeblock'],
};
```

## Using presets

Presets are bundles of plugins and themes. For example, instead of letting you register and configure `@docusaurus/plugin-content-docs`, `@docusaurus/plugin-content-blog`, etc. one after the other in the config file, we have `@docusaurus/preset-classic` preset allows you to configure them in one centralized place.

### `@docusaurus/preset-classic`

The classic preset is shipped by default to new Docusaurus websites created with `create-docusaurus`. It contains the following themes and plugins:

- `@docusaurus/theme-classic`
- `@docusaurus/theme-search-algolia`
- `@docusaurus/plugin-content-docs`
- `@docusaurus/plugin-content-blog`
- `@docusaurus/plugin-content-pages`
- `@docusaurus/plugin-debug`
- `@docusaurus/plugin-google-gtag`
- `@docusaurus/plugin-google-tag-manager`
- `@docusaurus/plugin-google-analytics` (deprecated)
- `@docusaurus/plugin-sitemap`
- `@docusaurus/plugin-svgr`

The classic preset will relay each option entry to the respective plugin/theme.

## docusaurus.config.js

```
export default {
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 // Debug defaults to true in dev, false in prod
 debug: undefined,
 // Will be passed to @docusaurus/theme-classic.
 theme: {
 customCss: ['./src/css/custom.css'],
 },
 // Will be passed to @docusaurus/plugin-content-docs (false to disable)
 docs: {},
 // Will be passed to @docusaurus/plugin-content-blog (false to disable)
 blog: {},
 // Will be passed to @docusaurus/plugin-content-pages (false to disable)
 pages: {},
 // Will be passed to @docusaurus/plugin-sitemap (false to disable)
 sitemap: {},
 // Will be passed to @docusaurus/plugin-svgr (false to disable)
 svgr: {},
 // Will be passed to @docusaurus/plugin-google-gtag (only enabled when
 // explicitly specified)
 gtag: {},
 // Will be passed to @docusaurus/plugin-google-tag-manager (only enabled
 // when explicitly specified)
 googleTagManager: {},
 // DEPRECATED: Will be passed to @docusaurus/plugin-google-analytics
 // (only enabled when explicitly specified)
 googleAnalytics: {},
 },
],
],
};
```

## Installing presets

A preset is usually an npm package, so you install them like other npm packages using npm.

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm install --save @docusaurus/preset-classic
```

Then, add it in your site's `docusaurus.config.js`'s `presets` option:

### docusaurus.config.js

```
export default {
 // ...
 presets: ['@docusaurus/preset-classic'],
};
```

Preset paths can be relative to the config file:

### docusaurus.config.js

```
export default {
 // ...
 presets: ['./src/presets/docusaurus-local-preset'],
};
```

## Creating presets

A preset is a function with the same shape as the [plugin constructor](#). It should return an object of `{ plugins: PluginConfig[], themes: PluginConfig[] }`, in the same as how they are accepted in the site config. For example, you can specify a preset that includes the following themes and plugins:

### src/presets/docusaurus-preset-multi-docs.js

```
export default function preset(context, opts = {}) {
 return {
 themes: [['docusaurus-theme-awesome', opts.theme]],
 plugins: [
 // Using three docs plugins at the same time!
 // Assigning a unique ID for each without asking the user to do it
 ['@docusaurus/plugin-content-docs', {...opts.docs1, id: 'docs1'}],
 ['@docusaurus/plugin-content-docs', {...opts.docs2, id: 'docs2'}],
 ['@docusaurus/plugin-content-docs', {...opts.docs3, id: 'docs3'}],
],
 };
}
```

Then in your Docusaurus config, you may configure the preset:

### docusaurus.config.js

```

export default {
 presets: [
 [
 './src/presets/docusaurus-preset-multi-docs.js',
 {
 theme: {hello: 'world'},
 docs1: {path: '/docs'},
 docs2: {path: '/community'},
 docs3: {path: '/api'},
 },
],
],
};


```

This is equivalent of doing:

#### docusaurus.config.js

```

export default {
 themes: [['docusaurus-theme-awesome', {hello: 'world'}]],
 plugins: [
 ['@docusaurus/plugin-content-docs', {id: 'docs1', path: '/docs'}],
 ['@docusaurus/plugin-content-docs', {id: 'docs2', path: '/community'}],
 ['@docusaurus/plugin-content-docs', {id: 'docs3', path: '/api'}],
],
};


```

This is especially useful when some plugins and themes are intended to be used together. You can even link their options together, e.g. pass one option to multiple plugins.

## Module shorthands

Docusaurus supports shorthands for plugins, themes, and presets. When it sees a plugin/theme/preset name, it tries to load one of the following, in that order:

- [name] (like content-docs)
- @docusaurus/[moduleType]-[name] (like @docusaurus/plugin-content-docs)
- docusaurus-[moduleType]-[name] (like docusaurus-plugin-content-docs)

where moduleType is one of 'preset', 'theme', 'plugin', depending on which field the module name is declared in. The first module name that's successfully found is loaded.

If the name is scoped (beginning with @), the name is first split into scope and package name by the first slash:

```
@scope
^----^
 scope (no name!)
```

```
@scope/awesome
^----^ ^----^
 scope name
```

```
@scope/awesome/main
^----^ ^-----^
 scope name
```

If there is no name (like @jquery), [scope]/docusaurus-[moduleType] (i.e. @jquery/docusaurus-plugin) is loaded. Otherwise, the following are attempted:

- [scope]/[name] (like @jquery/content-docs)
- [scope]/docusaurus-[moduleType]-[name] (like @jquery/docusaurus-plugin-content-docs)

Below are some examples, for a plugin registered in the `plugins` field. Note that unlike [ESLint](#) or [Babel](#) where a consistent naming convention for plugins is mandated, Docusaurus permits greater naming freedom, so the resolutions are not certain, but follows the priority defined above.

Declaration	May be resolved as
awesome	docusaurus-plugin-awesome
sitemap	@docusaurus/plugin-sitemap
@my-company	@my-company/docusaurus-plugin (the only possible resolution!)
@my-company/awesome	@my-company/docusaurus-plugin-awesome
@my-company/awesome/web	@my-company/docusaurus-plugin-awesome/web

# Deployment

To build the static files of your website for production, run:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run build
```

Once it finishes, the static files will be generated within the `build` directory.

 **NOTE**

The only responsibility of Docusaurus is to build your site and emit static files in `build`.

It is now up to you to choose how to host those static files.

You can deploy your site to static site hosting services such as [Vercel](#), [GitHub Pages](#), [Netlify](#), [Render](#), and [Surge](#).

A Docusaurus site is statically rendered, and it can generally work without JavaScript!

## Configuration

The following parameters are required in `docusaurus.config.js` to optimize routing and serve files from the correct location:

Name	Description
<code>url</code>	URL for your site. For a site deployed at <a href="https://my-org.com/my-project/">https://my-org.com/my-project/</a> , <code>url</code> is <a href="https://my-org.com/">https://my-org.com/</a> .
<code>baseUrl</code>	Base URL for your project, with a trailing slash. For a site deployed at <a href="https://my-org.com/my-project/">https://my-org.com/my-project/</a> , <code>baseUrl</code> is <code>/my-project/</code> .

## Testing your Build Locally

It is important to test your build locally before deploying it for production. Docusaurus provides a `docusaurus serve` command for that:

`npm`   `Yarn`   `pnpm`   `Bun`

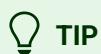
```
npm run serve
```

By default, this will load your site at `http://localhost:3000/`.

## Trailing slash configuration

Docusaurus has a `trailingSlash` config to allow customizing URLs/links and emitted filename patterns.

The default value generally works fine. Unfortunately, each static hosting provider has a **different behavior**, and deploying the exact same site to various hosts can lead to distinct results. Depending on your host, it can be useful to change this config.



Use [slorber/trailing-slash-guide](#) to understand better the behavior of your host and configure `trailingSlash` appropriately.

## Using environment variables

Putting potentially sensitive information in the environment is common practice. However, in a typical Docusaurus website, the `docusaurus.config.js` file is the only interface to the Node.js environment (see [our architecture overview](#)), while everything else (MDX pages, React components, etc.) are client side and do not have direct access to the `process` global variable. In this case, you can consider using `customFields` to pass environment variables to the client side.

`docusaurus.config.js`

```
// If you are using dotenv (https://www.npmjs.com/package/dotenv)
import 'dotenv/config';

export default {
 title: '...',
```

```
url: process.env.URL, // You can use environment variables to control site
specifies as well
customFields: {
 // Put your custom environment here
 teamEmail: process.env.EMAIL,
},
};
```

## home.jsx

```
import useDocusaurusContext from '@docusaurus/useDocusaurusContext';

export default function Home() {
 const {
 siteConfig: {customFields},
 } = useDocusaurusContext();
 return <div>Contact us through {customFields.teamEmail}!</div>;
}
```

# Choosing a hosting provider

There are a few common hosting options:

- [Self hosting](#) with an HTTP server like Apache2 or Nginx.
- Jamstack providers (e.g. [Netlify](#) and [Vercel](#)). We will use them as references, but the same reasoning can apply to other providers.
- [GitHub Pages](#) (by definition, it is also Jamstack, but we compare it separately).

If you are unsure of which one to choose, ask the following questions:

- ▼ How many resources (money, person-hours, etc.) am I willing to invest in this?
- ▼ How much server-side customization do I need?
- ▼ Do I need collaboration-friendly deployment workflows?

There isn't a silver bullet. You need to weigh your needs and resources before making a choice.

# Self-Hosting

Docusaurus can be self-hosted using `docusaurus serve`. Change port using `--port` and `--host` to change host.

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run serve -- --build --port 80 --host 0.0.0.0
```

### WARNING

It is not the best option, compared to a static hosting provider / CDN.

### WARNING

In the following sections, we will introduce a few common hosting providers and how they should be configured to deploy Docusaurus sites most efficiently. Docusaurus is not affiliated with any of these services, and this information is provided for convenience only. Some of the write-ups are provided by third-parties, and recent API changes may not be reflected on our side. If you see outdated content, PRs are welcome.

Because we can only provide this content on a best-effort basis only, we have stopped accepting PRs adding new hosting options. You can, however, publish your writeup on a separate site (e.g. your blog, or the provider's official website), and ask us to include a link to your writeup.

## Deploying to Netlify

To deploy your Docusaurus sites to [Netlify](#), first make sure the following options are properly configured:

`docusaurus.config.js`

```
export default {
 url: 'https://docusaurus-2.netlify.app', // Url to your site with no trailing
 slash
 baseUrl: '/', // Base directory of your site relative to your repo
 // ...
};
```

Then, [create your site with Netlify](#).

While you set up the site, specify the build commands and directories as follows:

- build command: `npm run build`
- publish directory: `build`

If you did not configure these build options, you may still go to "Site settings" -> "Build & deploy" after your site is created.

Once properly configured with the above options, your site should deploy and automatically redeploy upon merging to your deploy branch, which defaults to `main`.

### WARNING

Some Docusaurus sites put the `docs` folder outside of `website` (most likely former Docusaurus v1 sites):

```
repo # git root
├── docs # MD files
└── website # Docusaurus root
```

If you decide to use the `website` folder as Netlify's base directory, Netlify will not trigger builds when you update the `docs` folder, and you need to configure a custom ignore command:

```
website/netlify.toml
```

```
[build]
ignore = "git diff --quiet $CACHED_COMMIT_REF $COMMIT_REF/docs/"
```

### WARNING

By default, Netlify adds trailing slashes to Docusaurus URLs.

It is recommended to disable the Netlify setting Post Processing > Asset Optimization > PrettyUrls to prevent lowercase URLs, unnecessary redirects, and 404 errors.

**Be very careful:** the `Disable asset optimization` global checkbox is broken and does not really disable the `Pretty URLs` setting in practice. Please make sure to **uncheck it independently**.

If you want to keep the `Pretty URLs` Netlify setting on, adjust the `trailingSlash` Docusaurus config appropriately.

Refer to [slorber/trailing-slash-guide](#) for more information.

# Deploying to Vercel

Deploying your Docusaurus project to [Vercel](#) will provide you with [various benefits](#) in the areas of performance and ease of use.

To deploy your Docusaurus project with a [Vercel for Git Integration](#), make sure it has been pushed to a Git repository.

Import the project into Vercel using the [Import Flow](#). During the import, you will find all relevant options preconfigured for you; however, you can choose to change any of these [options](#).

After your project has been imported, all subsequent pushes to branches will generate [Preview Deployments](#), and all changes made to the [Production Branch](#) (usually "main" or "master") will result in a [Production Deployment](#).

# Deploying to GitHub Pages

Docusaurus provides an easy way to publish to [GitHub Pages](#), which comes free with every GitHub repository.

## Overview

Usually, there are two repositories (at least two branches) involved in a publishing process: the branch containing the source files, and the branch containing the build output to be served with GitHub Pages. In the following tutorial, they will be referred to as "**source**" and "**deployment**", respectively.

Each GitHub repository is associated with a GitHub Pages service. If the deployment repository is called `my-org/my-project` (where `my-org` is the organization name or username), the deployed site will appear at <https://my-org.github.io/my-project/>. If the deployment repository is called `my-org/my-org.github.io` (the *organization GitHub Pages repo*), the site will appear at <https://my-org.github.io/>.

### ! INFO

In case you want to use your custom domain for GitHub Pages, create a `CNAME` file in the `static` directory. Anything within the `static` directory will be copied to the root of the `build` directory for deployment. When using a custom domain, you should be able to move back from `baseUrl: '/projectName/'` to `baseUrl: '/'`, and also set your `url` to your custom domain.

You may refer to GitHub Pages' documentation [User, Organization, and Project Pages](#) for more details.

GitHub Pages picks up deploy-ready files (the output from `docusaurus build`) from the default branch (`master` / `main`, usually) or the `gh-pages` branch, and either from the root or the `/docs` folder. You can configure that through `Settings > Pages` in your repository. This branch will be called the "deployment branch".

We provide a `docusaurus deploy` command that helps you deploy your site from the source branch to the deployment branch in one command: clone, build, and commit.

## `docusaurus.config.js` settings

First, modify your `docusaurus.config.js` and add the following params:

Name	Description
<code>organizationName</code>	The GitHub user or organization that owns the deployment repository.
<code>projectName</code>	The name of the deployment repository.
<code>deploymentBranch</code>	The name of the deployment branch. It defaults to ' <code>gh-pages</code> ' for non-organization GitHub Pages repos ( <code>projectName</code> not ending in <code>.github.io</code> ). Otherwise, it needs to be explicit as a config field or environment variable.

These fields also have their environment variable counterparts which have a higher priority:

`ORGANIZATION_NAME`, `PROJECT_NAME`, and `DEPLOYMENT_BRANCH`.

### WARNING

GitHub Pages adds a trailing slash to Docusaurus URLs by default. It is recommended to set a `trailingSlash` config (`true` or `false`, not `undefined`).

Example:

### `docusaurus.config.js`

```
export default {
 // ...
 url: 'https://endiliey.github.io', // Your website URL
 baseUrl: '/',
 projectName: 'endiliey.github.io',
 organizationName: 'endiliey',
 trailingSlash: false,
```

```
// ...
};
```

### WARNING

By default, GitHub Pages runs published files through [Jekyll](#). Since Jekyll will discard any files that begin with `_`, it is recommended that you disable Jekyll by adding an empty file named `.nojekyll` file to your `static` directory.

## Environment settings

Name	Description
<code>USE_SSH</code>	Set to <code>true</code> to use SSH instead of the default HTTPS for the connection to the GitHub repo. If the source repo URL is an SSH URL (e.g. <code>git@github.com:facebook/docusaurus.git</code> ), <code>USE_SSH</code> is inferred to be <code>true</code> .
<code>GIT_USER</code>	The username for a GitHub account that <b>has push access to the deployment repo</b> . For your own repositories, this will usually be your GitHub username. Required if not using SSH, and ignored otherwise.
<code>GIT_PASS</code>	Personal access token of the git user (specified by <code>GIT_USER</code> ), to facilitate non-interactive deployment (e.g. continuous deployment)
<code>CURRENT_BRANCH</code>	The source branch. Usually, the branch will be <code>main</code> or <code>master</code> , but it could be any branch except for <code>gh-pages</code> . If nothing is set for this variable, then the current branch from which <code>docusaurus deploy</code> is invoked will be used.
<code>GIT_USER_NAME</code>	The <code>git config user.name</code> value to use when pushing to the deployment repo
<code>GIT_USER_EMAIL</code>	The <code>git config user.email</code> value to use when pushing to the deployment repo

GitHub enterprise installations should work in the same manner as `github.com`; you only need to set the organization's GitHub Enterprise host as an environment variable:

Name	Description
GITHUB_HOST	The domain name of your GitHub enterprise site.
GITHUB_PORT	The port of your GitHub enterprise site.

## Deploy

Finally, to deploy your site to GitHub Pages, run:

Bash   Windows   PowerShell

```
GIT_USER=<GITHUB_USERNAME> yarn deploy
```

### ⚠️ WARNING

Beginning in August 2021, GitHub requires every command-line sign-in to use the **personal access token** instead of the password. When GitHub prompts for your password, enter the PAT instead. See the [GitHub documentation](#) for more information.

Alternatively, you can use SSH (`USE_SSH=true`) to log in.

## Triggering deployment with GitHub Actions

[GitHub Actions](#) allow you to automate, customize, and execute your software development workflows right in your repository.

The workflow examples below assume your website source resides in the `main` branch of your repository (the *source branch* is `main`), and your [publishing source](#) is configured for [publishing with a custom GitHub Actions Workflow](#).

Our goal is that:

1. When a new pull request is made to `main`, there's an action that ensures the site builds successfully, without actually deploying. This job will be called `test-deploy`.
2. When a pull request is merged to the `main` branch or someone pushes to the `main` branch directly, it will be built and deployed to GitHub Pages. This job will be called `deploy`.

Here are two approaches to deploying your docs with GitHub Actions. Based on the location of your deployment repository, choose the relevant tab below:

- Source repo and deployment repo are the **same** repository.
- The deployment repo is a **remote** repository, different from the source. Instructions for this scenario assume **publishing source** is the `gh-pages` branch.

**Same**    **Remote**

While you can have both jobs defined in the same workflow file, the original `deploy` workflow will always be listed as skipped in the PR check suite status, which is not indicative of the actual status and provides no value to the review process. We therefore propose to manage them as separate workflows instead.

▼ GitHub action files

▼ Site not deployed properly?

## Triggering deployment with Travis CI

Continuous integration (CI) services are typically used to perform routine tasks whenever new commits are checked in to source control. These tasks can be any combination of running unit tests and integration tests, automating builds, publishing packages to npm, and deploying changes to your website. All you need to do to automate the deployment of your website is to invoke the `yarn deploy` script whenever your website is updated. The following section covers how to do just that using [Travis CI](#), a popular continuous integration service provider.

1. Go to <https://github.com/settings/tokens> and generate a new **personal access token**. When creating the token, grant it the `repo` scope so that it has the permissions it needs.
2. Using your GitHub account, [add the Travis CI app](#) to the repository you want to activate.
3. Open your Travis CI dashboard. The URL looks like <https://travis-ci.com/USERNAME/REPO>, and navigate to the `More options > Setting > Environment Variables` section of your repository.
4. Create a new environment variable named `GH_TOKEN` with your newly generated token as its value, then `GH_EMAIL` (your email address) and `GH_NAME` (your GitHub username).
5. Create a `.travis.yml` on the root of your repository with the following:

`.travis.yml`

```

language: node_js
node_js:
 - 18
branches:
 only:
 - main
cache:
 yarn: true
script:
 - git config --global user.name "${GH_NAME}"
 - git config --global user.email "${GH_EMAIL}"
 - echo "machine github.com login ${GH_NAME} password ${GH_TOKEN}" > ~/.netrc
 - yarn install
 - GIT_USER="${GH_NAME}" yarn deploy

```

Now, whenever a new commit lands in `main`, Travis CI will run your suite of tests and if everything passes, your website will be deployed via the `yarn deploy` script.

## Triggering deployment with Buddy

Buddy is an easy-to-use CI/CD tool that allows you to automate the deployment of your portal to different environments, including GitHub Pages.

Follow these steps to create a pipeline that automatically deploys a new version of your website whenever you push changes to the selected branch of your project:

1. Go to <https://github.com/settings/tokens> and generate a new [personal access token](#). When creating the token, grant it the `repo` scope so that it has the permissions it needs.
2. Sign in to your Buddy account and create a new project.
3. Choose GitHub as your git hosting provider and select the repository with the code of your website.
4. Using the left navigation panel, switch to the `Pipelines` view.
5. Create a new pipeline. Define its name, set the trigger mode to `On push`, and select the branch that triggers the pipeline execution.
6. Add a `Node.js` action.
7. Add these commands in the action's terminal:

```

GIT_USER=<GH_PERSONAL_ACCESS_TOKEN>
git config --global user.email "<YOUR_GH_EMAIL>"
git config --global user.name "<YOUR_GH_USERNAME>"
yarn deploy

```

After creating this simple pipeline, each new commit pushed to the branch you selected deploys your website to GitHub Pages using `yarn deploy`. Read [this guide](#) to learn more about setting up a CI/CD pipeline for Docusaurus.

