# The #[doc] attribute

The `#[doc]` attribute lets you control various aspects of how `rustdoc` does its job.

The most basic function of `#[doc]` is to handle the actual documentation text. That is, `///` is syntax sugar for `#[doc]`. This means that these two are the same:

```
/// This is a doc comment.
#[doc = " This is a doc comment."]
# fn f() {}
```

(Note the leading space in the attribute version.)

In most cases, `///` is easier to use than `#[doc]`. One case where the latter is easier is when generating documentation in macros; the `collapse-docs` pass will combine multiple `#[doc]` attributes into a single doc comment, letting you generate code like this:

```
#[doc = "This is"]
#[doc = " a "]
#[doc = "doc comment"]
# fn f() {}
```

Which can feel more flexible. Note that this would generate this:

```
#[doc = "This is\n a \ndoc comment"]
# fn f() {}
```

but given that docs are rendered via Markdown, it will remove these newlines.

Another use case is for including external files as documentation:

```
#[doc = include_str!("../../README.md")]
# fn f() {}
```

The `doc` attribute has more options though! These don't involve the text of the output, but instead, various aspects of the presentation of the output. We've split them into two kinds below: attributes that are useful at the crate level, and ones that are useful at the item level.

## At the crate level

These options control how the docs look at a crate level.

### `html_favicon_url`

This form of the `doc` attribute lets you control the favicon of your docs.

```
#![doc(html_favicon_url = "https://example.com/favicon.ico")]
```

This will put `<link rel="icon" href="{}">` into your docs, where the string for the attribute goes into the `{}`.

If you don't use this attribute, there will be no favicon.

### `html_logo_url`

This form of the `doc` attribute lets you control the logo in the upper left hand side of the docs.

```
#![doc(html_logo_url = "https://example.com/logo.jpg")]
```

This will put `<a href='../index.html'><img src='{}' alt='logo' width='100'></a>` into your docs, where the string for the attribute goes into the `{}`.

If you don't use this attribute, there will be no logo.

### `html_playground_url`

This form of the `doc` attribute lets you control where the "run" buttons on your documentation examples make requests to.

```
#![doc(html_playground_url = "https://playground.example.com/")]
```

Now, when you press "run", the button will make a request to this domain.

If you don't use this attribute, there will be no run buttons.

### `issue_tracker_base_url`

This form of the `doc` attribute is mostly only useful for the standard library; When a feature is unstable, an issue number for tracking the feature must be given. `rustdoc` uses this number, plus the base URL given here, to link to the tracking issue.

```
#![doc(issue_tracker_base_url = "https://github.com/rust-lang/rust/issues/")]
```

### `html_root_url`

The `#[doc(html_root_url = "...")]` attribute value indicates the URL for generating links to external crates. When rustdoc needs to generate a link to an item in an external crate, it will first check if the extern crate has been documented locally on-disk, and if so link directly to it. Failing that, it will use the URL given by the `--extern-html-root-url` command-line flag if available. If that is not available, then it will use the `html_root_url` value in the extern crate if it is available. If that is not available, then the extern items will not be linked.

```
#![doc(html_root_url = "https://docs.rs/serde/1.0")]
```

### `html_no_source`

By default, `rustdoc` will include the source code of your program, with links to it in the docs. But if you include this:

```
#![doc(html_no_source)]
```

it will not.

### `test(no_crate_inject)`

By default, `rustdoc` will automatically add a line with `extern crate my_crate;` into each doctest. But if you include this:

```
#![doc(test(no_crate_inject))]
```

it will not.

### `test(attr(...))`

This form of the `doc` attribute allows you to add arbitrary attributes to all your doctests. For example, if you want your doctests to fail if they produce any warnings, you could add this:

```
#![doc(test(attr(deny(warnings))))]
```

## At the item level

These forms of the `#[doc]` attribute are used on individual items, to control how they are documented.

### `inline` and `no_inline`

These attributes are used on `use` statements, and control where the documentation shows up. For example, consider this Rust code:

```
pub use bar::Bar;

/// bar docs
pub mod bar {
    /// the docs for Bar
    pub struct Bar;
}
# fn main() {}
```

The documentation will generate a "Re-exports" section, and say `pub use bar::Bar;`, where `Bar` is a link to its page.

If we change the `use` line like this:

```
#[doc(inline)]
pub use bar::Bar;
# pub mod bar { pub struct Bar; }
# fn main() {}
```

Instead, `Bar` will appear in a `Structs` section, just like `Bar` was defined at the top level, rather than `pub use`'d.

Let's change our original example, by making `bar` private:

```
pub use bar::Bar;

/// bar docs
mod bar {
    /// the docs for Bar
    pub struct Bar;
}
# fn main() {}
```

Here, because `bar` is not public, `Bar` wouldn't have its own page, so there's nowhere to link to. `rustdoc` will inline these definitions, and so we end up in the same case as the `#[doc(inline)]` above; `Bar` is in a `Structs` section, as if it were defined at the top level. If we add the `no_inline` form of the attribute:

```
#[doc(no_inline)]
pub use bar::Bar;

/// bar docs
mod bar {
    /// the docs for Bar
    pub struct Bar;
}
# fn main() {}
```

Now we'll have a `Re-exports` line, and `Bar` will not link to anywhere.

One special case: In Rust 2018 and later, if you `pub use` one of your dependencies, `rustdoc` will not eagerly inline it as a module unless you add `#[doc(inline)]`.

### `hidden`

Any item annotated with `#[doc(hidden)]` will not appear in the documentation, unless the `strip-hidden` pass is removed.

### `alias`

This attribute adds an alias in the search index.

Let's take an example:

```
#[doc(alias = "TheAlias")]
pub struct SomeType;
```

So now, if you enter "TheAlias" in the search, it'll display `SomeType`. Of course, if you enter `SomeType` it'll return `SomeType` as expected!

**FFI example** This doc attribute is especially useful when writing bindings for a C library. For example, let's say we have a C function that looks like this:

```
int lib_name_do_something(Obj *obj);
```

It takes a pointer to an `Obj` type and returns an integer. In Rust, it might be written like this:

"'ignore (using non-existing ffi types) pub struct Obj { inner: *mut ffi::Obj, }

impl Obj { pub fn do_something(&mut self) -> i32 { unsafe { ffi::lib_name_do_something(self.inner) } } } "'

The function has been turned into a method to make it more convenient to use. However, if you want to look for the Rust equivalent of `lib_name_do_something`, you have no way to do so.

To get around this limitation, we just add `#[doc(alias = "lib_name_do_something")]` on the `do_something` method and then it's all good! Users can now look for `lib_name_do_something` in our crate directly and find `Obj::do_something`.