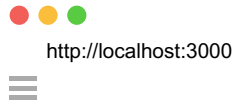
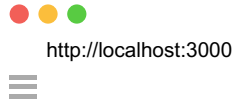
On this page

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>;
}
```
```



/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

```
module.exports = {
  themeConfig: {
    prism: {
      theme: require('prism-react-renderer/themes/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.

CAUTION

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
```
module.exports = {
  // ...
  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

```
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.

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';
}
```
````


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

  return 'Nothing highlighted';
}

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

  return 'Nothing highlighted';
}
```

Supported commenting syntax:

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

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

```css
:root {
  --docusaurus-highlighted-code-line-bg: rgb(72, 77, 91);
}

/* 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;
```
````


```jsx
import React from 'react';

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

  return <div>Foo</div>;
}

export default MyComponent;
```

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.

````
- ```jsx {3}
+ ```jsx {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
````

```
module.exports = {
  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

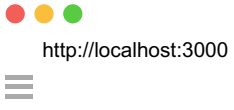
```
npm run swizzle @docusaurus/theme-classic CodeBlock/Line
```

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

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

  return <div>Foo</div>;
}

export default MyComponent;
```
````



```jsx
import React from 'react';

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

  return <div>Foo</div>;
}

export default MyComponent;
```

Interactive code editor

(Powered by React Live)

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

npm
Yarn
pnpm

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

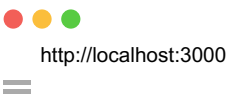
You will also need to add the plugin to your `docusaurus.config.js`.

```
module.exports = {
  // ...
  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.



LIVE EDITOR

```
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>
  );
}
```

RESULT

Loading...

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

```
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:

http://localhost:3000

LIVE EDITOR

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

RESULT

Loading...

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.

LIVE EDITOR

```
const project = "Docusaurus";

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

render(
  <Greeting />
);
```

RESULT

Loading...

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>
```

Input: 1 2 3 4
Output: "366300745"

⚠ 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.

⚠ CAUTION

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

Multi-language support code blocks

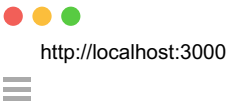
With MDX, you can easily create interactive components within your documentation, for example, to display code in multiple programming languages and switch between them using a tabs component.

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

```
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
```
module.exports = {
  // ...
  presets: [
    [
      '@docusaurus/preset-classic',
      {
        docs: {
          remarkPlugins: [
            [require('@docusaurus/remark-plugin-npm2yarn'), {sync: true}],
          ],
        },
        pages: {
          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
`sync`	`boolean`	`false`	Whether to sync the selected converter across all code blocks.
`converters`	`array`	`'yarn'`, `'pnpm'`	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>
  );
}
```

/src/components/HelloCodeTitle.js

```
function HelloCodeTitle(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

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.

Edit this page

*Last updated on **Mar 23, 2023** by **Sébastien Lorber***

Previous
« Tabs

Next
Admonitions »

Legal

Privacy

Terms

Data Policy

Cookie Policy