## Using Azure Pipelines

1. Sign Up at [Azure Pipelines](#) if you haven't already.
2. Create an organization. Within the organization, create a project and connect your repository from GitHub.
3. Go to <https://github.com/settings/tokens> and generate a new [personal access token](#) with the `repo` scope.
4. In the project page (which looks like [https://dev.azure.com/ORG\\_NAME/REPO\\_NAME/\\_build](https://dev.azure.com/ORG_NAME/REPO_NAME/_build)), create a new pipeline with the following text. Also, click on edit and add a new environment variable named `GH_TOKEN` with your newly generated token as its value, then `GH_EMAIL` (your email address) and `GH_NAME` (your GitHub username). Make sure to mark them as secret. Alternatively, you can also add a file named `azure-pipelines.yml` at your repository root.

```
azure-pipelines.yml
```

```
trigger:
- main

pool:
 vmImage: ubuntu-latest

steps:
- checkout: self
 persistCredentials: true

- task: NodeTool@0
 inputs:
 versionSpec: '18'
 displayName: Install Node.js

- script: |
 git config --global user.name "${GH_NAME}"
 git config --global user.email "${GH_EMAIL}"
 git checkout -b main
 echo "machine github.com login ${GH_NAME} password ${GH_TOKEN}" > ~/.netrc
 yarn install
 GIT_USER="${GH_NAME}" yarn deploy
env:
 GH_NAME: $(GH_NAME)
 GH_EMAIL: $(GH_EMAIL)
```

```
GH_TOKEN: $(GH_TOKEN)
displayName: Install and build
```

## Using Drone

1. Create a new SSH key that will be the [deploy key](#) for your project.
2. Name your private and public keys to be specific and so that it does not overwrite your other [SSH keys](#).
3. Go to <https://github.com/USERNAME/REPO/settings/keys> and add a new deploy key by pasting in the public key you just generated.
4. Open your Drone.io dashboard and log in. The URL looks like <https://cloud.drone.io/USERNAME/REPO>.
5. Click on the repository, click on activate repository, and add a secret called [git\\_deploy\\_private\\_key](#) with your private key value that you just generated.
6. Create a [.drone.yml](#) on the root of your repository with the below text.

.drone.yml

```
kind: pipeline
type: docker
trigger:
 event:
 - tag
- name: Website
 image: node
 commands:
 - mkdir -p $HOME/.ssh
 - ssh-keyscan -t rsa github.com >> $HOME/.ssh/known_hosts
 - echo "$GITHUB_PRIVATE_KEY" > "$HOME/.ssh/id_rsa"
 - chmod 0600 $HOME/.ssh/id_rsa
 - cd website
 - yarn install
 - yarn deploy
environment:
 USE_SSH: true
 GITHUB_PRIVATE_KEY:
 from_secret: git_deploy_private_key
```

Now, whenever you push a new tag to GitHub, this trigger will start the drone CI job to publish your website.

## Deploying to Flightcontrol

[Flightcontrol](#) is a service that automatically builds and deploys your web apps to AWS Fargate directly from your Git repository. It gives you full access to inspect and make infrastructure changes without the limitations of a traditional PaaS.

Get started by following [Flightcontrol's step-by-step Docusaurus guide](#).

## Deploying to Koyeb

[Koyeb](#) is a developer-friendly serverless platform to deploy apps globally. The platform lets you seamlessly run Docker containers, web apps, and APIs with git-based deployment, native autoscaling, a global edge network, and built-in service mesh and discovery. Check out the [Koyeb's Docusaurus deployment guide](#) to get started.

## Deploying to Render

[Render](#) offers [free static site hosting](#) with fully managed SSL, custom domains, a global CDN, and continuous auto-deploy from your Git repo. Get started in just a few minutes by following [Render's guide to deploying Docusaurus](#).

## Deploying to Qovery

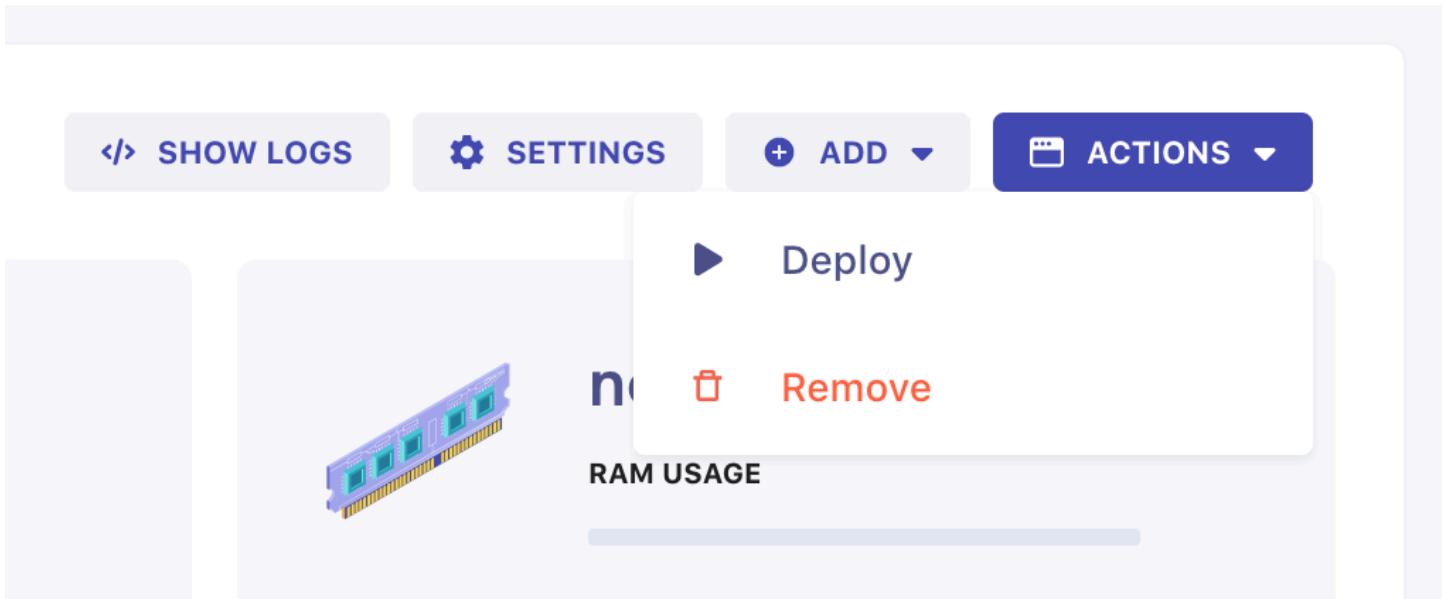
[Qovery](#) is a fully-managed cloud platform that runs on your AWS, Digital Ocean, and Scaleway account where you can host static sites, backend APIs, databases, cron jobs, and all your other apps in one place.

1. Create a Qovery account. Visit the [Qovery dashboard](#) to create an account if you don't already have one.
2. Create a project.
  - Click on **Create project** and give a name to your project.
  - Click on **Next**.
3. Create a new environment.
  - Click on **Create environment** and give a name (e.g. staging, production).
4. Add an application.
  - Click on **Create an application**, give a name and select your GitHub or GitLab repository where your Docusaurus app is located.
  - Define the main branch name and the root application path.
  - Click on **Create**. After the application is created:

- Navigate to your application **Settings**
- Select **Port**
- Add port used by your Docusaurus application

## 5. Deploy

- All you have to do now is to navigate to your application and click on **Deploy**.



That's it. Watch the status and wait till the app is deployed. To open the application in your browser, click on **Action** and **Open** in your application overview.

## Deploying to Hostman

[Hostman](#) allows you to host static websites for free. Hostman automates everything, you just need to connect your repository and follow these easy steps:

### 1. Create a service.

- To deploy a Docusaurus static website, click **Create** in the top-left corner of your [Dashboard](#) and choose **Front-end app or static website**.

### 2. Select the project to deploy.

- If you are logged in to Hostman with your GitHub, GitLab, or Bitbucket account, you will see the repository with your projects, including the private ones.
- Choose the project you want to deploy. It must contain the directory with the project's files (e.g. [website](#)).
- To access a different repository, click **Connect another repository**.

- If you didn't use your Git account credentials to log in, you'll be able to access the necessary account now, and then select the project.

### 3. Configure the build settings.

- Next, the **Website customization** window will appear. Choose the **Static website** option from the list of frameworks.
- The **Directory with app** points at the directory that will contain the project's files after the build. If you selected the repository with the contents of the website (or `my_website`) directory during Step 2, you can leave it empty.
- The standard build command for Docusaurus is:

**npm**    **Yarn**    **pnpm**    **Bun**

---

```
npm run build
```

- You can modify the build command if needed. You can enter multiple commands separated by `&&`.

### 4. Deploy.

- Click **Deploy** to start the build process.
- Once it starts, you will enter the deployment log. If there are any issues with the code, you will get warning or error messages in the log specifying the cause of the problem. Usually, the log contains all the debugging data you'll need.
- When the deployment is complete, you will receive an email notification and also see a log entry. All done! Your project is up and ready.

## Deploying to Surge

Surge is a [static web hosting platform](#) that you can use to deploy your Docusaurus project from the command line in seconds. Deploying your project to Surge is easy and free (including custom domains and SSL certs).

Deploy your app in a matter of seconds using Surge with the following steps:

1. First, install Surge using npm by running the following command:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm install -g surge
```

2. To build the static files of your site for production in the root directory of your project, run:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run build
```

3. Then, run this command inside the root directory of your project:

```
surge build/
```

First-time users of Surge would be prompted to create an account from the command line (which happens only once).

Confirm that the site you want to publish is in the `build` directory. A randomly generated subdomain `*.surge.sh subdomain` is always given (which can be edited).

## Using your domain

If you have a domain name you can deploy your site using the command:

```
surge build/ your-domain.com
```

Your site is now deployed for free at `subdomain.surge.sh` or `your-domain.com` depending on the method you chose.

## Setting up CNAME file

Store your domain in a CNAME file for future deployments with the following command:

```
echo subdomain.surge.sh > CNAME
```

You can deploy any other changes in the future with the command `surge`.

# Deploying to Stormkit

You can deploy your Docusaurus project to [Stormkit](#), a deployment platform for static websites, single-page applications (SPAs), and serverless functions. For detailed instructions, refer to this [guide](#).

# Deploying to QuantCDN

1. Install [Quant CLI](#)
2. Create a QuantCDN account by [signing up](#)
3. Initialize your project with `quant init` and fill in your credentials:

```
quant init
```

4. Deploy your site.

```
quant deploy
```

See [docs](#) and [blog](#) for more examples and use cases for deploying to QuantCDN.

# Deploying to Cloudflare Pages

[Cloudflare Pages](#) is a Jamstack platform for frontend developers to collaborate and deploy websites. Get started within a few minutes by following [this page](#).

# Deploying to Azure Static Web Apps

[Azure Static Web Apps](#) is a service that automatically builds and deploys full-stack web apps to Azure directly from the code repository, simplifying the developer experience for CI/CD. Static Web Apps separates the web application's static assets from its dynamic (API) endpoints. Static assets are served from globally-distributed content servers, making it faster for clients to retrieve files using servers nearby. Dynamic APIs are scaled with serverless architectures using an event-driven functions-based approach that is more cost-effective and scales on demand. Get started in a few minutes by following [this step-by-step guide](#).

# Deploying to Kinsta

[Kinsta Static Site Hosting](#) lets you deploy up to 100 static sites for free, custom domains with SSL, 100 GB monthly bandwidth, and 260+ Cloudflare CDN locations.

Get started in just a few clicks by following our [Docusaurus on Kinsta](#) article.

# i18n - Tutorial

This tutorial will walk you through the basics of the **Docusaurus i18n system**.

We will add **French** translations to a **newly initialized English Docusaurus website**.

Initialize a new site with `npx create-docusaurus@latest website classic` (like [this one](#)).

## Configure your site

Modify `docusaurus.config.js` to add the i18n support for the French language.

### Site configuration

Use the [site i18n configuration](#) to declare the i18n locales:

`docusaurus.config.js`

```
export default {
 i18n: {
 defaultLocale: 'en',
 locales: ['en', 'fr', 'fa'],
 localeConfigs: {
 en: {
 htmlLang: 'en-GB',
 },
 // You can omit a locale (e.g. fr) if you don't need to override the
 // defaults
 fa: {
 direction: 'rtl',
 },
 },
 },
};
```

The locale names are used for the translation files' locations, as well as your translated locales' base URL. When building all locales, only the default locale will have its name omitted in the base URL.

Docusaurus uses the locale names to provide **sensible defaults**: the `<html lang="...>` attribute, locale label, calendar format, etc. You can customize these defaults with the `localeConfigs`.

# Theme configuration

Add a **navbar item** of type `localeDropdown` so that users can select the locale they want:

`docusaurus.config.js`

```
export default {
 themeConfig: {
 navbar: {
 items: [
 {
 type: 'localeDropdown',
 position: 'left',
 },
],
 },
 },
};
```



TIP

You can pass a query parameter that will be appended to the URL when a user changes the locale using the dropdown (e.g. `queryString: '?persistLocale=true'`).

This is useful for implementing an automatic locale detection on your server. For example, you can use this parameter to store the user's preferred locale in a cookie.

## Start your site

Start your localized site in dev mode, using the locale of your choice:

`npm`   `Yarn`   `pnpm`   `Bun`

```
npm run start -- --locale fr
```

Your site is accessible at <http://localhost:3000/fr/>.

We haven't provided any translation yet, so the site is mostly untranslated.



TIP

Docusaurus provides **default translations** for generic theme labels, such as "Next" and "Previous" for the pagination.

Please help us complete those [default translations](#).

### WARNING

Each locale is a **distinct standalone single-page application**: it is not possible to start the Docusaurus sites in all locales at the same time.

## Translate your site

All translation data for the French locale is stored in `website/i18n/fr`. Each plugin sources its own translated content under the corresponding folder, while the `code.json` file defines all text labels used in the React code.

### NOTE

After copying files around, restart your site with `npm run start -- --locale fr`. Hot-reload will work better when editing existing files.

## Translate your React code

For any React code you've written yourself: React pages, React components, etc., you will use the [translation APIs](#).

Locate all text labels in your React code that will be visible to your users, and mark them with the translation APIs. There are two kinds of APIs:

- The `<Translate>` component wraps a string as a JSX element;
- The `translate()` callback takes a message and returns a string.

Use the one that better fits the context semantically. For example, the `<Translate>` can be used as React children, while for props that expect a string, the callback can be used.

### WARNING

A JSX element is an *object*, not a string. Using it in contexts expecting strings (such as the children of `<option>`) would coerce it to a string, which returns `"[object Object]"`. While we encourage you to use `<Translate>` as JSX children, only use the element form when it actually works.

```
src/pages/index.js
```

```
import React from 'react';
import Layout from '@theme/Layout';
import Link from '@docusaurus/Link';

export default function Home() {
 return (
 <Layout>
 <h1>Welcome to my website</h1>
 <main>
 You can also visit my
 <Link to="https://docusaurus.io/blog">blog</Link>

 </main>
 </Layout>
);
}
```

### ⚠ INFO

Docusaurus provides a **very small and lightweight translation runtime** on purpose, and only supports basic [placeholders interpolation](#), using a subset of the [ICU Message Format](#).

Most documentation websites are generally **static** and don't need advanced i18n features ([plurals](#), [genders](#), etc.). Use a library like [react-intl](#) for more advanced use-cases.

The `docusaurus write-translations` command will statically analyze all React code files used in your site, extract calls to these APIs, and aggregate them in the `code.json` file. The translation files will be stored as maps from IDs to translation message objects (including the translated label and the description of the label). In your calls to the translation APIs (`<Translate>` or `translate()`), you need to specify either the default untranslated message or the ID, in order for Docusaurus to correctly correlate each translation entry to the API call.

### ⚠ TEXT LABELS MUST BE STATIC

The `docusaurus write-translations` command only does **static analysis** of your code. It doesn't actually run your site. Therefore, dynamic messages can't be extracted, as the message is an *expression*, not a *string*:

```

const items = [
 {id: 1, title: 'Hello'},
 {id: 2, title: 'World'},
];

function ItemsList() {
 return (

 {/* DON'T DO THIS: doesn't work with the write-translations command */}
 {items.map((item) => (
 <li key={item.id}>
 <Translate>{item.title}</Translate>

))}

);
}

```

This still behaves correctly at runtime. However, in the future, we may provide a "no-runtime" mechanism, allowing the translations to be directly inlined in the React code through Babel transformations, instead of calling the APIs at runtime. Therefore, to be future-proof, you should always prefer statically analyzable messages. For example, we can refactor the code above to:

```

const items = [
 {id: 1, title: <Translate>Hello</Translate>},
 {id: 2, title: <Translate>World</Translate>},
];

function ItemsList() {
 return (

 {/* The titles are now already translated when rendering! */}
 {items.map((item) => (
 <li key={item.id}>{item.title}
))}

);
}

```

You can see the calls to the translation APIs as purely *markers* that tell Docusaurus that "here's a text label to be replaced with a translated message".

## Pluralization

When you run `write-translations`, you will notice that some labels are pluralized:

```
{
 // ...
 "theme.blog.post.plurals": "One post|{count} posts"
 // ...
}
```

Every language will have a list of [possible plural categories](#). Docusaurus will arrange them in the order of `["zero", "one", "two", "few", "many", "other"]`. For example, because English (`en`) has two plural forms ("one" and "other"), the translation message has two labels separated by a pipe (`|`). For Polish (`pl`) which has three plural forms ("one", "few", and "many"), you would provide three labels in that order, joined by pipes.

You can pluralize your own code's messages as well:

```
import {translate} from '@docusaurus/Translate';
import {usePluralForm} from '@docusaurus/theme-common';

function ItemsList({items}) {
 // `usePluralForm` will provide the plural selector for the current locale
 const {selectMessage} = usePluralForm();
 // Select the appropriate pluralized label based on `items.length`
 const message = selectMessage(
 items.length,
 translate(
 {message: 'One item|{count} items'},
 {count: items.length},
),
);
 return (
 <>
 <h2>{message}</h2>
 {items.map((item) => <li key={item.id}>{item.title})}
 </>
);
}
```

### NOTE

Docusaurus uses `Intl.PluralRules` to resolve and select plural forms. It is important to provide the right number of plural forms in the right order for `selectMessage` to work.

## Translate plugin data

JSON translation files are used for everything that is interspersed in your code:

- React code, including the translated labels you have marked above
- Navbar and footer labels in theme config
- Docs sidebar category labels in `sidebars.js`
- Blog sidebar title in plugin options
- ...

Run the `write-translations` command:

**npm**    **Yarn**    **pnpm**    **Bun**

---

```
npm run write-translations -- --locale fr
```

It will extract and initialize the JSON translation files that you need to translate. The `code.json` file at the root includes all translation API calls extracted from the source code, which could either be written by you or provided by the themes, some of which may already be translated by default.

i18n/fr/code.json

```
{
 // No ID for the <Translate> component: the default message is used as ID
 "Welcome to my website": {
 "message": "Welcome to my website"
 },
 "home.visitMyBlog": {
 "message": "You can also visit my {blog}",
 "description": "The homepage message to ask the user to visit my blog"
 },
 "homepage.visitMyBlog.linkLabel": {
 "message": "Blog",
 "description": "The label for the link to my blog"
 },
 "Home icon": {
 "message": "Home icon",
 "description": "The homepage icon alt message"
 }
}
```

Plugins and themes will also write their own JSON translation files, such as:

```
{
 "title": {
 "message": "My Site",
 "description": "The title in the navbar"
 },
 "item.label.Docs": {
 "message": "Docs",
 "description": "Navbar item with label Docs"
 },
 "item.label.Blog": {
 "message": "Blog",
 "description": "Navbar item with label Blog"
 },
 "item.label.GitHub": {
 "message": "GitHub",
 "description": "Navbar item with label GitHub"
 }
}
```

Translate the `message` attribute in the JSON files of `i18n/fr`, and your site layout and homepage should now be translated.

## Translate Markdown files

Official Docusaurus content plugins extensively use Markdown/MDX files and allow you to translate them.

### Translate the docs

Copy your docs Markdown files from `docs/` to `i18n/fr/docusaurus-plugin-content-docs/current`, and translate them:

```
mkdir -p i18n/fr/docusaurus-plugin-content-docs/current
cp -r docs/** i18n/fr/docusaurus-plugin-content-docs/current
```

#### (!) INFO

Notice that the `docusaurus-plugin-content-docs` plugin always divides its content by versions. The data in `./docs` folder will be translated in the `current` subfolder and `current.json` file. See [the doc versioning guide](#) for more information about what "current" means.

### Translate the blog

Copy your blog Markdown files to `i18n/fr/docusaurus-plugin-content-blog`, and translate them:

```
mkdir -p i18n/fr/docusaurus-plugin-content-blog
cp -r blog/** i18n/fr/docusaurus-plugin-content-blog
```

## Translate the pages

Copy your pages Markdown files to `i18n/fr/docusaurus-plugin-content-pages`, and translate them:

```
mkdir -p i18n/fr/docusaurus-plugin-content-pages
cp -r src/pages/**.md i18n/fr/docusaurus-plugin-content-pages
cp -r src/pages/**.mdx i18n/fr/docusaurus-plugin-content-pages
```

### WARNING

We only copy `.md` and `.mdx` files, as React pages are translated through JSON translation files already.

### USE EXPLICIT HEADING IDs

By default, a Markdown heading `### Hello World` will have a generated ID `hello-world`. Other documents can link it with `[link](#hello-world)`. However, after translation, the heading becomes `### Bonjour le Monde`, with ID `bonjour-le-monde`.

