What to include (and exclude)

It is easy to say everything must be documented in a project and often times that is correct, but how can we get there, and are there things that don't belong?

At the top of the `src/lib.rs` or `main.rs` file in your binary project, include the following attribute:

```
#![warn(missing_docs)]
```

Now run `cargo doc` and examine the output. Here's a sample:

```
 Documenting docdemo v0.1.0 (/Users/username/docdemo)
warning: missing documentation for the crate
 --> src/main.rs:1:1
  |
1 | / #![warn(missing_docs)]
2 | |
3 | | fn main() {
4 | |     println!("Hello, world!");
5 | | }
  | |_^
  |
note: the lint level is defined here
 --> src/main.rs:1:9
  |
1 | #![warn(missing_docs)]
  |         ^^^^^^^^^^^^

warning: 1 warning emitted

    Finished dev [unoptimized + debuginfo] target(s) in 2.96s
```

As a library author, adding the lint `#![deny(missing_docs)]` is a great way to ensure the project does not drift away from being documented well, and `#![warn(missing_docs)]` is a good way to move towards comprehensive documentation. In addition to docs, `#![deny(missing_doc_code_examples)]` ensures each function contains a usage example. In our example above, the warning is resolved by adding crate level documentation.

There are more lints in the upcoming chapter Lints.

Examples

Of course this is contrived to be simple, but part of the power of documentation is showing code that is easy to follow, rather than being realistic. Docs often take shortcuts with error handling because examples can become complicated to follow with all the necessary set up required for a simple example.

`Async` is a good example of this. In order to execute an `async` example, an executor needs to be available. Examples will often shortcut this, and leave users to figure out how to put the `async` code into their own runtime.

It is preferred that `unwrap()` not be used inside an example, and some of the error handling components be hidden if they make the example too difficult to follow.

```
/// Example
/// ```rust
/// let fourtytwo = "42".parse::<u32>()?;
/// println!("{} + 10 = {}", fourtytwo, fourtytwo+10);
/// ```
```

When rustdoc wraps that in a main function, it will fail to compile because the `ParseIntError` trait is not implemented. In order to help both your audience and your test suite, this example needs some additional code:

```
/// Example
/// ```rust
/// # main() -> Result<(), std::num::ParseIntError> {
/// let fortytwo = "42".parse::<u32>()?;
/// println!("{} + 10 = {}", fortytwo, fortytwo+10);
/// #     Ok(())
/// # }
/// ```
```

The example is the same on the doc page, but has that extra information available to anyone trying to use your crate. More about tests in the upcoming Documentation tests chapter.

What to Exclude

Certain parts of your public interface may be included by default in the output of rustdoc. The attribute `#[doc(hidden)]` can hide implementation details to encourage idiomatic use of the crate.

For example, an internal `macro!` that makes the crate easier to implement can become a footgun for users when it appears in the public documentation. An internal `Error` type may exist, and `impl` details should be hidden, as detailed in the API Guidelines.

Customizing the output

It is possible to pass a custom css file to `rustdoc` and style the documentation.

```
rustdoc --extend-css custom.css src/lib.rs
```

A good example of using this feature to create a dark theme is documented on this blog. Just remember, dark theme is already included in the rustdoc output by clicking on the paintbrush. Adding additional options to the themes are as easy as creating a custom theme `.css` file and using the following syntax:

```
rustdoc --theme awesome.css src/lib.rs
```

Here is an example of a new theme, Ayu.