Generated IDs are not always a good fit for localized sites, as it requires you to localize all the anchor links:

- `[link](#hello-world)`.
- + `[link](#bonjour-le-monde)`

For localized sites, it is recommended to use [explicit heading IDs](#).

# Deploy your site

You can choose to deploy your site under a **single domain** or use **multiple (sub)domains**.

## Single-domain deployment

Run the following command:

```
npm run build
```

Docusaurus will build **one single-page application per locale**:

- `website/build`: for the default, English language
- `website/build/fr`: for the French language

You can now [deploy](#) the `build` folder to the static hosting solution of your choice.

 **NOTE**

The Docusaurus website uses this strategy:

- `https://docusaurus.io`
- `https://docusaurus.io/fr`

 **TIP**

Static hosting providers generally redirect `/unknown/url` to `/404.html` by convention, always showing an **English 404 page**.

**Localize your 404 pages** by configuring your host to redirect `/fr/*` to `/fr/404.html`.

This is not always possible, and depends on your host: GitHub Pages can't do this, [Netlify](#) can.

## Multi-domain deployment

You can also build your site for a single locale:

```
npm run build -- --locale fr
```

Docusaurus will not add the `/fr/` URL prefix.

On your [static hosting provider](#):

- create one deployment per locale

- configure the appropriate build command, using the `--locale` option
- configure the (sub)domain of your choice for each deployment

### WARNING

This strategy is **not possible** with GitHub Pages, as it is only possible to **have a single deployment**.

## Hybrid

It is possible to have some locales using sub-paths, and others using subdomains.

It is also possible to deploy each locale as a separate subdomain, assemble the subdomains in a single unified domain at the CDN level:

- Deploy your site as `fr.docusaurus.io`
- Configure a CDN to serve it from `docusaurus.io/fr`

## Managing translations

Docusaurus doesn't care about how you manage your translations: all it needs is that all translation files (JSON, Markdown, or other data files) are available in the file system during building. However, as site creators, you would need to consider how translations are managed so your translation contributors could collaborate well.

We will share two common translation collaboration strategies: [using git](#) and [using Crowdin](#).

# i18n - Using git

A possible translation strategy is to **version control the translation files** with Git (or any other [VCS](#)).

## Tradeoffs

This strategy has advantages:

- **Easy to get started:** just commit the `i18n` folder to Git
- **Easy for developers:** Git, GitHub and pull requests are mainstream developer tools
- **Free** (or without any additional cost, assuming you already use Git)
- **Low friction:** does not require signing up to an external tool
- **Rewarding:** contributors are happy to have a nice contribution history

Using Git also present some shortcomings:

- **Hard for non-developers:** they do not master Git and pull-requests
- **Hard for professional translators:** they are used to SaaS translation software and advanced features
- **Hard to maintain:** you have to keep the translated files **in sync** with the untranslated files

### NOTE

Some **large-scale technical projects** (React, Vue.js, MDN, TypeScript, Nuxt.js, etc.) use Git for translations.

Refer to the [Docusaurus i18n RFC](#) for our notes and links studying these systems.

## Initialization

This is a walk-through of using Git to translate a newly initialized English Docusaurus website into French, and assume you already followed the [i18n tutorial](#).

### Prepare the Docusaurus site

Initialize a new Docusaurus site:

```
npx create-docusaurus@latest website classic
```

Add the site configuration for the French language:

docusaurus.config.js

```
export default {
 i18n: {
 defaultLocale: 'en',
 locales: ['en', 'fr'],
 },
 themeConfig: {
 navbar: {
 items: [
 // ...
 {
 type: 'localeDropdown',
 position: 'left',
 },
 // ...
],
 },
 // ...
 },
};
```

Translate the homepage:

src/pages/index.js

```
import React from 'react';
import Translate from '@docusaurus/Translate';
import Layout from '@theme/Layout';

export default function Home() {
 return (
 <Layout>
 <h1 style={{margin: 20}}>
 <Translate description="The homepage main heading">
 Welcome to my Docusaurus translated site!
 </Translate>
 </h1>
 </Layout>
);
}
```

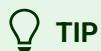
# Initialize the i18n folder

Use the [write-translations](#) CLI command to initialize the JSON translation files for the French locale:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run write-translations -- --locale fr

1 translations written at i18n/fr/code.json
11 translations written at i18n/fr/docusaurus-theme-classic/footer.json
4 translations written at i18n/fr/docusaurus-theme-classic/navbar.json
3 translations written at i18n/fr/docusaurus-plugin-content-docs/current.json
```



Use the `--messagePrefix '(fr) '` option to make the untranslated strings stand out.

`Hello` will appear as `(fr) Hello` and makes it clear a translation is missing.

Copy your untranslated Markdown files to the French folder:

```
mkdir -p i18n/fr/docusaurus-plugin-content-docs/current
cp -r docs/** i18n/fr/docusaurus-plugin-content-docs/current

mkdir -p i18n/fr/docusaurus-plugin-content-blog
cp -r blog/** i18n/fr/docusaurus-plugin-content-blog

mkdir -p i18n/fr/docusaurus-plugin-content-pages
cp -r src/pages/**.md i18n/fr/docusaurus-plugin-content-pages
cp -r src/pages/**.mdx i18n/fr/docusaurus-plugin-content-pages
```

Add all these files to Git.

## Translate the files

Translate the Markdown and JSON files in `i18n/fr` and commit the translation.

You should now be able to start your site in French and see the translations:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run start -- --locale fr
```

You can also build the site locally or on your CI:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run build
or
npm run build -- --locale fr
```

## Repeat

Follow the same process for each locale you need to support.

# Maintenance

Keeping translated files **consistent** with the originals **can be challenging**, in particular for Markdown documents.

## Markdown translations

When an untranslated Markdown document is edited, it is **your responsibility to maintain the respective translated files**, and we unfortunately don't have a good way to help you do so.

To keep your translated sites consistent, when the `website/docs/doc1.md` doc is edited, you need **backport these edits** to `i18n/fr/docusaurus-plugin-content-docs/current/doc1.md`.

## JSON translations

To help you maintain the JSON translation files, it is possible to run again the `write-translations` CLI command:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run write-translations -- --locale fr
```

New translations will be appended, and existing ones will not be overridden.



**TIP**

Reset your translations with the `--override` option.

## Localize edit URLs

When the user is browsing a page at `/fr/doc1`, the edit button will link by default to the unlocalized doc at `website/docs/doc1.md`.

Your translations are on Git, and you can use the `editLocalizedFiles: true` option of the docs and blog plugins.

The edit button will link to the localized doc at `i18n/fr/docusaurus-plugin-content-docs/current/doc1.md`.

# i18n - Using Crowdin

The i18n system of Docusaurus is **decoupled from any translation software**.

You can integrate Docusaurus with the **tools and SaaS of your choice**, as long as you put the **translation files at the correct location**.

We document the usage of [Crowdin](#), as **one possible integration example**.

## WARNING

This is **not an endorsement of Crowdin** as the unique choice to translate a Docusaurus site, but it is successfully used by Facebook to translate documentation projects such as [Jest](#), [Docusaurus](#), and [ReasonML](#).

Refer to the [Crowdin documentation](#) and [Crowdin support](#) for help.

## TIP

Use this [community-driven GitHub discussion](#) to discuss anything related to Docusaurus + Crowdin.

## Crowdin overview

Crowdin is a translation SaaS, offering a [free plan for open-source projects](#).

We recommend the following translation workflow:

- **Upload sources** to Crowdin (untranslated files)
- Use Crowdin to **translate the content**
- **Download translations** from Crowdin (localized translation files)

Crowdin provides a [CLI](#) to **upload sources** and **download translations**, allowing you to automate the translation process.

The [crowdin.yml](#) configuration file is convenient for Docusaurus, and permits to **download the localized translation files at the expected location** (in `i18n/[locale]/..`).

Read the [official documentation](#) to know more about advanced features and different translation workflows.

# Crowdin tutorial

This is a walk-through of using Crowdin to translate a newly initialized English Docusaurus website into French, and assume you already followed the [i18n tutorial](#).

The end result can be seen at [docusaurus-crowdin-example.netlify.app \(repository\)](https://docusaurus-crowdin-example.netlify.app).

## Prepare the Docusaurus site

Initialize a new Docusaurus site:

```
npx create-docusaurus@latest website classic
```

Add the site configuration for the French language:

docusaurus.config.js

```
export default {
 i18n: {
 defaultLocale: 'en',
 locales: ['en', 'fr'],
 },
 themeConfig: {
 navbar: {
 items: [
 // ...
 {
 type: 'localeDropdown',
 position: 'left',
 },
 // ...
],
 },
 // ...
 },
 // ...
};
```

Translate the homepage:

src/pages/index.js

```
import React from 'react';
import Translate from '@docusaurus/Translate';
```

```
import Layout from '@theme/Layout';

export default function Home() {
 return (
 <Layout>
 <h1 style={{margin: 20}}>
 <Translate description="The homepage main heading">
 Welcome to my Docusaurus translated site!
 </Translate>
 </h1>
 </Layout>
);
}
```

## Create a Crowdin project

Sign up on [Crowdin](#), and create a project.

Use English as the source language, and French as the target language.

# Create Crowdin Project

Project name

docusaurus-crowdin-example

Project address

<https://crowdin.com/project/docusaurus-crowdin-example>

Public project

Visible to anyone. You can restrict access to specific languages after the project is created.

Private project

Visible only to the invited project members.

Source language

English

Target languages

fre

- French
- French, Belgium
- French, Canada
- French, Luxembourg
- French, Quebec
- French, Switzerland

1 language selected

French

trash bin icon

Your project is created, but it is empty for now. We will upload the files to translate in the next steps.

## Create the Crowdin configuration

This configuration ([doc](#)) provides a mapping for the Crowdin CLI to understand:

- Where to find the source files to upload (JSON and Markdown)
- Where to download the files after translation (in `i18n/[locale]`)

Create `crowdin.yml` in `website`:

`crowdin.yml`

```
project_id: '123456'
api_token_env: CROWDIN_PERSONAL_TOKEN
preserve_hierarchy: true
```

```
files:
 # JSON translation files
 - source: /i18n/en/**/*
 translation: /i18n/%two_letters_code%/**/%original_file_name%
 # Docs Markdown files
 - source: /docs/**/*
 translation: /i18n/%two_letters_code%/docusaurus-plugin-content-
docs/current/**/%original_file_name%
 # Blog Markdown files
 - source: /blog/**/*
 translation: /i18n/%two_letters_code%/docusaurus-plugin-content-
blog/**/%original_file_name%
```

Crowdin has its own syntax for declaring source/translation paths:

- `**/*`: everything in a subfolder
- `%two_letters_code%`: the 2-letters variant of Crowdin target languages (`fr` in our case)
- `**/%original_file_name%`: the translations will preserve the original folder/file hierarchy

### ! INFO

The Crowdin CLI warnings are not always easy to understand.

We advise to:

- change one thing at a time
- re-upload sources after any configuration change
- use paths starting with `/` (`./` does not work)
- avoid fancy globbing patterns like `/docs/**/*.(md|mdx)` (does not work)

## Access token

The `api_token_env` attribute defines the **env variable name** read by the Crowdin CLI.

You can obtain a `Personal Access Token` on [your personal profile page](#).

### 💡 TIP

You can keep the default value `CROWDIN_PERSONAL_TOKEN`, and set this environment variable and on your computer and on the CI server to the generated access token.

### ⚠️ WARNING

A Personal Access Tokens grant **read-write access to all your Crowdin projects**.

You should **not commit** it, and it may be a good idea to create a dedicated **Crowdin profile for your company** instead of using a personal account.

## Other configuration fields

- `project_id`: can be hardcoded, and is found on  
[https://crowdin.com/project/<MY\\_PROJECT\\_NAME>/settings#api](https://crowdin.com/project/<MY_PROJECT_NAME>/settings#api)
- `preserve_hierarchy`: preserve the folder's hierarchy of your docs on Crowdin UI instead of flattening everything

## Install the Crowdin CLI

This tutorial uses the CLI version `3.5.2`, but we expect `3.x` releases to keep working.

Install the Crowdin CLI as an npm package to your Docusaurus site:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm install @crowdin/cli@3
```

Add a `crowdin` script:

`package.json`

```
{
 "scripts": {
 // ...
 "write-translations": "docusaurus write-translations",
 "crowdin": "crowdin"
 }
}
```

Test that you can run the Crowdin CLI:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run crowdin -- --version
```

Set the `CROWDIN_PERSONAL_TOKEN` env variable on your computer, to allow the CLI to authenticate with the Crowdin API.

 **TIP**

Temporarily, you can hardcode your personal token in `crowdin.yml` with `api_token: 'MY-TOKEN'`.

## Upload the sources

Generate the JSON translation files for the default language in `website/i18n/en`:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run write-translations
```

Upload all the JSON and Markdown translation files:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

---

```
npm run crowdin upload
```

- ✓ Fetching project info
- ✓ Directory 'i18n'
- ✓ Directory 'i18n/en'
- ✓ Directory 'i18n/en/docusaurus-theme-classic'
- ✓ Directory 'i18n/en/docusaurus-plugin-content-docs'
- ✓ File 'i18n/en/docusaurus-theme-classic/footer.json'
- ✓ File 'i18n/en/docusaurus-theme-classic/navbar.json'
- ✓ File 'i18n/en/docusaurus-plugin-content-docs/current.json'
- ✓ Directory 'docs'
- ✓ File 'docs/doc3.md'
- ✓ File 'docs/mdx.md'
- ✓ File 'docs/doc2.md'
- ✓ File 'docs/doc1.md'
- ✓ Directory 'blog'
- ✓ File 'blog/2019-05-30-welcome.md'
- ✓ File 'blog/2019-05-29-hello-world.md'
- ✓ File 'blog/2019-05-28-hola.md'

Your source files are now visible on the Crowdin interface:

[https://crowdin.com/project/<MY\\_PROJECT\\_NAME>/settings#files](https://crowdin.com/project/<MY_PROJECT_NAME>/settings#files)

The screenshot shows the 'Files' tab in the Crowdin project settings. The interface includes a top navigation bar with tabs for General, Translations, Files, Members, Vendors, Reports, Screenshots, TM, Glossary, Integrations, API & Webhooks, Strings, and More. Below the navigation is a toolbar with buttons for 'Add File', 'New Folder', 'New Version Branch', and a trash bin icon. A search bar labeled 'Search files' is also present. The main content area displays a table of files and folders. The table has columns for Name, Strings, and Revision. The 'Name' column lists the folder structure: 'blog', 'docs', and 'i18n'. The 'Strings' column shows the count of strings for each: 38 for blog, 166 for docs, and 18 for i18n. The 'Revision' column shows the revision number for each. On the far right of each row are three dots and a small up arrow icon.

Name	Strings	Revision
► blog	38	↑ ...
► docs	166	↑ ...
► i18n	18	↑ ...

## Translate the sources

On [https://crowdin.com/project/<MY\\_PROJECT\\_NAME>](https://crowdin.com/project/<MY_PROJECT_NAME>), click on the French target language.

# French translation

The screenshot shows a Crowdin project interface for the 'docusaurus-crowdin-example' repository. The file tree includes:

- blog** folder:
  - 2019-05-28-hola.md: 0% • 0%
  - 2019-05-29-hello-world.md: 0% • 0%
  - 2019-05-30-welcome.md: 0% • 0%
- docs** folder:
  - doc1.md: 0% • 0%
  - doc2.md: 0% • 0%
  - doc3.md: 0% • 0%
  - mdx.md: 0% • 0%
- i18n** folder:
  - en** folder:
    - docusaurus-plugin-content-docs** folder:
      - current.json: 0% • 0%
    - docusaurus-theme-classic** folder:
      - footer.json: 0% • 0%
      - navbar.json: 0% • 0%

At the top right, there are buttons for 'Translate All', a refresh icon, a cloud icon, and a gear icon.

Translate some Markdown files.

The screenshot shows the Crowdin editor interface for the 'DOC1.MD' file. The left panel displays the Markdown content:

```

id: doc1
title: Style Guide
sidebar_label: Style Guide
slug: /

```

The right panel shows the translation process:

**SOURCE STRING**

To serve as an example page when styling markdown based Docusaurus sites.

**CONTEXT** ▾ **EDIT**

Paragraph text  
XPath: /p[2]

**FRENCH TRANSLATIONS** ▾

Sert de page d'exemple pour styliser un site Docusaurus basé sur Markdown.

73 • 74 **SAVE**

No translations suggested yet

**TM AND MT SUGGESTIONS** ▾

No suggestions

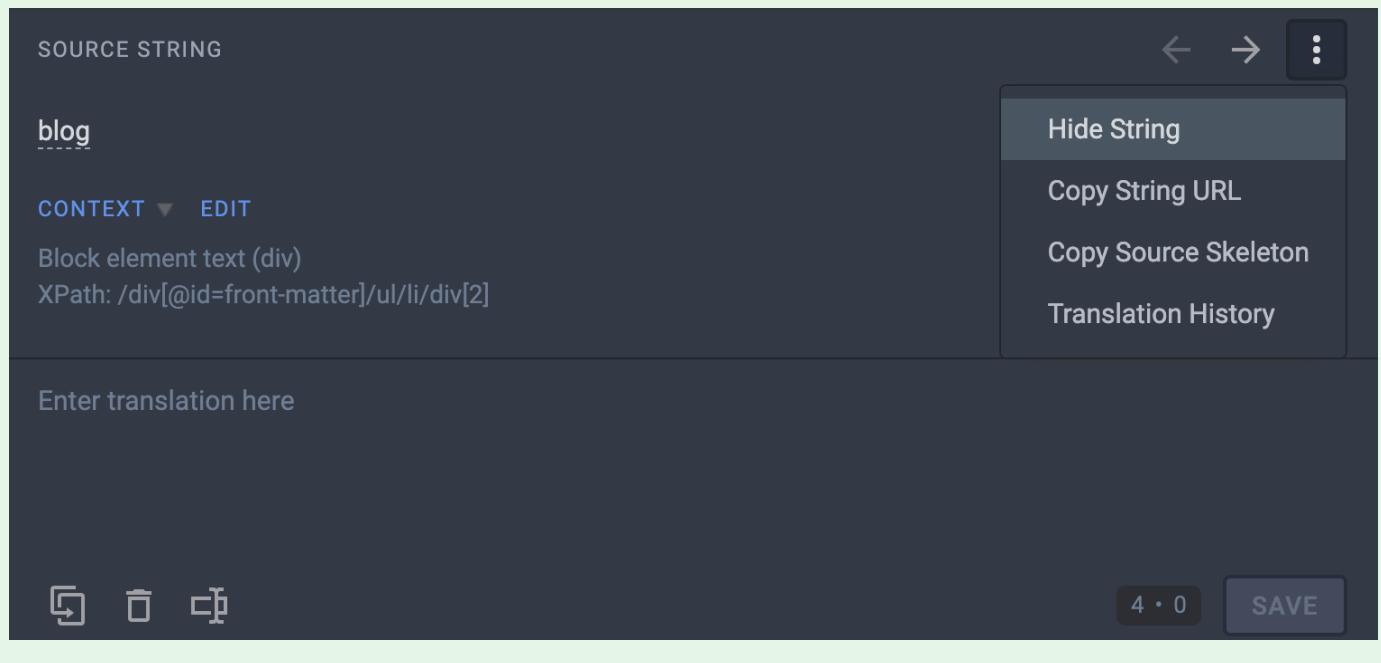
**OTHER LANGUAGES** ▾



**TIP**

Use **Hide String** to make sure translators **don't translate things that should not be**:

- Front matter: `id`, `slug`, `tags` ...
- Admonitions: `:::`, `:::note`, `:::tip` ...



Translate some JSON files.

The screenshot shows the Crowdin interface translating a JSON file named "NAVBAR.JSON". The left sidebar shows a tree structure with "My Site" selected. The main area shows a "SOURCE STRING" with the text "My Site" and its "CONTEXT" as "title". The "FRENCH TRANSLATIONS" section shows a single entry by "Sébastien Lorber" with the translation "Mon site". The "TM AND MT SUGGESTIONS" section shows "No suggestions". The "OTHER LANGUAGES" section has a link "▶". A status bar at the bottom right shows "7 · 8" and a "SAVE" button.

### !(INFO)

The `description` attribute of JSON translation files is visible on Crowdin to help translate the strings.



TIP

**Pre-translate** your site, and **fix pre-translation mistakes manually** (enable the Global Translation Memory in settings first).

Use the `Hide String` feature first, as Crowdin is pre-translating things too optimistically.

## Download the translations

Use the Crowdin CLI to download the translated JSON and Markdown files.

**npm**    **Yarn**    **pnpm**    **Bun**

```
npm run crowdin download
```

The translated content should be downloaded in `i18n/fr`.

Start your site on the French locale:

**npm**    **Yarn**    **pnpm**    **Bun**

```
npm run start -- --locale fr
```

Make sure that your website is now translated in French at <http://localhost:3000/fr>.

## Automate with CI

We will configure the CI to **download the Crowdin translations at build time** and keep them outside of Git.

Add `website/i18n` to `.gitignore`.

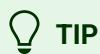
Set the `CROWDIN_PERSONAL_TOKEN` env variable on your CI.

Create an npm script to `sync` Crowdin (extract sources, upload sources, download translations):

`package.json`

```
{
 "scripts": {
 "crowdin:sync": "docusaurus write-translations && crowdin upload && crowdin
 download"
 }
}
```

Call the `npm run crowdin:sync` script in your CI, just before building the Docusaurus site.



### TIP

Keep your deploy-previews fast: don't download translations, and use `npm run build -- --locale en` for feature branches.



### WARNING

Crowdin does not support well multiple concurrent uploads/downloads: it is preferable to only include translations to your production deployment, and keep deploy previews untranslated.

## Advanced Crowdin topics

### MDX



### WARNING

Pay special attention to the JSX fragments in MDX documents!

Crowdin **does not support officially MDX**, but they added **support for the `.mdx` extension**, and interpret such files as Markdown (instead of plain text).

### MDX problems

Crowdin thinks that the JSX syntax is embedded HTML and can mess up with the JSX markup when you download the translations, leading to a site that fails to build due to invalid JSX.

Simple JSX fragments using simple string props like `<Username name="Sebastien"/>` will work fine; more complex JSX fragments using object/array props like `<User person={{name: "Sebastien"}}/>` are more likely to fail due to a syntax that does not look like HTML.

### MDX solutions

We recommend extracting the complex embedded JSX code as separate standalone components. We also added an `mdx-code-block` escape hatch syntax:

## # How to deploy Docusaurus

To deploy Docusaurus, run the following command:

```
```mdx-code-block

import Tabs from '@theme/Tabs';

import TabItem from '@theme/TabItem';

<Tabs>
  <TabItem value="bash" label="Bash">

    ```bash
 GIT_USER=<GITHUB_USERNAME> yarn deploy
    ```

  </TabItem>
  <TabItem value="windows" label="Windows">

    ```batch
 cmd /C "set "GIT_USER=<GITHUB_USERNAME>" && yarn deploy"
    ```

  </TabItem>
</Tabs>
```
```

This will:

- be interpreted by Crowdin as code blocks (and not mess-up with the markup on download)
- be interpreted by Docusaurus as regular JSX (as if it was not wrapped by any code block)
- unfortunately opt-out of MDX tooling (IDE syntax highlighting, Prettier...)

## Docs versioning

Configure translation files for the `website/versioned_docs` folder.

When creating a new version, the source strings will generally be quite similar to the current version (`website/docs`), and you don't want to translate the new version docs again and again.

Crowdin provides a `Duplicate Strings` setting.

## Translations

### Duplicate Strings

Hide – all duplicates will share the same translation



You can save time by translating all duplicates with the same translation and hiding these instances from translators. This may, however, affect accuracy.

We recommend using `Hide`, but the ideal setting depends on how much your versions are different.

#### WARNING

Not using `Hide` leads to a much larger amount of `source strings` in quotas, and will affect the pricing.

## Multi-instance plugins

You need to configure translation files for each plugin instance.

If you have a docs plugin instance with `id=ios`, you will need to configure those source files as well

- `website/ios`
- `website/ios_versioned_docs` (if versioned)

## Maintaining your site

Sometimes, you will **remove or rename a source file** on Git, and Crowdin will display CLI warnings:

```
2:53:51 PM: ✓ Building ZIP archive with the latest translations
2:53:51 PM: [●.....] Fetching project info
2:53:51 PM: [●.....] Fetching project info
2:53:51 PM: ✓ Building translation (100%)
2:53:51 PM: [●.....] Downloading translation
2:53:51 PM: ✓ Downloading translation
2:53:52 PM: ✓ Extracted: 'i18n/fr/code.json'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-blog/2019-05-28-hola.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-blog/2019-05-29-hello-world.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-blog/2019-05-30-welcome.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-docs/current.json'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-docs/current/doc1.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-docs/current/doc2.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-docs/current/doc3.md'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-plugin-content-docs/current/interactiveDoc.mdx'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-theme-classic/footer.json'
2:53:52 PM: ✓ Extracted: 'i18n/fr/docusaurus-theme-classic/navbar.json'
2:53:52 PM: △ Downloaded translations don't match the current project configuration. The translations for the following sources will be omitted (use --verbose to get the list of the omitted translations):
 - docs/doc4.md (1)
2:53:52 PM: Done in 20.84s.
```

When your sources are refactored, you should use the Crowdin UI to **update your Crowdin files manually**:

## docusaurus-crowdin-example settings

The screenshot shows the Crowdin UI interface with the 'Files' tab selected. On the left, there's a tree view of a repository structure: 'blog' and 'docs'. Under 'docs', there are files 'doc1.md', 'doc2.md', 'doc3.md', and 'interactiveDoc.mdx', along with a folder 'i18n' containing 'en'. Inside 'en', there are sub-folders 'docusaurus-plug' and 'docusaurus-theme'. A context menu is open over 'doc1.md', listing options: 'Settings', 'Progress', 'View strings', 'Download source', 'Change segmentation', 'Update', 'Rename', and 'Delete'. The 'Delete' option is highlighted in red. To the right of the tree view is a table showing string statistics and a list of strings with their revision numbers and update buttons.

| Name    | Strings | Revision | Actions  |
|---------|---------|----------|----------|
| doc1.md | 100     | 1        | Update ⚡ |
|         | 2       | 1        | Update ⚡ |
|         | 42      | 1        | Update ⚡ |
|         | 4       | 2        | Update ⚡ |
|         | 2       |          | Update ⚡ |
|         | 12      |          | Update ⚡ |
|         | 1       | 1        | Update ⚡ |

## VCS (Git) integrations

Crowdin has multiple VCS integrations for [GitHub](#), GitLab, Bitbucket.



We recommend avoiding them.

It could have been helpful to be able to edit the translations in both Git and Crowdin, and have a **bi-directional sync** between the 2 systems.

In practice, it **didn't work very reliably** for a few reasons:

- The Crowdin -> Git sync works fine (with a pull request)
- The Git -> Crowdin sync is manual (you have to press a button)
- The heuristics used by Crowdin to match existing Markdown translations to existing Markdown sources are not 100% reliable, and you have to verify the result on Crowdin UI after any sync from Git
- 2 users concurrently editing on Git and Crowdin can lead to a translation loss
- It requires the `crowdin.yml` file to be at the root of the repository

## In-Context localization

Crowdin has an [In-Context localization](#) feature.

### WARNING

Unfortunately, it does not work yet for technical reasons, but we have good hope it can be solved.

Crowdin replaces Markdown strings with technical IDs such as `crowdin:id12345`, but it does so too aggressively, including hidden strings, and messes up with front matter, admonitions, JSX...

## Localize edit URLs

When the user is browsing a page at `/fr/doc1`, the edit button will link by default to the unlocalized doc at `website/docs/doc1.md`.

You may prefer the edit button to link to the Crowdin interface instead by using the `editUrl` function to customize the edit URLs on a per-locale basis.

`docusaurus.config.js`

```
const DefaultLocale = 'en';

export default {
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 docs: {
 editUrl: ({locale, versionDocsDirPath, docPath}) => {
 // Link to Crowdin for French docs
 if (locale !== DefaultLocale) {
 return `https://crowdin.com/project/docusaurus-v2/${locale}`;
 }
 // Link to GitHub for English docs
 return
`https://github.com/facebook/docusaurus/edit/main/website/${versionDocsDirPath}/${docPath}`;
 },
 },
 blog: {
 editUrl: ({locale, blogDirPath, blogPath}) => {
 if (locale !== DefaultLocale) {
 return `https://crowdin.com/project/docusaurus-v2/${locale}`;
 }
 return
`https://github.com/facebook/docusaurus/edit/main/website/${blogDirPath}/${blogPath}`;
 },
 },
],
],
],
}
```

```
],
],
};
```

 **NOTE**

It is currently **not possible to link to a specific file** in Crowdin.

## Example configuration

The **Docusaurus configuration file** is a good example of using versioning and multi-instance:

`crowdin.yml`

```
project_id: '428890'
api_token_env: CROWDIN_PERSONAL_TOKEN
preserve_hierarchy: true
languages_mapping: &languages_mapping
 two_letters_code:
 pt-BR: pt-BR
files:
 - source: /website/i18n/en/**/*
 translation: /website/i18n/%two_letters_code%/**/%original_file_name%
 languages_mapping: *languages_mapping
 - source: /website/docs/**/*
 translation: /website/i18n/%two_letters_code%/docusaurus-plugin-content-
docs/current/**/%original_file_name%
 languages_mapping: *languages_mapping
 - source: /website/community/**/*
 translation: /website/i18n/%two_letters_code%/docusaurus-plugin-content-docs-
community/current/**/%original_file_name%
 languages_mapping: *languages_mapping
 - source: /website/versioned_docs/**/*
 translation: /website/i18n/%two_letters_code%/docusaurus-plugin-content-
docs/**/%original_file_name%
 languages_mapping: *languages_mapping
 - source: /website/blog/**/*
 translation: /website/i18n/%two_letters_code%/docusaurus-plugin-content-
blog/**/%original_file_name%
 languages_mapping: *languages_mapping
 - source: /website/src/pages/**/*
 translation: /website/i18n/%two_letters_code%/docusaurus-plugin-content-
pages/**/%original_file_name%
 ignore: [/**/*.js, /**/*.jsx, /**/*.ts, /**/*.tsx, /**/*.css]
 languages_mapping: *languages_mapping
```

## Machine Translation (MT) issue: links/image handling

Crowdin recently rolled out some major changes to the markdown file format and now the links are treated differently than they were before. Before they were considered as tags, but now they appear as plain text. Because of these changes the plain text links are passed to the MT engine which attempts to translate the target, thus breaking the translation (for instance: this string `Allez voir [ma  
merveilleuse page](/ma-merveilleuse-page)` is translated `Check out [my wonderful page]  
(/my-wonderful-page)`, and this breaks docusaurus i18n workflow as the page name should not be translated).

As of 2023 Dec.7, they are not planning to change the logic of how links are treated, so you should have this in mind if you plan to use Crowdin with MT.

# What's next?

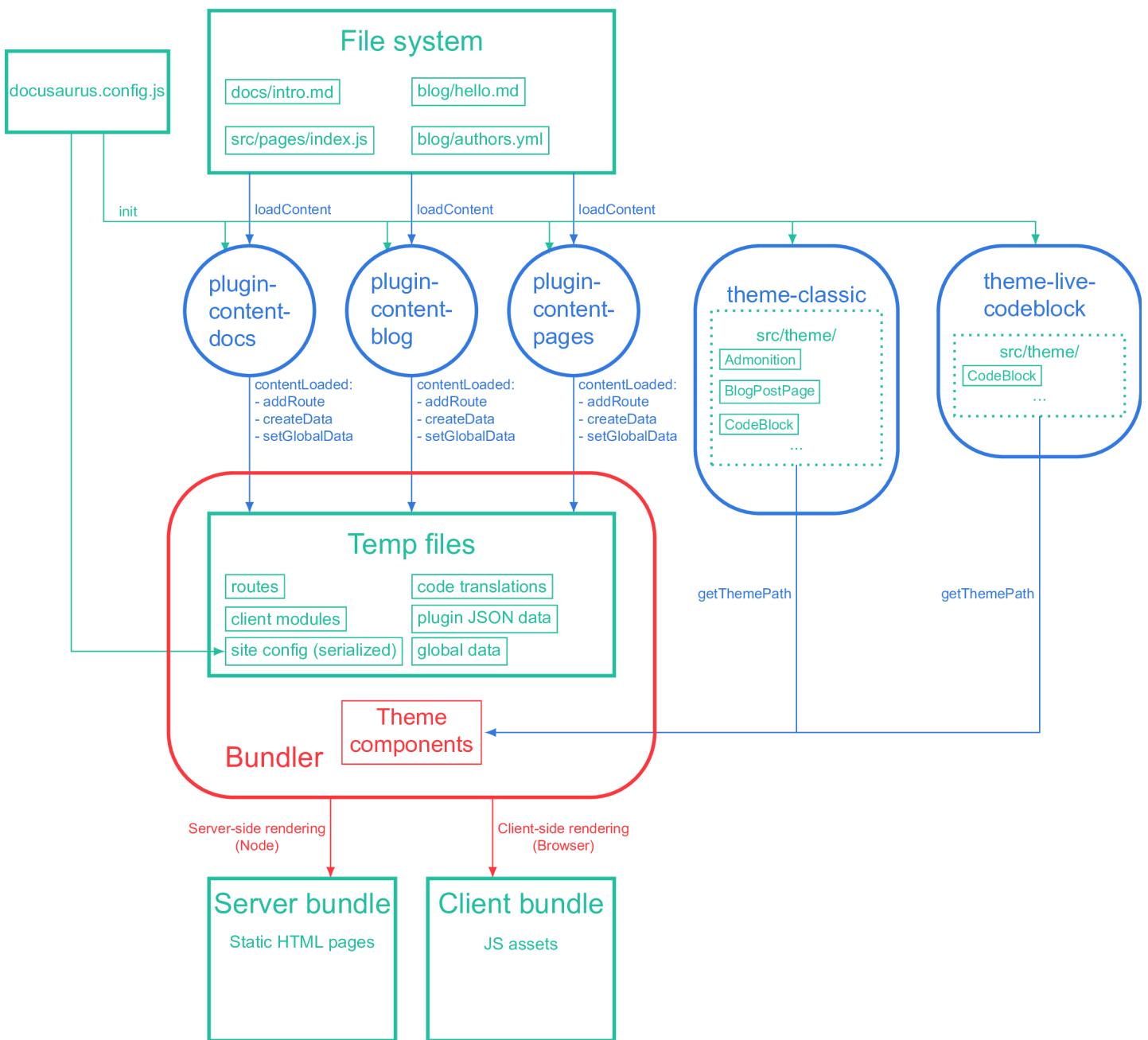
Congratulations! You have understood most core features of Docusaurus now. You have:

- Used the [pages plugin](#) to create a standalone React / Markdown page
- Used the [docs plugin](#) to create documentation pages. This includes [configuring the sidebar](#), and even [versioning](#)
- Used the [blog plugin](#) to create a fully featured blog
- Tried your hands on [a range of Markdown features](#), which are useful for all content plugins
- Used [stylesheets](#) or [swizzling](#) to customize your site's appearance
- Included [images and other assets](#) in your pages
- Added [search](#) to your site
- Understood how [browser support](#) and [SEO](#) are done through standard Docusaurus APIs
- Learned about how [individual plugins](#) are installed and configured
- Deployed your site to a content host
- Internationalized your site to include multiple languages

At this point, you probably have a big `docusaurus.config.js` already 😊 However, you haven't written much code yet! Most of the features are implemented through calling encapsulated Docusaurus APIs. As you continue your journey, you can take three paths:

- Learn more advanced Docusaurus concepts. This will help you gain a deeper understand of what the APIs do.
- Read about [all existing APIs](#). Many of them have not been covered in the Guides!
- Learn to [develop a plugin](#) to extend the functionality of your site.

# Architecture



This diagram shows how Docusaurus works to build your app. Plugins each collect their content and emit JSON data; themes provide layout components which receive the JSON data as route modules. The bundler bundles all the components and emits a server bundle and a client bundle.

Although you (either plugin authors or site creators) are writing JavaScript all the time, bear in mind that the JS is actually run in different environments:

- All plugin lifecycle methods are run in Node. Therefore, until we support ES Modules in our codebase, plugin source code must be provided as ES modules that can be imported, or CommonJS that can be `require'd`.
- The theme code is built with Webpack. They can be provided as ESM—following React conventions.

Plugin code and theme code never directly import each other: they only communicate through protocols (in our case, through JSON temp files and calls to `addRoute`). A useful mental model is to imagine that the plugins are not written in JavaScript, but in another language like Rust. The only way to interact with plugins for the user is through `docusaurus.config.js`, which itself is run in Node (hence you can use `require` and pass callbacks as plugin options).

During bundling, the config file itself is serialized and bundled, allowing the theme to access config options like `themeConfig` or `baseUrl` through `useDocusaurusContext()`. However, the `siteConfig` object only contains **serializable values** (values that are preserved after `JSON.stringify()`). Functions, regexes, etc. would be lost on the client side. The `themeConfig` is designed to be entirely serializable.

# Plugins

Plugins are the building blocks of features in a Docusaurus site. Each plugin handles its own individual feature. Plugins may work and be distributed as part of a bundle via presets.

## Creating plugins

A plugin is a function that takes two parameters: `context` and `options`. It returns a plugin instance object (or a promise). You can create plugins as functions or modules. For more information, refer to the [plugin method references section](#).

### Function definition

You can use a plugin as a function directly included in the Docusaurus config file:

`docusaurus.config.js`

```
export default {
 // ...
 plugins: [
 async function myPlugin(context, options) {
 // ...
 return {
 name: 'my-plugin',
 async loadContent() {
 // ...
 },
 async contentLoaded({content, actions}) {
 // ...
 },
 /* other lifecycle API */
 };
 },
],
};
```

### Module definition

You can use a plugin as a module path referencing a separate file or npm package:

## docusaurus.config.js

```
export default {
 // ...
 plugins: [
 // without options:
 './my-plugin',
 // or with options:
 ['./my-plugin', options],
],
};
```

Then in the folder `my-plugin`, you can create an `index.js` such as this:

### my-plugin/index.js

```
export default async function myPlugin(context, options) {
 // ...
 return {
 name: 'my-plugin',
 async loadContent() {
 /* ... */
 },
 async contentLoaded({content, actions}) {
 /* ... */
 },
 /* other lifecycle API */
 };
}
```

You can view all plugins installed in your site using the [debug plugin's metadata panel](#).

Plugins come as several types:

- `package`: an external package you installed
- `project`: a plugin you created in your project, given to Docusaurus as a local file path
- `local`: a plugin created using the function definition
- `synthetic`: a "fake plugin" Docusaurus created internally, so we take advantage of our modular architecture and don't let the core do much special work. You won't see this in the metadata because it's an implementation detail.

You can access them on the client side with

```
useDocusaurusContext().siteMetadata.pluginVersions.
```

## Plugin design

Docusaurus' implementation of the plugins system provides us with a convenient way to hook into the website's lifecycle to modify what goes on during development/build, which involves (but is not limited to) extending the webpack config, modifying the data loaded, and creating new components to be used in a page.

## Theme design

When plugins have loaded their content, the data is made available to the client side through actions like `createData` + `addRoute` or `setGlobalData`. This data has to be *serialized* to plain strings, because [plugins and themes run in different environments](#). Once the data arrives on the client side, the rest becomes familiar to React developers: data is passed along components, components are bundled with Webpack, and rendered to the window through `ReactDOM.render`...

**Themes provide the set of UI components to render the content.** Most content plugins need to be paired with a theme in order to be actually useful. The UI is a separate layer from the data schema, which makes swapping designs easy.

For example, a Docusaurus blog may consist of a blog plugin and a blog theme.

### NOTE

This is a contrived example: in practice, `@docusaurus/theme-classic` provides the theme for docs, blog, and layouts.

```
docusaurus.config.js
```

```
export default {
 themes: ['theme-blog'],
 plugins: ['plugin-content-blog'],
};
```

And if you want to use Bootstrap styling, you can swap out the theme with `theme-blog-bootstrap` (another fictitious non-existing theme):

```
docusaurus.config.js
```

```
export default {
 themes: ['theme-blog-bootstrap'],
 plugins: ['plugin-content-blog'],
};
```

Now, although the theme receives the same data from the plugin, how the theme chooses to *render* the data as UI can be drastically different.

While themes share the exact same lifecycle methods with plugins, themes' implementations can look very different from those of plugins based on themes' designed objectives.

Themes are designed to complete the build of your Docusaurus site and supply the components used by your site, plugins, and the themes themselves. A theme still acts like a plugin and exposes some lifecycle methods, but most likely they would not use `loadContent`, since they only receive data from plugins, but don't generate data themselves; themes are typically also accompanied by an `src/theme` directory full of components, which are made known to the core through the `getThemePath` lifecycle.

To summarize:

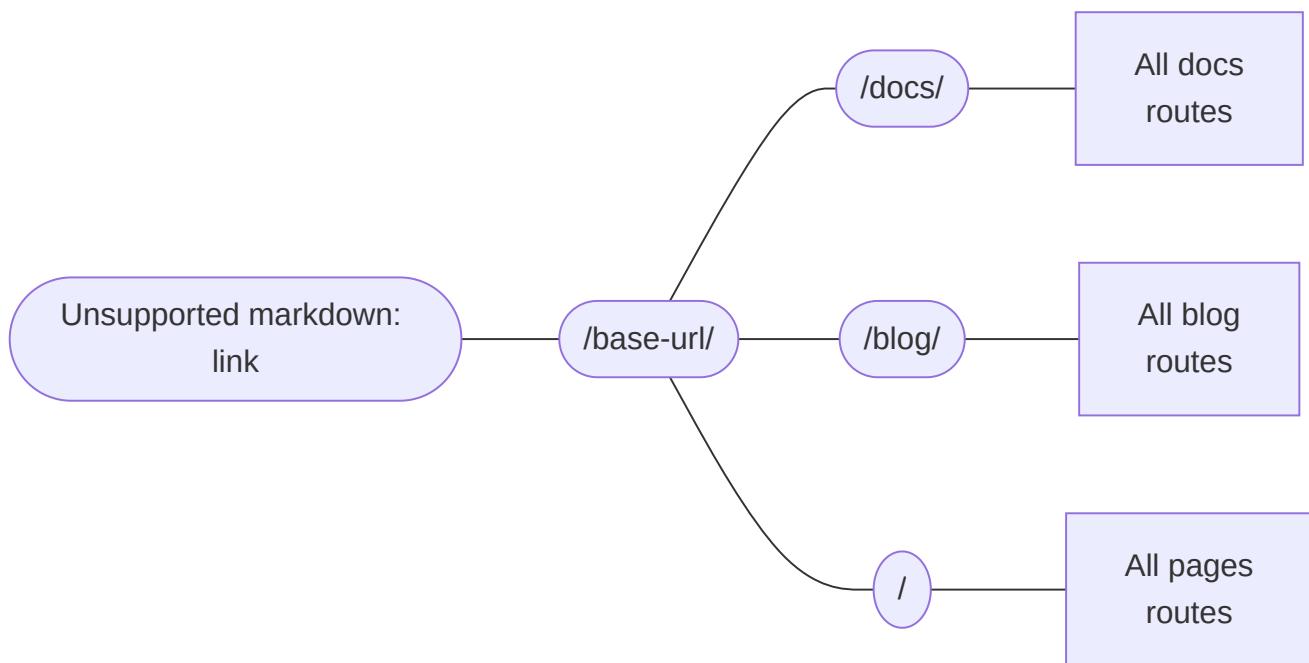
- Themes share the same lifecycle methods with Plugins
- Themes are run after all existing Plugins
- Themes add component aliases by providing `getThemePath`.

# Routing

Docusaurus' routing system follows single-page application conventions: one route, one component. In this section, we will begin by talking about routing within the three content plugins (docs, blog, and pages), and then go beyond to talk about the underlying routing system.

## Routing in content plugins

Every content plugin provides a `routebasePath` option. It defines where the plugins append their routes to. By default, the docs plugin puts its routes under `/docs`; the blog plugin, `/blog`; and the pages plugin, `/`. You can think about the route structure like this:



Any route will be matched against this nested route config until a good match is found. For example, when given a route `/docs/configuration`, Docusaurus first enters the `/docs` branch, and then searches among the subroutes created by the docs plugin.

Changing `routebasePath` can effectively alter your site's route structure. For example, in [Docs-only mode](#), we mentioned that configuring `routebasePath: '/'` for docs means that all routes that the docs plugin creates would not have the `/docs` prefix, yet it doesn't prevent you from having more subroutes like `/blog` created by other plugins.

Next, let's look at how the three plugins structure their own "boxes of subroutes".

## Pages routing

Pages routing are straightforward: the file paths directly map to URLs, without any other way to customize. See the [pages docs](#) for more information.

The component used for Markdown pages is `@theme/MDXPage`. React pages are directly used as the route's component.

## Blog routing

The blog creates the following routes:

- **Posts list pages:** `/`, `/page/2`, `/page/3`...
  - The route is customizable through the `pagebasePath` option.
  - The component is `@theme/BlogListPage`.
- **Post pages:** `/2021/11/21/algolia-docsearch-migration`, `/2021/05/12/announcing-docusaurus-two-beta`...
  - Generated from each Markdown post.
  - The routes are fully customizable through the `slug` front matter.
  - The component is `@theme/BlogPostPage`.
- **Tags list page:** `/tags`
  - The route is customizable through the `tagsbasePath` option.
  - The component is `@theme/BlogTagsListPage`.
- **Tag pages:** `/tags/adoption`, `/tags/beta`...
  - Generated through the tags defined in each post's front matter.
  - The routes always have base defined in `tagsbasePath`, but the subroutes are customizable through the tag's `permalink` field.
  - The component is `@theme/BlogTagsPostsPage`.
- **Archive page:** `/archive`
  - The route is customizable through the `archivebasePath` option.
  - The component is `@theme/BlogArchivePage`.

## Docs routing

The docs is the only plugin that creates **nested routes**. At the top, it registers **version paths**: `/`, `/next`, `/2.0.0-beta.13`... which provide the version context, including the layout and sidebar. This ensures that when switching between individual docs, the sidebar's state is preserved, and that you can switch between versions through the navbar dropdown while staying on the same doc. The component used is `@theme/DocPage`.

The individual docs are rendered in the remaining space after the navbar, footer, sidebar, etc. have all been provided by the `DocPage` component. For example, this page, `/docs/advanced/routing`, is generated from the file at `./versioned_docs/version-3.8.1/advanced/routing.md`. The component used is `@theme/DocItem`.

The doc's `slug` front matter customizes the last part of the route, but the base route is always defined by the plugin's `routebasePath` and the version's `path`.

## File paths and URL paths

Throughout the documentation, we always try to be unambiguous about whether we are talking about file paths or URL paths. Content plugins usually map file paths directly to URL paths, for example, `./docs/advanced/routing.md` will become `/docs/advanced/routing`. However, with `slug`, you can make URLs totally decoupled from the file structure.

When writing links in Markdown, you could either mean a *file path*, or a *URL path*, which Docusaurus would use several heuristics to determine.

- If the path has a `@site` prefix, it is *always* an asset file path.
- If the path has an `http(s)://` prefix, it is *always* a URL path.
- If the path doesn't have an extension, it is a URL path. For example, a link `[page](../plugins)` on a page with URL `/docs/advanced/routing` will link to `/docs/plugins`. Docusaurus will only detect broken links when building your site (when it knows the full route structure), but will make no assumptions about the existence of a file. It is exactly equivalent to writing `<a href="../plugins">page</a>` in a JSX file.
- If the path has an `.md(x)` extension, Docusaurus would try to resolve that Markdown file to a URL, and replace the file path with a URL path.
- If the path has any other extension, Docusaurus would treat it as *an asset* and bundle it.

The following directory structure may help you visualize this file → URL mapping. Assume that there's no slug customization in any page.

### ▼ A sample site structure

So much about content plugins. Let's take one step back and talk about how routing works in a Docusaurus app in general.

## Routes become HTML files

Because Docusaurus is a server-side rendering framework, all routes generated will be server-side rendered into static HTML files. If you are familiar with the behavior of HTTP servers like [Apache2](#), you will understand how this is done: when the browser sends a request to the route `/docs/advanced/routing`, the server interprets that as request for the HTML file `/docs/advanced/routing/index.html`, and returns that.

The `/docs/advanced/routing` route can correspond to either `/docs/advanced/routing/index.html` or `/docs/advanced/routing.html`. Some hosting providers differentiate between them using the presence of a trailing slash, and may or may not tolerate the other. Read more in the [trailing slash guide](#).

For example, the build output of the directory above is (ignoring other assets and JS bundle):

#### ▼ Output of the above workspace

If `trailingSlash` is set to `false`, the build would emit `intro.html` instead of `intro/index.html`.

All HTML files will reference its JS assets using absolute URLs, so in order for the correct assets to be located, you have to configure the `baseUrl` field. Note that `baseUrl` doesn't affect the emitted bundle's file structure: the base URL is one level above the Docusaurus routing system. You can see the aggregate of `url` and `baseUrl` as the actual location of your Docusaurus site.

For example, the emitted HTML would contain links like `<link rel="preload" href="/assets/js/runtime~main.7ed5108a.js" as="script">`. Because absolute URLs are resolved from the host, if the bundle placed under the path `https://example.com/base/`, the link will point to `https://example.com/assets/js/runtime~main.7ed5108a.js`, which is, well, non-existent. By specifying `/base/` as base URL, the link will correctly point to `/base/assets/js/runtime~main.7ed5108a.js`.

Localized sites have the locale as part of the base URL as well. For example, `https://docusaurus.io/zh-CN/docs/advanced/routing/` has base URL `/zh-CN/`.

## Generating and accessing routes

The `addRoute` lifecycle action is used to generate routes. It registers a piece of route config to the route tree, giving a route, a component, and props that the component needs. The props and the component are both provided as paths for the bundler to `require`, because as explained in the [architecture overview](#), server and client only communicate through temp files.

All routes are aggregated in `.docusaurus/routes.js`, which you can view with the debug plugin's [routes panel](#).

On the client side, we offer `@docusaurus/router` to access the page's route. `@docusaurus/router` is a re-export of the `react-router-dom` package. For example, you can use `useLocation` to get the current page's `location`, and `useHistory` to access the `history object`. (They are not the same as the browser API, although similar in functionality. Refer to the React Router documentation for specific APIs.)

This API is **SSR safe**, as opposed to the browser-only `window.location`.

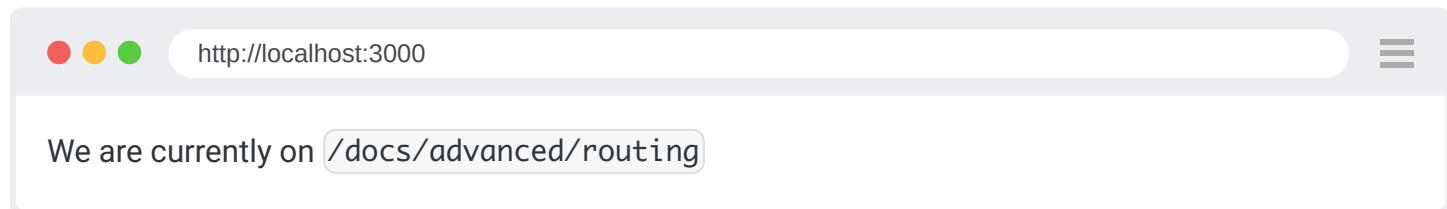
### myComponent.js

```
import React from 'react';
import {useLocation} from '@docusaurus/router';

export function PageRoute() {
 // React router provides the current component's route, even in SSR
 const location = useLocation();
 return (

 We are currently on <code>{location.pathname}</code>

);
}
```



## Escaping from SPA redirects

Docusaurus builds a **single-page application**, where route transitions are done through the `history.push()` method of React router. This operation is done on the client side. However, the prerequisite for a route transition to happen this way is that the target URL is known to our router. Otherwise, the router catches this path and displays a 404 page instead.

If you put some HTML pages under the `static` folder, they will be copied to the build output and therefore become accessible as part of your website, yet it's not part of the Docusaurus route system. We provide a `pathname://` protocol that allows you to redirect to another part of your domain in a non-SPA fashion, as if this route is an external link.

- `[pathname:///pure-html](pathname:///pure-html)`



- `pathname:///pure-html`

The `pathname://` protocol is useful for referencing any content in the static folder. For example, Docusaurus would convert all `Markdown static assets to require() calls`. You can use `pathname://` to keep it a regular link instead of being hashed by Webpack.

### my-doc.md

`![An image from the static](pathname:///img/docusaurus.png)`

`[An asset from the static](pathname:///files/asset.pdf)`

Docusaurus will only strip the `pathname://` prefix without processing the content.

# Static site generation (SSG)

In [architecture](#), we mentioned that the theme is run in Webpack. But beware: that doesn't mean it always has access to browser globals! The theme is built twice:

- During **server-side rendering**, the theme is compiled in a sandbox called [React DOM Server](#). You can see this as a "headless browser", where there is no `window` or `document`, only React. SSR produces static HTML pages.
- During **client-side rendering**, the theme is compiled to JavaScript that gets eventually executed in the browser, so it has access to browser variables.

## ⚠ SSR OR SSG?

*Server-side rendering* and *static site generation* can be different concepts, but we use them interchangeably.

Strictly speaking, Docusaurus is a static site generator, because there's no server-side runtime—we statically render to HTML files that are deployed on a CDN, instead of dynamically pre-rendering on each request. This differs from the working model of [Next.js](#).

Therefore, while you probably know not to access Node globals like `process` ([or can we?](#)) or the `'fs'` module, you can't freely access browser globals either.

```
import React from 'react';

export default function WhereAmI() {
 return {window.location.href};
}
```

This looks like idiomatic React, but if you run `docusaurus build`, you will get an error:

```
ReferenceError: window is not defined
```

This is because during server-side rendering, the Docusaurus app isn't actually run in browser, and it doesn't know what `window` is.

▼ What about `process.env.NODE_ENV`?

## Understanding SSR

React is not just a dynamic UI runtime—it's also a templating engine. Because Docusaurus sites mostly contain static contents, it should be able to work without any JavaScript (which React runs in), but only plain HTML/CSS. And that's what server-side rendering offers: statically rendering your React code into HTML, without any dynamic content. An HTML file has no concept of client state (it's purely markup), hence it shouldn't rely on browser APIs.

These HTML files are the first to arrive at the user's browser screen when a URL is visited (see [routing](#)). Afterwards, the browser fetches and runs other JS code to provide the "dynamic" parts of your site—anything implemented with JavaScript. However, before that, the main content of your page is already visible, allowing faster loading.

In CSR-only apps, all DOM elements are generated on client side with React, and the HTML file only ever contains one root element for React to mount DOM to; in SSR, React is already facing a fully built HTML page, and it only needs to correlate the DOM elements with the virtual DOM in its model. This step is called "hydration". After React has hydrated the static markup, the app starts to work as any normal React app.

Note that Docusaurus is ultimately a single-page application, so static site generation is only an optimization (*progressive enhancement*, as it's called), but our functionality does not fully depend on those HTML files. This is contrary to site generators like [Jekyll](#) and [Docusaurus v1](#), where all files are statically transformed to markup, and interactivity is added through external JavaScript linked with `<script>` tags. If you inspect the build output, you will still see JS assets under `build/assets/js`, which are, really, the core of Docusaurus.

## Escape hatches

If you want to render any dynamic content on your screen that relies on the browser API to be functional at all, for example:

- Our [live codeblock](#), which runs in the browser's JS runtime
- Our [themed image](#) that detects the user's color scheme to display different images
- The JSON viewer of our debug panel which uses the `window` global for styling

You may need to escape from SSR since static HTML can't display anything useful without knowing the client state.

### WARNING

It is important for the first client-side render to produce the exact same DOM structure as server-side rendering, otherwise, React will correlate virtual DOM with the wrong DOM elements.

Therefore, the naïve attempt of `if (typeof window !== 'undefined') /* render something */` won't work appropriately as a browser vs. server detection, because the first client render would instantly render different markup from the server-generated one.

You can read more about this pitfall in [The Perils of Rehydration](#).

We provide several more reliable ways to escape SSR.

## <BrowserOnly>

If you need to render some component in browser only (for example, because the component relies on browser specifics to be functional at all), one common approach is to wrap your component with `<BrowserOnly>` to make sure it's invisible during SSR and only rendered in CSR.

```
import BrowserOnly from '@docusaurus/BrowserOnly';

function MyComponent(props) {
 return (
 <BrowserOnly fallback=<div>Loading...</div>>
 {() => {
 const LibComponent =
 require('some-lib-that-accesses-window').LibComponent;
 return <LibComponent {...props} />;
 }}
 </BrowserOnly>
);
}
```

It's important to realize that the children of `<BrowserOnly>` is not a JSX element, but a function that *returns* an element. This is a design decision. Consider this code:

```
import BrowserOnly from '@docusaurus/BrowserOnly';

function MyComponent() {
 return (
 <BrowserOnly>
 {/* DON'T DO THIS - doesn't actually work */}
 page url = {window.location.href}
 </BrowserOnly>
);
}
```

While you may expect that `BrowserOnly` hides away the children during server-side rendering, it actually can't. When the React renderer tries to render this JSX tree, it does see the

`{window.location.href}` variable as a node of this tree and tries to render it, although it's actually not used! Using a function ensures that we only let the renderer see the browser-only component when it's needed.

## useIsBrowser

You can also use the `useIsBrowser()` hook to test if the component is currently in a browser environment. It returns `false` in SSR and `true` in CSR, after first client render. Use this hook if you only need to perform certain conditional operations on client-side, but not render an entirely different UI.

```
import useIsBrowser from '@docusaurus/useIsBrowser';

function MyComponent() {
 const isBrowser = useIsBrowser();
 const location = isBrowser ? window.location.href : 'fetching location...';
 return {location};
}
```

## useEffect

Lastly, you can put your logic in `useEffect()` to delay its execution until after first CSR. This is most appropriate if you are only performing side-effects but don't get data from the client state.

```
function MyComponent() {
 useEffect(() => {
 // Only logged in the browser console; nothing is logged during server-side
 // rendering
 console.log("I'm now in the browser");
 }, []);
 return Some content...;
}
```

## ExecutionEnvironment

The `ExecutionEnvironment` namespace contains several values, and `canUseDOM` is an effective way to detect browser environment.

Beware that it essentially checked `typeof window !== 'undefined'` under the hood, so you should not use it for rendering-related logic, but only imperative code, like reacting to user input by sending web requests, or dynamically importing libraries, where DOM isn't updated at all.

## a-client-module.js

```
import ExecutionEnvironment from '@docusaurus/ExecutionEnvironment';

if (ExecutionEnvironment.canUseDOM) {
 document.title = "I'm loaded!";
}
```

# Client architecture

## Theme aliases

A theme works by exporting a set of components, e.g. `Navbar`, `Layout`, `Footer`, to render the data passed down from plugins. Docusaurus and users use these components by importing them using the `@theme` webpack alias:

```
import Navbar from '@theme/Navbar';
```

The alias `@theme` can refer to a few directories, in the following priority:

1. A user's `website/src/theme` directory, which is a special directory that has the higher precedence.
2. A Docusaurus theme package's `theme` directory.
3. Fallback components provided by Docusaurus core (usually not needed).

This is called a *layered architecture*: a higher-priority layer providing the component would shadow a lower-priority layer, making swizzling possible. Given the following structure:

```
website
└── node_modules
 └── @docusaurus/theme-classic
 └── theme
 └── Navbar.js
└── src
 └── theme
 └── Navbar.js
```

`website/src/theme/Navbar.js` takes precedence whenever `@theme/Navbar` is imported. This behavior is called component swizzling. If you are familiar with Objective C where a function's implementation can be swapped during runtime, it's the exact same concept here with changing the target `@theme/Navbar` is pointing to!

We already talked about how the "userland theme" in `src/theme` can re-use a theme component through the `@theme-original` alias. One theme package can also wrap a component from another theme, by importing the component from the initial theme, using the `@theme-init` import.

Here's an example of using this feature to enhance the default theme `CodeBlock` component with a `react-live` playground feature.

```
import InitialCodeBlock from '@theme-init/CodeBlock';
import React from 'react';

export default function CodeBlock(props) {
 return props.live ? (
 <ReactLivePlayground {...props} />
) : (
 <InitialCodeBlock {...props} />
);
}
```

Check the code of `@docusaurus/theme-live-codeblock` for details.

### WARNING

Unless you want to publish a re-usable "theme enhancer" (like `@docusaurus/theme-live-codeblock`), you likely don't need `@theme-init`.

It can be quite hard to wrap your mind around these aliases. Let's imagine the following case with a super convoluted setup with three themes/plugins and the site itself all trying to define the same component. Internally, Docusaurus loads these themes as a "stack".

```
+-----+
| `website/src/theme/CodeBlock.js` | <-- `@theme/CodeBlock` always
points to the top
+-----+
| `theme-live-codeblock/theme/CodeBlock/index.js` | <-- `@theme-
original/CodeBlock` points to the topmost non-swizzled component
+-----+
| `plugin-awesome-codeblock/theme/CodeBlock.js` |
+-----+
| `theme-classic/theme/CodeBlock/index.js` | <-- `@theme-init/CodeBlock`
always points to the bottom
+-----+
```

The components in this "stack" are pushed in the order of `preset plugins > preset themes > plugins > themes > site`, so the swizzled component in `website/src/theme` always comes out on top because it's loaded last.

`@theme/*` always points to the topmost component—when `CodeBlock` is swizzled, all other components requesting `@theme/CodeBlock` receive the swizzled version.

`@theme-original/*` always points to the topmost non-swizzled component. That's why you can import `@theme-original/CodeBlock` in the swizzled component—it points to the next one in the

"component stack", a theme-provided one. Plugin authors should not try to use this because your component could be the topmost component and cause a self-import.

`@theme-init/*` always points to the bottommost component—usually, this comes from the theme or plugin that first provides this component. Individual plugins / themes trying to enhance code block can safely use `@theme-init/CodeBlock` to get its basic version. Site creators should generally not use this because you likely want to enhance the *topmost* instead of the *bottommost* component. It's also possible that the `@theme-init/CodeBlock` alias does not exist at all—Docusaurus only creates it when it points to a different one from `@theme-original/CodeBlock`, i.e. when it's provided by more than one theme. We don't waste aliases!

## Client modules

Client modules are part of your site's bundle, just like theme components. However, they are usually side-effect-ful. Client modules are anything that can be `imported` by Webpack—CSS, JS, etc. JS scripts usually work on the global context, like registering event listeners, creating global variables...

These modules are imported globally before React even renders the initial UI.

```
@docusaurus/core/App.tsx
```

```
// How it works under the hood
import '@generated/client-modules';
```

Plugins and sites can both declare client modules, through `getClientModules` and `siteConfig.clientModules`, respectively.

Client modules are called during server-side rendering as well, so remember to check the `execution environment` before accessing client-side globals.

```
mySiteGlobalJs.js
```

```
import ExecutionEnvironment from '@docusaurus/ExecutionEnvironment';

if (ExecutionEnvironment.canUseDOM) {
 // As soon as the site loads in the browser, register a global event listener
 window.addEventListener('keydown', (e) => {
 if (e.code === 'Period') {
 location.assign(location.href.replace('.com', '.dev'));
 }
 });
}
```

```
});
}
```

CSS stylesheets imported as client modules are [global](#).

### mySiteGlobalCss.css

```
/* This stylesheet is global. */
.globalSelector {
 color: red;
}
```

## Client module lifecycles

Besides introducing side-effects, client modules can optionally export two lifecycle functions:

`onRouteUpdate` and `onRouteDidUpdate`.

Because Docusaurus builds a single-page application, `script` tags will only be executed the first time the page loads, but will not re-execute on page transitions. These lifecycles are useful if you have some imperative JS logic that should execute every time a new page has loaded, e.g., to manipulate DOM elements, to send analytics data, etc.

For every route transition, there will be several important timings:

1. The user clicks a link, which causes the router to change its current location.
2. Docusaurus preloads the next route's assets, while keeping displaying the current page's content.
3. The next route's assets have loaded.
4. The new location's route component gets rendered to DOM.

`onRouteUpdate` will be called at event (2), and `onRouteDidUpdate` will be called at (4). They both receive the current location and the previous location (which can be `null`, if this is the first screen).

`onRouteUpdate` can optionally return a "cleanup" callback, which will be called at (3). For example, if you want to display a progress bar, you can start a timeout in `onRouteUpdate`, and clear the timeout in the callback. (The classic theme already provides an `nprogress` integration this way.)

Note that the new page's DOM is only available during event (4). If you need to manipulate the new page's DOM, you'll likely want to use `onRouteDidUpdate`, which will be fired as soon as the DOM on the new page has mounted.

### myClientModule.js

```

export function onRouteDidUpdate({location, previousLocation}) {
 // Don't execute if we are still on the same page; the lifecycle may be fired
 // because the hash changes (e.g. when navigating between headings)
 if (location.pathname !== previousLocation?.pathname) {
 const title = document.getElementsByTagName('h1')[0];
 if (title) {
 title.innerText += ' ❤';
 }
 }
}

export function onRouteUpdate({location, previousLocation}) {
 if (location.pathname !== previousLocation?.pathname) {
 const progressBarTimeout = window.setTimeout(() => {
 nprogress.start();
 }, delay);
 return () => window.clearTimeout(progressBarTimeout);
 }
 return undefined;
}

```

Or, if you are using TypeScript and you want to leverage contextual typing:

#### myClientModule.ts

```

import type {ClientModule} from '@docusaurus/types';

const module: ClientModule = {
 onRouteUpdate({location, previousLocation}){
 // ...
 },
 onRouteDidUpdate({location, previousLocation}){
 // ...
 },
};
export default module;

```

Both lifecycles will fire on first render, but they will not fire on server-side, so you can safely access browser globals in them.

#### PREFER USING REACT

Client module lifecycles are purely imperative, and you can't use React hooks or access React contexts within them. If your operations are state-driven or involve complicated DOM manipulations, you should consider [swizzling components](#) instead.

# Upgrading to Docusaurus v3

This documentation will help you upgrade your site from Docusaurus v2 to Docusaurus v3.

Docusaurus v3 is a new **major version**, including **breaking changes** requiring you to adjust your site accordingly. We will guide you through this process, and also mention a few optional recommendations.

This is not a full rewrite, and the breaking changes are relatively easy to handle. The simplest sites will eventually upgrade by simply updating their npm dependencies.

The main breaking change is the upgrade from MDX v1 to MDX v3. Read the [MDX v2](#) and [MDX v3](#) release notes for details. MDX will now compile your Markdown content **more strictly** and with **subtle differences**.

## BEFORE UPGRADING

Before upgrading, we recommend [preparing your site for Docusaurus v3](#). There are changes that you can already **handle incrementally, under Docusaurus v2**. Doing so will help reduce the work needed to finally upgrade to Docusaurus v3.

For complex sites, we also recommend to set up [visual regression tests](#), a good way to ensure your site stays visually identical. Docusaurus v3 mainly upgrades dependencies, and is not expected to produce any visual changes.

## NOTE

Check the release notes for [Docusaurus v3.0.0](#), and browse the pull-requests for additional useful information and the motivation behind each change mentioned here.

## Upgrading Dependencies

Upgrading to Docusaurus v3 requires upgrading core Docusaurus dependencies ([@docusaurus/name](#)), but also other related packages.

Docusaurus v3 now uses the following dependencies:

- Node.js v18.0+
- React v18.0+
- MDX v3.0+
- TypeScript v5.1+

- prism-react-renderer v2.0+
- react-live v4.0+
- remark-emoji v4.0+
- mermaid v10.4+

## UPGRADING COMMUNITY PLUGINS

If your site uses third-party community plugins and themes, you might need to upgrade them.

Make sure those plugins are compatible with Docusaurus v3 before attempting an upgrade.

A typical `package.json` dependency upgrade example:

### package.json

```
{
 "dependencies": {
 // upgrade to Docusaurus v3
 - "@docusaurus/core": "2.4.3",
 - "@docusaurus/preset-classic": "2.4.3",
 + "@docusaurus/core": "3.0.0",
 + "@docusaurus/preset-classic": "3.0.0",
 // upgrade to MDX v3
 - "@mdx-js/react": "^1.6.22",
 + "@mdx-js/react": "^3.0.0",
 // upgrade to prism-react-renderer v2.0+
 - "prism-react-renderer": "^1.3.5",
 + "prism-react-renderer": "^2.1.0",
 // upgrade to React v18.0+
 - "react": "^17.0.2",
 - "react-dom": "^17.0.2"
 + "react": "^18.2.0",
 + "react-dom": "^18.2.0"
 },
 "devDependencies": {
 // upgrade Docusaurus dev dependencies to v3
 - "@docusaurus/module-type-aliases": "2.4.3",
 - "@docusaurus/types": "2.4.3"
 + "@docusaurus/module-type-aliases": "3.0.0",
 + "@docusaurus/types": "3.0.0"
 }
 "engines": {
 // require Node.js 18.0+
 - "node": ">=16.14"
 + "node": ">=18.0"
 }
}
```

For TypeScript users:

### package.json

```
{
 "devDependencies": {
 // swap the external TypeScript config package for the new official one
 - "@tsconfig/docusaurus": "^1.0.7",
 + "@docusaurus/tsconfig": "3.0.0",
 // upgrade React types to v18.0+
 - "@types/react": "^17.0.69",
 + "@types/react": "^18.2.29",
 // upgrade TypeScript to v5.1+
 - "typescript": "~4.7.4"
 + "typescript": "~5.2.2"
 }
}
```

## Upgrading MDX

MDX is a major dependency of Docusaurus responsible for compiling your `.md` and `.mdx` files to React components.

The transition from MDX v1 to MDX v3 is the **main challenge** to the adoption of Docusaurus v3. Most breaking changes come from MDX v2, and MDX v3 is a relatively small release.

Some documents that compiled successfully under Docusaurus v2 might now **fail to compile** under Docusaurus v3.



### FIND PROBLEMATIC CONTENT AHEAD OF TIME

Run `npx docusaurus-mdx-checker` on your site to get a list of files that will now fail to compile under Docusaurus v3.

This command is also a good way to estimate the amount of work to be done to make your content compatible. Remember most of this work can be executed ahead of the upgrade by [preparing your content for Docusaurus v3](#).

Other documents might also **render differently**.



### USE VISUAL REGRESSION TESTS

For large sites where a manual review of all pages is complicated, we recommend you to setup [visual regression tests](#).

Upgrading MDX comes with all the breaking changes documented on the [MDX v2](#) and [MDX v3](#) release blog posts. Most breaking changes come from MDX v2, and MDX v3 is a relatively small release. The [MDX v2 migration guide](#) has a section on how to [update MDX files](#) that will be particularly relevant to us. Also make sure to read the [Troubleshooting MDX](#) page that can help you interpret common MDX error messages.

Make sure to also read our updated [MDX and React](#) documentation page.

## Using the MDX playground

The MDX playground is your new best friend. It permits to understand how your content is **compiled to React components**, and troubleshoot compilation or rendering issues in isolation.

- [MDX playground - current version](#)
- [MDX playground - v1](#)

### ▼ Configuring the MDX playground options for Docusaurus

Using the two MDX playgrounds side-by-side, you will soon notice that some content is compiled differently or fails to compile in v2.

#### MAKING YOUR CONTENT FUTURE-PROOF

The goal will be to refactor your problematic content so that it **works fine with both versions of MDX**. This way, when you upgrade to Docusaurus v3, this content will already work out-of-the-box.

## Using the MDX checker CLI

We provide a [docusaurus-mdx-checker](#) CLI that permits to easily spot problematic content. Run this command on your site to obtain a list of files that will fail to compile under MDX v3.

```
npx docusaurus-mdx-checker
```

For each compilation issue, the CLI will log the file path and a line number to look at.

```
[+ docusaurus-v2 git:(main) npx docusaurus-mdx-checker
[ERROR] 7/248 MDX files couldn't compile!

Error while compiling file _dogfooding/_pages tests/markdownPageTests.md
Details: These MDX global variables do not seem to be available in scope: b

Error while compiling file docs/api/plugins/plugin-pwa.md (Line=123 Column=48)
Details: Unexpected character '=' (U+003D) before name, expected a character that can start a name, such as a letter, '$', or '_'

Error while compiling file docs/deployment.mdx (Line=443 Column=1)
Details: Unexpected closing tag '</details>', expected corresponding closing tag for '<TabItem>' (320:1-320:48)

Error while compiling file docs/introduction.mdx (Line=57 Column=6)
Details: Unexpected character '/' (U+002F) after self-closing slash, expected '>' to end the tag (note: JS comments in JSX tags are not supported in MDX)

Error while compiling file versioned_docs/version-2.0.0-rc.1/api/plugins/plugin-pwa.md (Line=123 Column=48)
Details: Unexpected character '=' (U+003D) before name, expected a character that can start a name, such as a letter, '$', or '_'

Error while compiling file versioned_docs/version-2.0.0-rc.1/deployment.mdx (Line=443 Column=1)
Details: Unexpected closing tag '</details>', expected corresponding closing tag for '<TabItem>' (320:1-320:48)

Error while compiling file versioned_docs/version-2.0.0-rc.1/introduction.md (Line=57 Column=6)
Details: Unexpected character '/' (U+002F) after self-closing slash, expected '>' to end the tag (note: JS comments in JSX tags are not supported in MDX)

```

## TIP

Use this CLI to estimate of how much work will be required to make your content compatible with MDX v3.

## WARNING

This CLI is a best effort, and will **only report compilation errors**.

It will not report subtle compilation changes that do not produce errors but can affect how your content is displayed. To catch these problems, we recommend using [visual regression tests](#).

## Common MDX problems

Docusaurus cannot document exhaustively all the changes coming with MDX. That's the responsibility of the [MDX v2](#) and [MDX v3](#) migration guides.

However, by upgrading a few Docusaurus sites, we noticed that most of the issues come down to only a few cases that we have documented for you.

### Bad usage of `{}`

The `{}` character is used for opening [JavaScript expressions](#). MDX will now fail if what you put inside `{expression}` is not a valid expression.

#### example.md

The object shape looks like `{username: string, age: number}`

## ERROR MESSAGE

## HOW TO UPGRADE

Available options to fix this error:

- Use inline code: `{username: string, age: number}`
- Use the HTML code: `&#123;`
- Escape it: `\{`

## Bad usage of `<`

The `<` character is used for opening [JSX tags](#). MDX will now fail if it thinks your JSX is invalid.

### example.md

Use Android version <5

You can use a generic type like `Array<T>`

Follow the template "Road to <YOUR\_MINOR\_VERSION>"

## ERROR MESSAGES

Unexpected character `5` (U+0035) before name, expected a character that can start a name, such as a letter, `$`, or `_`

Expected a closing tag for `<T>` (1:6-1:9) before the end of `paragraph` end-tag-mismatch mdast-util-mdx-jsx

Expected a closing tag for `<YOUR_MINOR_VERSION>` (134:19-134:39) before the end of `paragraph`

## HOW TO UPGRADE

Available options to fix this error:

- Use inline code: `Array<T>`
- Use the HTML code: `&lt;` or `&#60;`
- Escape it: `\<`

## Bad usage of GFM Autolink

Docusaurus supports [GitHub Flavored Markdown \(GFM\)](#), but [autolink](#) using the `<link>` syntax is not supported anymore by MDX.

## example.md

```
<sebastien@thisweekinreact.com>
<http://localhost:3000>
```

### 🔥 ERROR MESSAGES

Unexpected character @ (U+0040) in name, expected a name character such as letters, digits, \$, or \_; whitespace before attributes; or the end of the tag (note: to create a link in MDX, use [text](url))

Unexpected character / (U+002F) before local name, expected a character that can start a name, such as a letter, \$, or \_ (note: to create a link in MDX, use [text](url))

### 💡 HOW TO UPGRADE

Use regular Markdown links, or remove the < and >. MDX and GFM are able to autolink literals already.

## example.md

```
sebastien@thisweekinreact.com
sebastien@thisweekinreact.com

http://localhost:3000
http://localhost:3000
```

## Lower-case MDXComponent mapping

For users providing a custom MDXComponent mapping, components are now "sandboxed":

- a MDXComponent mapping for h1 only gets used for # hi but not for <h1>hi</h1>
- a **lower-cased** custom element name will not be substituted by its respective MDXComponent component anymore

### 🔥 VISUAL DIFFERENCE

Your MDXComponent component mapping might not be applied as before, and your custom components might no longer be used.

### 💡 HOW TO UPGRADE

For native Markdown elements, you can keep using **lower-case**: p, h1, img, a...

For any other element, use upper-case names.

src/theme/MDXComponents.js

```
import MDXComponents from '@theme-original/MDXComponents';

export default {
 ...MDXComponents,
 p: (props) => <p {...props} className="my-paragraph"/>
- myElement: (props) => <div {...props} className="my-class" />,
+ MyElement: (props) => <div {...props} className="my-class" />,
};
```

## Unintended extra paragraphs

In MDX v3, it is now possible to interleave JSX and Markdown more easily without requiring extra line breaks. Writing content on multiple lines can also produce new expected `<p>` tags.

### 🔥 VISUAL DIFFERENCE

See how this content is rendered differently by MDX v1 and v3.

example.md

```
<div>Some **Markdown** content</div>
<div>
 Some **Markdown** content
</div>
```

MDX v1 output

```
<div>Some **Markdown** content</div>
<div>Some **Markdown** content</div>
```

MDX v3 output

```
<div>Some Markdown content</div>
<div><p>Some Markdown content</p></div>
```

### 💡 HOW TO UPGRADE

If you don't want an extra `<p>` tag, refactor content on a case by case basis to use a single-line JSX tag.

```
<figure>

- <figcaption>
- My image caption
- </figcaption>
+ <figcaption>My image caption</figcaption>
</figure>
```

You can also wrap such content with `{` and `}` to avoid extra `<p>` tags if you don't intend to use Markdown syntax there yet.

```
-<figure>
+{<figure>

 <figcaption>
 My image caption
 </figcaption>
-</figure>
+</figure>}
```

## Unintended usage of directives

Docusaurus v3 now uses [Markdown Directives](#) (implemented with [remark-directive](#)) as a generic way to provide support for admonitions, and other upcoming Docusaurus features.

### example.md

```
This is a :textDirective

::leafDirective

:::containerDirective

Container directive content

:::
```

#### 🔥 VISUAL CHANGE

Directives are parsed with the purpose of being handled by other Remark plugins. Unhandled directives will be ignored, and won't be rendered back in their original form.

example.md

The AWS re:Invent conf is great

Due to `:Invent` being parsed as a text directive, this will now be rendered as:

The AWS re  
conf is great

### HOW TO UPGRADE

- Use the HTML code: `&#58;`
- Add a space after `:` (if it makes sense): `: text`
- Escape it: `\:`

## Unsupported indented code blocks

MDX does not transform indented text as code blocks anymore.

example.md

```
console.log("hello");
```

### VISUAL CHANGE

The upgrade does not generally produce new MDX compilation errors, but can lead to content being rendered in an unexpected way because there isn't a code block anymore.

### HOW TO UPGRADE

Use the regular code block syntax instead of indentation:

example.md

```
```js
console.log('hello');
```
```

## Other Markdown incompatibilities

## Emphasis starting or ending with a space or a punctuation

New MDX parser now strictly complies with the CommonMark spec. CommonMark spec has introduced rules for emphasis around spaces and punctuation, which are incompatible especially with languages that do not use a space to split words, since v0.14.

Japanese and Chinese are most affected by this, but there are some other languages that can be affected (e.g. Thai and Khmer), for example when you try to emphasize an inline code or a link. Languages that use a space to split words are much less affected.

\*\* (other than `\*\*) in the following example were parsed as intended in Docusaurus 2, but are not now in Docusaurus 3.

### example.md

```
Do not end a range of emphasis with a space. **Or `` will not work as intended.
```

```
<!-- Japanese -->
「。」の後に文を続けると``が意図した動作をしません。**また、**[リンク]
(https://docusaurus.io/)**や**`コード`**のすぐ外側に`**`、そのさらに外側に句読点以外がある場合も同様です。
```

▼ See the detailed conditions and how to upgrade

## MDX plugins

All the official packages (Unified, Remark, Rehype...) in the MDX ecosystem are now **ES Modules only** and do not support **CommonJS** anymore.

In practice this means that you can't do `require("remark-plugin")` anymore.

### 💡 HOW TO UPGRADE

Docusaurus v3 now supports **ES Modules** configuration files. We recommend that you migrate your config file to ES module, that enables you to import the Remark plugins easily:

#### docusaurus.config.js

```
import remarkPlugin from 'remark-plugin';

export default {
 title: 'Docusaurus',
```

```
/* site config using remark plugins here */
};
```

If you want to keep using CommonJS modules, you can use dynamic imports as a workaround that enables you to import ES modules inside a CommonJS module. Fortunately, the [Docusaurus config supports the usage of an async function](#) to let you do so.

#### docusaurus.config.js

```
module.exports = async function () {
 const myPlugin = (await import('remark-plugin')).default;
 return {
 // site config...
 };
};
```

#### (!) FOR PLUGIN AUTHORS

If you created custom Remark or Rehype plugins, you may need to refactor those, or eventually rewrite them completely, due to how the new AST is structured. We have created a [dedicated support discussion](#) to help plugin authors upgrade their code.

## Formatters

Prettier, the most common formatter, supports only the legacy MDX v1, not v3 yet as of Docusaurus v3.0.0. You can add `{/* prettier-ignore */}` before the incompatible parts of your code to make it work with Prettier.

```
{/* prettier-ignore */}
<SomeComponent>Some long text in the component</SomeComponent>
```

If you get tired of too many `{/* prettier-ignore */}` insertions, you can consider disabling MDX formatting by Prettier by adding the following to your `.prettierignore` file, until it starts supporting MDX v3:

```
.prettierignore
```

```
*.mdx
```

# Other Breaking Changes

Apart the MDX v3 upgrade, here is an exhaustive list of breaking changes coming with Docusaurus v3.

## Node.js v18.0

Node.js 16 [reached End-of-Life](#), and Docusaurus v3 now requires **Node.js >= 18.0**.

### 💡 HOW TO UPGRADE

Install Node.js 18.0+ on your computer.

Eventually, configure your continuous integration, CDN or host to use this new Node.js version.

You can also update your site `package.json` to prevent usage of an older unsupported version:

`package.json`

```
{
 "engines": {
 - "node": ">=16.14"
 + "node": ">=18.0"
 }
}
```

Upgrade your Docusaurus v2 site to Node.js 18 before upgrading to Docusaurus v3.

## React v18.0+

Docusaurus v3 now requires **React >= 18.0**.

React 18 comes with its own breaking changes that should be relatively easy to handle, depending on the amount of custom React code you created for your site. The official themes and plugins are compatible with React 18.

### ❗ HOW TO UPGRADE

Read the official [React v18.0](#) and [How to Upgrade to React 18](#), and look at your own React code to figure out which components might be affected this upgrade.

We recommend to particularly look for:

- Automatic batching for stateful components

- New React hydration errors reported to the console

## EXPERIMENTAL SUPPORT FOR REACT 18 FEATURES

React 18 comes with new features:

- `<Suspense>`
- `React.lazy()`
- `startTransition`

Their Docusaurus support is considered as experimental. We might have to adjust the integration in the future, leading to a different runtime behavior.

## Prism-React-Renderer v2.0+

Docusaurus v3 upgrades [prism-react-renderer](#) to v2.0+. This library is used for code block syntax highlighting.

### HOW TO UPGRADE

This is a new major library version containing breaking changes, and we can't guarantee a strict retro-compatibility. The [prism-react-renderer v2 release notes](#) are not super exhaustive, but there are 3 major changes to be aware of for Docusaurus users.

The dependency should be upgraded:

`package.json`

```
{
 "dependencies": {
 - "prism-react-renderer": "^1.3.5",
 + "prism-react-renderer": "^2.1.0",
 }
}
```

The API to import themes in your Docusaurus config file has been updated:

`docusaurus.config.js`

```
- const lightTheme = require('prism-react-renderer/themes/github');
- const darkTheme = require('prism-react-renderer/themes/dracula');
+ const {themes} = require('prism-react-renderer');
+ const lightTheme = themes.github;
+ const darkTheme = themes.dracula;
```

Previously, `react-prism-render` v1 included more languages by default. From v2.0+, less languages are included by default. You may need to add extra languages to your Docusaurus config:

#### docusaurus.config.js

```
const siteConfig = {
 themeConfig: {
 prism: {
 additionalLanguages: ['bash', 'diff', 'json'],
 },
 },
};
```

## React-Live v4.0+

For users of the `@docusaurus/theme-live-codeblock` package, Docusaurus v3 upgrades `react-live` to v4.0+.

#### ⚠ HOW TO UPGRADE

In theory, you have nothing to do, and your existing interactive code blocks should keep working as before.

However, this is a new major library version containing breaking changes, and we can't guarantee a strict retro-compatibility. Read the [v3](#) and [v4](#) changelogs in case of problems.

## remark-emoji v4.0+

Docusaurus v3 upgrades `remark-emoji` to v4.0+. This library is to support `:emoji:` shortcuts in Markdown.

#### ⚠ HOW TO UPGRADE

Most Docusaurus users have nothing to do. Users of emoji shortcodes should read the [changelog](#) and double-check their emojis keep rendering as expected.

**Breaking Change** Update `node-emoji` from v1 to v2. This change introduces support for many new emojis and removes old emoji short codes which are no longer valid on GitHub.

## Mermaid v10.4+

### (!) HOW TO UPGRADE

In theory, you have nothing to do, and your existing diagrams should keep working as before.

However, this is a new major library version containing breaking changes, and we can't guarantee a strict retro-compatibility. Read the [v10](#) changelog in case of problem.

## TypeScript v5.1+

Docusaurus v3 now requires **TypeScript >= 5.1**.

### (!) HOW TO UPGRADE

Upgrade your dependencies to use TypeScript 5+

package.json

```
{
 "devDependencies": {
 - "typescript": "~4.7.4"
 + "typescript": "~5.2.2"
 }
}
```

## TypeScript base config

The official Docusaurus TypeScript config has been re-internalized from the external package [@tsconfig/docusaurus](#) to our new monorepo package [@docusaurus/tsconfig](#).

This new package is versioned alongside all the other Docusaurus core packages, and will be used to ensure TypeScript retro-compatibility and breaking changes on major version upgrades.

### (!) HOW TO UPGRADE

Swap the external TypeScript config package for the new official one

package.json

```
{
 "devDependencies": {
 - "@tsconfig/docusaurus": "^1.0.7",
 }
}
```

```
+ "@docusaurus/tsconfig": "3.0.0",
 }
}
```

Use it in your `tsconfig.json` file:

### tsconfig.json

```
{
- "extends": "@tsconfig/docusaurus/tsconfig.json",
+ "extends": "@docusaurus/tsconfig",
 "compilerOptions": {
 "baseUrl": "."
 }
}
```

## New Config Loader

Docusaurus v3 changes its internal config loading library from `import-fresh` to `jiti`. It is responsible for loading files such as `docusaurus.config.js` or `sidebars.js`, and Docusaurus plugins.

### ⚠ HOW TO UPGRADE

In theory, you have nothing to do, and your existing config files should keep working as before.

However, this is a major dependency swap and subtle behavior changes could occur.

## Admonition Warning

For historical reasons, we support an undocumented admonition `:::warning` that renders with a red color.

### 🔥 WARNING

This is a Docusaurus v2 `:::warning` admonition.

However, the color and icon have always been wrong. Docusaurus v3 re-introduces `:::warning` admonition officially, documents it, and fix the color and icon.

### ⚠ WARNING

This is a Docusaurus v3 `:::warning` admonition.

## ⚠️ HOW TO UPGRADE

If you previously used the undocumented `:::warning` admonition, make sure to verify for each usage if yellow is now an appropriate color. If you want to keep the red color, use `:::danger` instead.

Docusaurus v3 also deprecated the `:::caution` admonition. Please refactor `:::caution` (yellow) to either `:::warning` (yellow) or `:::danger` (red).

If you want to keep the title “caution”, you might want to refactor it to `:::warning[caution]` (yellow).

## Versioned Sidebars

This breaking change will only affect **Docusaurus v2 early adopters** who versioned their docs before `v2.0.0-beta.10` (December 2021).

When creating version `v1.0.0`, the sidebar file contained a prefix `version-v1.0.0/` that **Docusaurus v3 does not support anymore**.

```
versioned_sidebars/version-v1.0.0-sidebars.json
```

```
{
 "version-v1.0.0/docs": [
 "version-v1.0.0/introduction",
 "version-v1.0.0/prerequisites"
]
}
```

## ⚠️ HOW TO UPGRADE

Remove the useless versioned prefix from your versioned sidebars.

```
versioned_sidebars/version-v1.0.0-sidebars.json
```

```
{
 "docs": ["introduction", "prerequisites"]
}
```

## Blog Feed Limit

The `@docusaurus/plugin-content-blog` now limits the RSS feed to the last 20 entries by default. For large Docusaurus blogs, this is a more sensible default value to avoid an increasingly large RSS file.

### (!) HOW TO UPGRADE

In case you don't like this new default behavior, you can revert to the former "unlimited feed" behavior with the new `limit: false` feed option:

`docusaurus.config.js`

```
const blogOptions = {
 feedOptions: {
 limit: false,
 },
};
```

## Docs Theme Refactoring

For users that swizzled docs-related theme components (like `@theme/DocPage`), these components have been significantly refactored to make it easier to customize.

Technically, **this is not a breaking change** because these components are **flagged as unsafe to swizzle**, however many Docusaurus sites ejected docs-related components, and will be interested to know their customizations might break Docusaurus.

### (!) HOW TO UPGRADE

Delete all your swizzled components, re-swizzle them, and re-apply your customizations on top of the newly updated components.

Alternatively, you can look at the [pull-request notes](#) to understand the new theme component tree structure, and eventually try to patch your swizzled components manually.

## Optional Changes

Some changes are not mandatory, but remain useful to be aware of to plainly leverage Docusaurus v3.

## Automatic JSX runtime

Docusaurus v3 now uses the React 18 "[automatic](#)" JSX runtime.

It is not needed anymore to import React in JSX files that do not use any React API.

src/components/MyComponent.js

```
- import React from 'react';

export default function MyComponent() {
 return <div>Hello</div>;
}
```

## ESM and TypeScript Configs

Docusaurus v3 supports ESM and TypeScript config files, and it might be a good idea to adopt those new options.

docusaurus.config.js

```
export default {
 title: 'Docusaurus',
 url: 'https://docusaurus.io',
 // your site config ...
};
```

docusaurus.config.ts

```
import type {Config} from '@docusaurus/types';
import type * as Preset from '@docusaurus/preset-classic';

const config: Config = {
 title: 'My Site',
 favicon: 'img/favicon.ico',
 presets: [
 [
 'classic',
 {
 /* Your preset config here */
 } satisfies Preset.Options,
],
],
 themeConfig: {
 /* Your theme config here */
 } satisfies Preset.ThemeConfig,
};
```

```
export default config;
```

## Using the `.mdx` extension

We recommend using the `.mdx` extension whenever you use JSX, `import`, or `export` (i.e. MDX features) inside a Markdown file. It is semantically more correct and improves compatibility with external tools (IDEs, formatters, linters, etc.).

In future versions of Docusaurus, `.md` files will be parsed as standard [CommonMark](#), which does not support these features. In Docusaurus v3, `.md` files keep being compiled as MDX files, but it will be possible to [opt-in for CommonMark](#).

## Upgrading math packages

If you use Docusaurus to render [Math Equations](#), you should upgrade the MDX plugins.

Make sure to use `remark-math` 6 and `rehype-katex` 7 for Docusaurus v3 (using MDX v3). We can't guarantee other versions will work.

```
{
- "remark-math": "^3.0.0",
+ "remark-math": "^6.0.0",
- "rehype-katex": "^5.0.0"
+ "rehype-katex": "^7.0.0"
}
```

`hast-util-is-element` is now unnecessary in Docusaurus v3. If you have installed it and don't use it somewhere else, you can just remove it by running `npm uninstall hast-util-is-element`.

## Turn off MDX v1 compat

Docusaurus v3 comes with [MDX v1 compatibility options](#), that are turned on by default.

`docusaurus.config.js`

```
export default {
 markdown: {
 mdx1Compat: {
 comments: true,
 admonitions: true,
 headingIds: true,
```

```
 },
 },
};
```

### comments option

This option allows the usage of HTML comments inside MDX, while HTML comments are officially not supported anymore.

For MDX files, we recommend to progressively use MDX `{/* comments */}` instead of HTML `<!-- comments -->`, and then turn this compatibility option off.

#### BLOG TRUNCATE MARKER

The default blog truncate marker now supports both `<!-- truncate -->` and `{/* truncate */}`.

### admonitions option

This option allows the usage of the Docusaurus v2 [admonition title](#) syntax:

```
:::note Your Title
content
:::
```

Docusaurus now implements admonitions with [Markdown Directives](#) (implemented with [remark-directive](#)), and the syntax to provide a directive label requires square brackets:

```
:::note[Your Title]
content
:::
```

We recommend to progressively use the new Markdown directive label syntax, and then turn this compatibility option off.

### headingIds option

This option allows the usage of the Docusaurus v2 [explicit heading id](#) syntax:

### ### Hello World {#my-explicit-id}

This syntax is now invalid MDX, and would require to escape the `{` character: `\{#my-explicit-id\}`.

We recommend to keep this compatibility option on for now, until we provide a new syntax compatible with newer versions of MDX.

## Troubleshooting

In case of any upgrade problem, the first things to try are:

- make sure all your docs compile in the [MDX playground](#), or using `npx docusaurus-mdx-checker`
- delete `node_modules` and `package-lock.json`, and then run `npm install` again
- run `docusaurus clear` to clear the caches
- remove third-party plugins that might not support Docusaurus v3
- delete all your swizzled components

Once you have tried that, you can ask for support through the following support channels:

- [Docusaurus v3 - Upgrade Support](#)
- [Docusaurus v3 - Discord channel #migration-v2-to-v3](#)
- [MDX v3 - Upgrade Support](#)
- [MDX v3 - Remark/Rehype Plugins Support](#)
- [MDX v3 - Discord channel #migration-mdx-v3](#)

Please consider **our time is precious**. To ensure that your support request is not ignored, we kindly ask you to:

- provide a **minimal** reproduction that we can easily run, ideally created with [docusaurus.new](#)
- provide a live deployment url showing the problem in action (if your site can build)
- explain clearly the problem, much more than an ambiguous "it doesn't work"
- include as much relevant material as possible: code snippets, repo url, git branch urls, full stack traces, screenshots and videos
- present your request clearly, concisely, showing us that you have made an effort to help us help you

Alternatively, you can look for a paid [Docusaurus Service Provider](#) to execute this upgrade for you. If your site is open source, you can also ask our community for [free, benevolent help](#).

# Overview

This doc guides you through migrating an existing Docusaurus 1 site to Docusaurus 2.

We try to make this as easy as possible, and provide a migration CLI.

## Main differences

Docusaurus 1 is a pure documentation site generator, using React as a server-side template engine, but not loading React on the browser.

Docusaurus 2, rebuilt from the ground up, generates a single-page-application, using the full power of React in the browser. It allows for more customizability but preserved the best parts of Docusaurus 1 - easy to get started, versioned docs, and i18n.

Beyond that, Docusaurus 2 is a **performant static site generator** and can be used to create common content-driven websites (e.g. Documentation, Blogs, Product Landing and Marketing Pages, etc) extremely quickly.

While our main focus will still be helping you get your documentations right and well, it is possible to build any kind of website using Docusaurus 2 as it is just a React application. **Docusaurus can now be used to build any website, not just documentation websites.**

## Docusaurus 1 structure

Your Docusaurus 1 site should have the following structure:

```
├── docs
└── website
 ├── blog
 ├── core
 │ └── Footer.js
 ├── package.json
 ├── pages
 ├── sidebars.json
 ├── siteConfig.js
 └── static
```

# Docusaurus 2 structure

After the migration, your Docusaurus 2 site could look like:

```
└── docs
 └── website
 ├── blog
 ├── src
 │ ├── components
 │ ├── css
 │ └── pages
 ├── static
 ├── package.json
 ├── sidebars.json
 └── docusaurus.config.js
```

## ! INFO

This migration does not change the `/docs` folder location, but Docusaurus v2 sites generally have the `/docs` folder inside `/website`

You are free to put the `/docs` folder anywhere you want after having migrated to v2.

## Migration process

There are multiple things to migrate to obtain a fully functional Docusaurus 2 website:

- packages
- CLI commands
- site configuration
- Markdown files
- sidebars file
- pages, components and CSS
- versioned docs
- i18n support 🚧

## Automated migration process

The [migration CLI](#) will handle many things of the migration for you.

However, some parts can't easily be automated, and you will have to fallback to the manual process.

### NOTE

We recommend running the migration CLI, and complete the missing parts thanks to the manual migration process.

## Manual migration process

Some parts of the migration can't be automated (particularly the pages), and you will have to migrate them manually.

The [manual migration guide](#) will give you all the manual steps.

## Support

For any questions, you can ask in the [#migration-v1-to-v2](#) Discord channel.

Feel free to tag [@slorber](#) in any migration PRs if you would like us to have a look.

We also have volunteers willing to [help you migrate your v1 site](#).

## Example migration PRs

You might want to refer to our migration PRs for [Create React App](#) and [Flux](#) as examples of how a migration for a basic Docusaurus v1 site can be done.

# Automated migration

The migration CLI automatically migrates your v1 website to a v2 website.

## ! INFO

Manual work is still required after using the migration CLI, as we can't automate a full migration

The migration CLI migrates:

- Site configurations (from `siteConfig.js` to `docusaurus.config.js`)
- `package.json`
- `sidebars.json`
- `/docs`
- `/blog`
- `/static`
- `versioned_sidebar.json` and `/versioned_docs` if your site uses versioning

To use the migration CLI, follow these steps:

1. Before using the migration CLI, ensure that `/docs`, `/blog`, `/static`, `sidebars.json`, `siteConfig.js`, `package.json` follow the expected structure.
2. To migrate your v1 website, run the migration CLI with the appropriate filesystem paths:

```
migration command format
npx @docusaurus/migrate migrate <v1 website directory> <desired v2 website
directory>

example
npx @docusaurus/migrate migrate ./v1-website ./v2-website
```

3. To view your new website locally, go into your v2 website's directory and start your development server.

[npm](#)    [Yarn](#)    [pnpm](#)    [Bun](#)

```
cd ./v2-website
npm install
```

```
npm start
```

## 🔥 DANGER

The migration CLI updates existing files. Be sure to have committed them first!

## Options

You can add option flags to the migration CLI to automatically migrate Markdown content and pages to v2. It is likely that you will still need to make some manual changes to achieve your desired result.

| Name   | Description                                            |
|--------|--------------------------------------------------------|
| --mdx  | Add this flag to convert Markdown to MDX automatically |
| --page | Add this flag to migrate pages automatically           |

*# example using options*

```
npx @docusaurus/migrate migrate --mdx --page ./v1-website ./v2-website
```

## 🔥 DANGER

The migration of pages and MDX is still a work in progress.

We recommend you to try to run the pages without these options, commit, and then try to run the migration again with the `--page` and `--mdx` options.

This way, you'd be able to easily inspect and fix the diff.

# Manual migration

This manual migration process should be run after the [automated migration process](#), to complete the missing parts, or debug issues in the migration CLI output.

## Project setup

### package.json

#### Scoped package names

In Docusaurus 2, we use scoped package names:

- `docusaurus` → `@docusaurus/core`

This provides a clear distinction between Docusaurus' official packages and community maintained packages. In other words, all Docusaurus' official packages are namespaced under `@docusaurus/`.

Meanwhile, the default doc site functionalities provided by Docusaurus 1 are now provided by `@docusaurus/preset-classic`. Therefore, we need to add this dependency as well:

#### package.json

```
{
 dependencies: {
 - "docusaurus": "^1.x.x",
 + "@docusaurus/core": "^2.0.0-beta.0",
 + "@docusaurus/preset-classic": "^2.0.0-beta.0",
 }
}
```



Please use the most recent Docusaurus 2 version, which you can check out [here](#) (using the `latest` tag).

#### CLI commands

Meanwhile, CLI commands are renamed to `docusaurus <command>` (instead of `docusaurus-<command>`).

The "scripts" section of your `package.json` should be updated as follows:

#### package.json

```
{
 "scripts": {
 "start": "docusaurus start",
 "build": "docusaurus build",
 "swizzle": "docusaurus swizzle",
 "deploy": "docusaurus deploy"
 // ...
 }
}
```

A typical Docusaurus 2 `package.json` may look like this:

#### package.json

```
{
 "scripts": {
 "docusaurus": "docusaurus",
 "start": "docusaurus start",
 "build": "docusaurus build",
 "swizzle": "docusaurus swizzle",
 "deploy": "docusaurus deploy",
 "serve": "docusaurus serve",
 "clear": "docusaurus clear"
 },
 "dependencies": {
 "@docusaurus/core": "^2.0.0-beta.0",
 "@docusaurus/preset-classic": "^2.0.0-beta.0",
 "clsx": "^1.1.1",
 "react": "^17.0.2",
 "react-dom": "^17.0.2"
 },
 "browserslist": {
 "production": [">0.5%", "not dead", "not op_mini all"],
 "development": [
 "last 1 chrome version",
 "last 1 firefox version",
 "last 1 safari version"
]
 }
}
```

## Update references to the `build` directory

In Docusaurus 1, all the build artifacts are located within `website/build/<PROJECT_NAME>`.

In Docusaurus 2, it is now moved to just `website/build`. Make sure that you update your deployment configuration to read the generated files from the correct `build` directory.

If you are deploying to GitHub pages, make sure to run `yarn deploy` instead of `yarn publish-gh-pages` script.

## .gitignore

The `.gitignore` in your `website` should contain:

```
.gitignore

dependencies
/node_modules

production
/build

generated files
.docusaurus
.cache-loader

misc
.DS_Store
.env.local
.env.development.local
.env.test.local
.env.production.local

npm-debug.log*
yarn-debug.log*
yarn-error.log*
```

## README

The D1 website may have an existing README file. You can modify it to reflect the D2 changes, or copy the default [Docusaurus v2 README](#).

# Site configurations

## docusaurus.config.js

Rename `siteConfig.js` to `docusaurus.config.js`.

In Docusaurus 2, we split each functionality (blog, docs, pages) into plugins for modularity. Presets are bundles of plugins and for backward compatibility we built a `@docusaurus/preset-classic` preset which bundles most of the essential plugins present in Docusaurus 1.

Add the following preset configuration to your `docusaurus.config.js`.

### docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 docs: {
 // Docs folder path relative to website dir.
 path: '../docs',
 // Sidebars file relative to website dir.
 sidebarPath: require.resolve('./sidebars.json'),
 },
 // ...
 },
],
],
],
 };
};
```

We recommend moving the `docs` folder into the `website` folder and that is also the default directory structure in v2. [Vercel](#) supports [Docusaurus project deployments out-of-the-box](#) if the `docs` directory is within the `website`. It is also generally better for the docs to be within the website so that the docs and the rest of the website code are co-located within one `website` directory.

If you are migrating your Docusaurus v1 website, and there are pending documentation pull requests, you can temporarily keep the `/docs` folder to its original place, to avoid producing conflicts.

Refer to migration guide below for each field in `siteConfig.js`.

## Updated fields

`baseUrl`, `tagline`, `title`, `url`, `favicon`, `organizationName`, `productName`, `githubHost`, `scripts`, `stylesheets`

No actions needed, these configuration fields were not modified.

## colors

Deprecated. We wrote a custom CSS framework for Docusaurus 2 called [Infima](#) which uses CSS variables for theming. The docs are not quite ready yet and we will update here when it is. To overwrite Infima's CSS variables, create your own CSS file (e.g. `./src/css/custom.css`) and import it globally by passing it as an option to `@docusaurus/preset-classic`:

### docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 theme: {
 customCss: [require.resolve('./src/css/custom.css')],

 },
 },
],
],
};
```

Infima uses 7 shades of each color.

### /src/css/custom.css

```
/**
 * You can override the default Infima variables here.
 * Note: this is not a complete list of --ifm- variables.
 */
:root {
 --ifm-color-primary: #25c2a0;
 --ifm-color-primary-dark: rgb(33, 175, 144);
 --ifm-color-primary-darker: rgb(31, 165, 136);
 --ifm-color-primary-darkest: rgb(26, 136, 112);
 --ifm-color-primary-light: rgb(70, 203, 174);
 --ifm-color-primary-lighter: rgb(102, 212, 189);
 --ifm-color-primary-lightest: rgb(146, 224, 208);
}
```

We recommend using [ColorBox](#) to find the different shades of colors for your chosen primary color.

Alternatively, use the following tool to generate the different shades for your website and copy the variables into `src/css/custom.css`.



## TIP

Aim for at least [WCAG-AA contrast ratio](#) for the primary color to ensure readability. Use the Docusaurus website itself to preview how your color palette would look like. You can use alternative palettes in dark mode because one color doesn't usually work in both light and dark mode.

Primary Color: #2e8555



Edit dark mode

Reset

Background:



| CSS Variable Name            | Hex     | Adjustment | Contrast Rating |
|------------------------------|---------|------------|-----------------|
| --ifm-color-primary-lightest | #3cad6e | -30        | Fail 🚫          |
| --ifm-color-primary-lighter  | #359962 | -15        | Fail 🚫          |
| --ifm-color-primary-light    | #33925d | -10        | Fail 🚫          |
| --ifm-color-primary          | #2e8555 | 0          | AA 👍            |
| --ifm-color-primary-dark     | #29784c | 10         | AA 👍            |
| --ifm-color-primary-darker   | #277148 | 15         | AA 👍            |
| --ifm-color-primary-darkest  | #205d3b | 30         | AAA 🎉           |

Replace the variables in `src/css/custom.css` with these new variables.

/src/css/custom.css

```
:root {
 --ifm-color-primary: #2e8555;
 --ifm-color-primary-dark: #29784c;
 --ifm-color-primary-darker: #277148;
 --ifm-color-primary-darkest: #205d3b;
 --ifm-color-primary-light: #33925d;
 --ifm-color-primary-lighter: #359962;
 --ifm-color-primary-lightest: #3cad6e;
}
```

`footerIcon`, `copyright`, `ogImage`, `twitterImage`, `docsSideNavCollapsible`

Site meta info such as assets, SEO, copyright info are now handled by themes. To customize them, use the `themeConfig` field in your `docusaurus.config.js`:

#### docusaurus.config.js

```
module.exports = {
 // ...
 themeConfig: {
 footer: {
 logo: {
 alt: 'Meta Open Source Logo',
 src: '/img/meta_oss_logo.png',
 href: 'https://opensource.facebook.com/',
 },
 copyright: `Copyright © ${new Date().getFullYear()} Facebook, Inc.`, // You
 can also put own HTML here.
 },
 image: 'img/docusaurus.png',
 // ...
 },
};
```

`headerIcon`, `headerLinks`

In Docusaurus 1, header icon and header links were root fields in `siteConfig`:

#### siteConfig.js

```
headerIcon: 'img/docusaurus.svg',
headerLinks: [
 { doc: "doc1", label: "Getting Started" },
 { page: "help", label: "Help" },
 { href: "https://github.com/", label: "GitHub" },
 { blog: true, label: "Blog" },
],
```

Now, these two fields are both handled by the theme:

#### docusaurus.config.js

```
module.exports = {
 // ...
 themeConfig: {
```

```
navbar: {
 title: 'Docusaurus',
 logo: {
 alt: 'Docusaurus Logo',
 src: 'img/docusaurus.svg',
 },
 items: [
 {to: 'docs/doc1', label: 'Getting Started', position: 'left'},
 {to: 'help', label: 'Help', position: 'left'},
 {
 href: 'https://github.com/',
 label: 'GitHub',
 position: 'right',
 },
 {to: 'blog', label: 'Blog', position: 'left'},
],
},
// ...
},
};

};
```

## algolia

### docusaurus.config.js

```
module.exports = {
 // ...
 themeConfig: {
 algolia: {
 apiKey: '47ecd3b21be71c5822571b9f59e52544',
 indexName: 'docusaurus-2',
 algoliaOptions: { //... },
 },
 // ...
 },
};

};
```

## ⚠ WARNING

Your Algolia DocSearch v1 config ([found here](#)) should be updated for Docusaurus v2 ([example](#)).

You can contact the DocSearch team (@shortcuts, @s-pace) for support. They can update it for you and trigger a recrawl of your site to restore the search (otherwise you will have to wait up to 24h for the next scheduled crawl)

## blogSidebarCount

Deprecated. Pass it as a blog option to `@docusaurus/preset-classic` instead:

#### docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 blog: {
 postsPerPage: 10,
 },
 // ...
 },
],
],
};

};
```

#### cname

Deprecated. Create a `CNAME` file in your `static` folder instead with your custom domain. Files in the `static` folder will be copied into the root of the `build` folder during execution of the build command.

`customDocsPath`, `docsUrl`, `editUrl`, `enableUpdateBy`, `enableUpdateTime`

**BREAKING:** `editUrl` should point to (website) Docusaurus project instead of `docs` directory.

Deprecated. Pass it as an option to `@docusaurus/preset-classic` docs instead:

#### docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 docs: {
 // Equivalent to `customDocsPath`.
 path: 'docs',
 // Equivalent to `editUrl` but should point to `website` dir instead of
 // `website/docs`.
 editUrl: 'https://github.com/facebook/docusaurus/edit/main/website',
 // Equivalent to `docsUrl`.
 routebasePath: 'docs',
 }
 }
]
};
```

```
// Remark and Rehype plugins passed to MDX. Replaces `markdownOptions`
and `markdownPlugins`.
 remarkPlugins: [],
 rehypePlugins: [],
 // Equivalent to `enableUpdateBy`.
 showLastUpdateAuthor: true,
 // Equivalent to `enableUpdateTime`.
 showLastUpdateTime: true,
},
// ...
},
],
],
};
```

## gaTrackingId

docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',

```

## gaGtag

docusaurus.config.js

```
module.exports = {
 // ...
 presets: [
 [
 '@docusaurus/preset-classic',
 {
 // ...
 gtag: {
 trackingID: 'UA-141789564-1',
 },
 },
],
],
};
```

```
 },
],
],
};

};
```

## Removed fields

The following fields are all deprecated, you may remove from your configuration file.

- `blogSidebarTitle`
- `cleanUrl` - Clean URL is used by default now.
- `defaultVersionShown` - Versioning is not ported yet. You'd be unable to migration to Docusaurus 2 if you are using versioning. Stay tuned.
- `disableHeaderTitle`
- `disableTitleTagline`
- `docsSideNavCollapsible` is available at `docsPluginOptions.sidebarCollapsible`, and this is turned on by default now.
- `facebookAppId`
- `facebookComments`
- `facebookPixelId`
- `fonts`
- `highlight` - We now use [Prism](#) instead of [highlight.js](#).
- `markdownOptions` - We use MDX in v2 instead of Remarkable. Your Markdown options have to be converted to Remark/Rehype plugins.
- `markdownPlugins` - We use MDX in v2 instead of Remarkable. Your Markdown plugins have to be converted to Remark/Rehype plugins.
- `manifest`
- `onPageNav` - This is turned on by default now.
- `separateCss` - It can imported in the same manner as `custom.css` mentioned above.
- `scrollToTop`
- `scrollToTopOptions`
- `translationRecruitingLink`
- `twitter`
- `twitterUsername`
- `useEnglishUrl`
- `users`

- `usePrism` - We now use [Prism](#) instead of [highlight.js](#)
- `wrapPagesHTML`

We intend to implement many of the deprecated config fields as plugins in future. Help will be appreciated!

## Urls

In v1, all pages were available with or without the `.html` extension.

For example, these 2 pages exist:

- <https://v1.docusaurus.io/docs/en/installation>
- <https://v1.docusaurus.io/docs/en/installation.html>

If `cleanUrl` was:

- `true`: links would target `/installation`
- `false`: links would target `/installation.html`

In v2, by default, the canonical page is `/installation`, and not `/installation.html`.

If you had `cleanUrl: false` in v1, it's possible that people published links to `/installation.html`.

For SEO reasons, and avoiding breaking links, you should configure server-side redirect rules on your hosting provider.

As an escape hatch, you could use [@docusaurus/plugin-client-redirects](#) to create client-side redirects from `/installation.html` to `/installation`.

```
module.exports = {
 plugins: [
 [
 '@docusaurus/plugin-client-redirects',
 {
 fromExtensions: ['html'],
 },
],
],
};
```

If you want to keep the `.html` extension as the canonical URL of a page, docs can declare a `slug`: `installation.html` front matter.

# Components

## Sidebar

In previous version, nested sidebar category is not allowed and sidebar category can only contain doc ID. However, v2 allows infinite nested sidebar and we have many types of [Sidebar Item](#) other than document.

You'll have to migrate your sidebar if it contains category type. Rename `subcategory` to `category` and `ids` to `items`.

`sidebars.json`

```
{
- type: 'subcategory',
+ type: 'category',
 label: 'My Example Subcategory',
+ items: ['doc1'],
- ids: ['doc1']
},
```

## Footer

`website/core/Footer.js` is no longer needed. If you want to modify the default footer provided by Docusaurus, [swizzle](#) it:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
npm run swizzle @docusaurus/theme-classic Footer
```

This will copy the current `<Footer />` component used by the theme to a `src/theme/Footer` directory under the root of your site, you may then edit this component for customization.

Do not swizzle the Footer just to add the logo on the left. The logo is intentionally removed in v2 and moved to the bottom. Just configure the footer in `docusaurus.config.js` with `themeConfig.footer`:

```
module.exports = {
 themeConfig: {
 footer: {
 logo: {
 alt: 'Meta Open Source Logo',
 src: '/img/meta_oss_logo.png',
 href: 'https://opensource.facebook.com',
 },
 },
 },
};
```

## Pages

Please refer to [creating pages](#) to learn how Docusaurus 2 pages work. After reading that, notice that you have to move `pages/en` files in v1 to `src/pages` instead.

In Docusaurus v1, pages received the `siteConfig` object as props.

In Docusaurus v2, get the `siteConfig` object from `useDocusaurusContext` instead.

In v2, you have to apply the theme layout around each page. The Layout component takes metadata props.

`CompLibrary` is deprecated in v2, so you have to write your own React component or use Infima styles (Docs will be available soon, sorry about that! In the meanwhile, inspect the V2 website or view <https://infima.dev/> to see what styles are available).

You can migrate CommonJS to ES6 imports/exports.

Here's a typical Docusaurus v2 page:

```
import React from 'react';
import Link from '@docusaurus/Link';
import useDocusaurusContext from '@docusaurus/useDocusaurusContext';
import Layout from '@theme/Layout';

const MyPage = () => {
 const {siteConfig} = useDocusaurusContext();
 return (
 <Layout title={siteConfig.title} description={siteConfig.tagline}>
 <div className="hero text--center">
 <div className="container">
 <div className="padding-vert--md">
 <h1 className="hero__title">{siteConfig.title}</h1>
```

```

 <p className="hero__subtitle">{siteConfig.tagline}</p>
 </div>
 <div>
 <Link
 to="/docs/get-started"
 className="button button--lg button--outline button--primary">
 Get started
 </Link>
 </div>
</div>
</div>
</Layout>
);
};

export default MyPage;

```

The following code could be helpful for migration of various pages:

- Index page - [Flux](#) (recommended), [Docusaurus 2](#), [Hermes](#)
- Help/Support page - [Docusaurus 2](#), [Flux](#)

## Content

### Replace AUTOGENERATED\_TABLE\_OF\_CONTENTS

This feature is replaced by [inline table of content](#)

### Update Markdown syntax to be MDX-compatible

In Docusaurus 2, the Markdown syntax has been changed to [MDX](#). Hence there might be some broken syntax in the existing docs which you would have to update. A common example is self-closing tags like `<img>` and `<br>` which are valid in HTML would have to be explicitly closed now (`<img/>` and `<br/>`). All tags in MDX documents have to be valid JSX.

Front matter is parsed by [gray-matter](#). If your front matter use special characters like `:`, you now need to quote it: `title: Part 1: my part1 title` → `title: "Part 1: my part1 title"`.

**Tips:** You might want to use some online tools like [HTML to JSX](#) to make the migration easier.

### Language-specific code tabs

Refer to the [multi-language support code blocks](#) section.

## Front matter

The Docusaurus front matter fields for the blog have been changed from camelCase to snake\_case to be consistent with the docs.

The fields `authorFBID` and `authorTwitter` have been deprecated. They are only used for generating the profile image of the author which can be done via the `authors` field.

## Deployment

The `CNAME` file used by GitHub Pages is not generated anymore, so be sure you have created it in `/static/CNAME` if you use a custom domain.

The blog RSS feed is now hosted at `/blog/rss.xml` instead of `/blog/feed.xml`. You may want to configure server-side redirects so that users' subscriptions keep working.

## Test your site

After migration, your folder structure should look like this:

```
my-project
├── docs
└── website
 ├── blog
 └── src
 ├── css
 │ └── custom.css
 └── pages
 └── index.js
 ├── package.json
 ├── sidebars.json
 └── .gitignore
 └── docusaurus.config.js
└── static
```

Start the development server and fix any errors:

[npm](#)   [Yarn](#)   [pnpm](#)   [Bun](#)

```
cd website
npm start
```

You can also try to build the site for production:

**npm**    **Yarn**    **pnpm**    **Bun**

---

```
npm run build
```

# Versioned sites

Read up <https://docusaurus.io/blog/2018/09/11/Towards-Docusaurus-2#versioning> first for problems in v1's approach.

 NOTE

The versioned docs should normally be migrated correctly by the [migration CLI](#)

## Migrate your `versioned_docs` front matter

Unlike v1, The Markdown header for each versioned doc is no longer altered by using `version-${version}-${original_id}` as the value for the actual ID field. See scenario below for better explanation.

For example, if you have a `docs/hello.md`.

```

id: hello
title: Hello, World !

Hi, Endilie here :)
```

When you cut a new version 1.0.0, in Docusaurus v1, `website/versioned_docs/version-1.0.0/hello.md` looks like this:

```

id: version-1.0.0-hello
title: Hello, World !
original_id: hello

Hi, Endilie here :)
```

In comparison, Docusaurus 2 `website/versioned_docs/version-1.0.0/hello.md` looks like this (exactly same as original)

```

id: hello
```

```
title: Hello, World !
```

```

```

```
Hi, Endilie here :)
```

Since we're going for snapshot and allow people to move (and edit) docs easily inside version. The `id` front matter is no longer altered and will remain the same. Internally, it is set as `version-${version}/${id}`.

Essentially, here are the necessary changes in each `versioned_docs` file:

```

```

```
- id: version-1.0.0-hello
+ id: hello
title: Hello, World !
- original_id: hello

```

```
Hi, Endilie here :)
```

## Migrate your `versioned_sidebars`

- Refer to `versioned_docs` ID as `version-${version}/${id}` (v2) instead of `version-${version}-${original_id}` (v1).

Because in v1 there is a good chance someone created a new file with front matter ID "`version-${version}-${id}`" that can conflict with `versioned_docs` ID.

For example, Docusaurus 1 can't differentiate `docs/xxx.md`

```

```

```
id: version-1.0.0-hello

```

```
Another content
```

vs `website/versioned_docs/version-1.0.0/hello.md`

```

```

```
id: version-1.0.0-hello
title: Hello, World !
original_id: hello

```

Hi, Endilie here :)

Since we don't allow `/` in v1 & v2 for front matter, conflicts are less likely to occur.

So v1 users need to migrate their `versioned_sidebar`s file

Example `versioned_sidebar/version-1.0.0-sidebar.json`:

`versioned_sidebar/version-1.0.0-sidebar.json`

```
{
+ "version-1.0.0/docs": {
- "version-1.0.0-docs": {
 "Test": [
+ "version-1.0.0/foo/bar",
- "version-1.0.0-foo/bar",
],
 "Guides": [
+ "version-1.0.0/hello",
- "version-1.0.0-hello"
]
}
}
```

## Populate your `versioned_sidebar`s and `versioned_docs`

In v2, we use snapshot approach for documentation versioning. **Every versioned docs does not depends on other version**. It is possible to have `foo.md` in `version-1.0.0` but it doesn't exist in `version-1.2.0`. This is not possible in previous version due to Docusaurus v1 fallback functionality (<https://v1.docusaurus.io/docs/en/versioning#fallback-functionality>).

For example, if your `versions.json` looks like this in v1

`versions.json`

```
["1.1.0", "1.0.0"]
```

Docusaurus v1 creates versioned docs **if and only if the doc content is different**. Your docs structure might look like this if the only doc changed from v1.0.0 to v1.1.0 is `hello.md`.

```
website
├── versioned_docs
| ├── version-1.1.0
| | └── hello.md
| └── version-1.0.0
| ├── foo
| | └── bar.md
| └── hello.md
└── versioned_sidebars
 └── version-1.0.0-sidebars.json
```

In v2, you have to populate the missing `versioned_docs` and `versioned_sidebars` (with the right front matter and ID reference too).

```
website
├── versioned_docs
| ├── version-1.1.0
| | ├── foo
| | | └── bar.md
| | └── hello.md
| └── version-1.0.0
| ├── foo
| | └── bar.md
| └── hello.md
└── versioned_sidebars
 ├── version-1.1.0-sidebars.json
 └── version-1.0.0-sidebars.json
```

## Convert style attributes to style objects in MDX

Docusaurus 2 uses JSX for doc files. If you have any style attributes in your Docusaurus 1 docs, convert them to style objects, like this:

```

id: demo
title: Demo

Section

hello world
```

```
- <pre style="background: black">zzz</pre>
+ <pre style={{background: 'black'}}>zzz</pre>
```

# Translated sites

This page explains how migrate a translated Docusaurus v1 site to Docusaurus v2.

## i18n differences

Docusaurus v2 i18n is conceptually quite similar to Docusaurus v1 i18n with a few differences.

It is not tightly coupled to Crowdin, and you can use Git or another SaaS instead.

### Different filesystem paths

On Docusaurus v2, localized content is generally found at `website/i18n/[locale]`.

Docusaurus v2 is modular based on a plugin system, and each plugin is responsible to manage its own translations.

Each plugin has its own i18n subfolder, like: `website/i18n/fr/docusaurus-plugin-content-blog`

### Updated translation APIs

With Docusaurus v1, you translate your pages with `<translate>`:

```
const translate = require('../server/translate.js').translate;

<h2>
 <translate desc="the header description">
 This header will be translated
 </translate>
</h2>;
```

On Docusaurus v2, you translate your pages with `<Translate>`

```
import Translate from '@docusaurus/Translate';

<h2>
 <Translate id="header.translation.id" description="the header description">
 This header will be translated
 </Translate>
</h2>;
```

## NOTE

The `write-translations` CLI still works to extract translations from your code.

The code translations are now added to `i18n/[locale]/code.json` using Chrome i18n JSON format.

## Stricter Markdown parser

Docusaurus v2 is using [MDX](#) to parse Markdown files.

MDX compiles Markdown files to React components, is stricter than the Docusaurus v1 parser, and will make your build fail on error instead of rendering some bad content.

Also, the HTML elements must be replaced by JSX elements.

This is particularly important for i18n because if your translations are not good on Crowdin and use invalid Markup, your v2 translated site might fail to build: you may need to do some translation cleanup to fix the errors.

## Migration strategies

This section will help you figure out how to **keep your existing v1 translations after you migrate to v2**.

There are **multiple possible strategies** to migrate a Docusaurus v1 site using Crowdin, with different tradeoffs.

## WARNING

This documentation is a best-effort to help you migrate, please help us improve it if you find a better way!

Before all, we recommend to:

- Migrate your v1 Docusaurus site to v2 without the translations
- Get familiar with the [new i18n system of Docusaurus v2](#) and the [Crowdin tutorial](#)
- Make Crowdin work for your v2 site, using a new and untranslated Crowdin project and the [Crowdin tutorial](#)

## DANGER

Don't try to migrate without understanding both Crowdin and Docusaurus v2 i18n.

# Create a new Crowdin project

To avoid any **risk of breaking your v1 site in production**, one possible strategy is to duplicate the original v1 Crowdin project.

## ! INFO

This strategy was used to [upgrade the Jest website](#).

Unfortunately, Crowdin does not have any "Duplicate/clone Project" feature, which makes things complicated.

- Download the translation memory of your original project in `.tmx` format  
([https://crowdin.com/project/<ORIGINAL\\_PROJECT>/settings#tm](https://crowdin.com/project/<ORIGINAL_PROJECT>/settings#tm) > View Records)
- Upload the translation memory to your new project  
([https://crowdin.com/project/<NEW\\_PROJECT>/settings#tm](https://crowdin.com/project/<NEW_PROJECT>/settings#tm) > View Records)
- Reconfigure `crowdin.yml` for Docusaurus v2 according to the i18n docs
- Upload the Docusaurus v2 source files with the Crowdin CLI to the new project
- Mark sensitive strings like `id` or `slug` as "hidden string" on Crowdin
- On the "Translations" tab, click on "Pre-Translation > via TM"  
([https://crowdin.com/project/<NEW\\_PROJECT>/settings#translations](https://crowdin.com/project/<NEW_PROJECT>/settings#translations))
- Try first with "100% match" (more content will be translated than "Perfect"), and pre-translate your sources
- Download the Crowdin translations locally
- Try to run/build your site and see if there are any errors

You will likely have errors on your first-try: the pre-translation might try to translate things that it should not be translated (front matter, admonition, code blocks...), and the translated MD files might be invalid for the MDX parser.

You will have to fix all the errors until your site builds. You can do that by modifying the translated MD files locally, and fix your site for one locale at a time using `docusaurus build --locale fr`.

There is no ultimate guide we could write to fix these errors, but common errors are due to:

- Not marking enough strings as "hidden strings" in Crowdin, leading to pre-translation trying to translate these strings.
- Having bad v1 translations, leading to invalid markup in v2: bad HTML elements inside translations and unclosed tags
- Anything rejected by the MDX parser, like using HTML elements instead of JSX elements (use the [MDX playground](#) for debugging)

You might want to repeat this pre-translation process, eventually trying the "Perfect" option and limiting pre-translation only some languages/files.

### TIP

Use `mdx-code-block` around problematic Markdown elements: Crowdin is less likely mess things up with code blocks.

### NOTE

You will likely notice that some things were translated on your old project, but are now untranslated in your new project.

The Crowdin Markdown parser is evolving other time and each Crowdin project has a different parser version, which can lead to pre-translation not being able to pre-translate all the strings.

This parser version is undocumented, and you will have to ask the Crowdin support to know your project's parser version and fix one specific version.

Using the same CLI version and parser version across the 2 Crowdin projects might give better results.

### DANGER

Crowdin has an "upload translations" feature, but in our experience it does not give very good results for Markdown

## Use the existing Crowdin project

If you don't mind modifying your existing Crowdin project and risking to mess things up, it may be possible to use the Crowdin branch system.

### WARNING

This workflow has not been tested in practice, please report us how good it is.

This way, you wouldn't need to create a new Crowdin project, transfer the translation memory, apply pre-translations, and try to fix the pre-translations errors.

You could create a Crowdin branch for Docusaurus v2, where you upload the v2 sources, and merge the Crowdin branch to main once ready.

## Use Git instead of Crowdin

It is possible to migrate away of Crowdin, and add the translation files to Git instead.

Use the Crowdin CLI to download the v1 translated files, and put these translated files at the correct Docusaurus v2 filesystem location.