Angular documentation style guide

This style guide covers the standards for writing Angular documentation on angular.io. These standards ensure consistency in writing style, Markdown conventions, and code snippets.

Prerequisites

Before contributing to the Angular documentation, it is helpful if you are familiar with the following:

- `git` : for an introduction, see GitHub's Git Handbook
- `GitHub` : for an introduction, see GitHub's Hello World
- Markdown: see GitHub's Mastering Markdown
- Angular coding style: see the Angular Style Guide
- Google writing style: The Google Developer Documentation Style Guide is a comprehensive resource that this Angular documentation style guide builds upon.

Kinds of Angular documentation

The categories of Angular documentation include:

- Guides: much of what's in the documentation section of angular.io. Guides walk the reader step-by-step through tasks to demonstrate concepts and are often accompanied by a working example. These include Getting Started, Tour of Heroes, and pages about Forms, Dependency Injection, and HttpClient. Contributing members of the community and Angular team members maintain this documentation in Markdown.
- API documentation: reference documents for the Angular Application Programming Interface, or API. These are more succinct than guides and serve as a reference for Angular features. They are especially helpful for people already acquainted with Angular concepts. The angular.io infrastructure generates these documents from source code and comments that contributors edit.
- CLI documentation: the angular.io infrastructure generates these documents from CLI source code.

Markdown and HTML

While the Angular guides are Markdown files, there are some sections within the guides that use HTML.

To enable HTML in an Angular guide, **always** follow every opening and closing HTML tag with a blank line.

Notice the required blank line after the opening `<div>` in the following example:

```
<div class="alert is-helpful">

  **Always** follow every opening and closing HTML tag with _a blank line_.

</div>
```

It is customary but not required to precede the closing HTML tag with a blank line as well.

Title

Every guide document must have a title, and it should appear at the top of the page.

Begin the title with the Markdown `#` character, which renders as an `<h1>` in the browser.

```
# Angular documentation style guide
```

A document can have only one `<h1>`.

Title text should be in *Sentence case*, which means the first word is capitalized and all other words are lower case. Technical terms that are always capitalized, like "Angular", are the exception.

```
# Deprecation policy in Angular
```

Always follow the title with at least one blank line.

The corresponding text in the left nav is in *Title Case*, which means that you use capital letters to start the first words and all principal words. Use lower case letters for secondary words such as "in", "of", and "the". You can also shorten the nav title to fit in the column.

Sections

A typical document has sections.

All section headings are in *Sentence case*, which means the first word is capitalized and all other words are lower case.

Always follow a section heading with at least one blank line.

Main section heading

There are usually one or more main sections that may be further divided into secondary sections.

Begin a main section heading with the Markdown `##` characters, which renders as an `<h2>` in the browser.

Follow main section headings with a blank line and then the content for that heading as in the following example:

```
## Main section heading

Content after a blank line.
```

Secondary section heading

A secondary section heading is related to a main heading and falls textually within the bounds of that main heading.

Begin a secondary heading with the Markdown `###` characters, which renders as an `<h3>` in the browser.

Follow a secondary heading by a blank line and then the content for that heading as in the following example:

```
### Secondary section heading

Content after a blank line.
```

Additional section headings

While you can use additional section headings, the Table-of-contents (TOC) generator only shows `<h2>` and `<h3>` headings in the TOC on the right of the page.

```
#### The TOC won't display this

Content after a blank line.
```

Table of contents

Most pages display a table of contents or TOC. The TOC appears in the right panel when the viewport is wide. When narrow, the TOC appears in a collapsible region near the top of the page.

You don't need to create your own TOC by hand because the TOC generator creates one automatically from the page's `<h2>` and `<h3>` headers.

To exclude a heading from the TOC, create the heading as an `<h2>` or `<h3>` element with a class called 'no-toc'.

```
<h3 class="no-toc">

This heading is not displayed in the TOC

</h3>
```

You can turn off TOC generation for the entire page by writing the title with an `<h1>` tag and the `no-toc` class.

```
<h1 class="no-toc">

A guide without a TOC

</h1>
```

Navigation

To generate the navigation links at the top, left, and bottom of the screen, use the JSON configuration file, `content/navigation.json`.

If you have an idea that would result in navigation changes, file an issue first so that the Angular team and community can discuss the change.

For a new guide page, edit the `SideNav` node in `navigation.json`. The `SideNav` node is an array of navigation nodes. Each node is either an item node for a single document or a header node with child nodes.

Find the header for your page. For example, a guide page that describes an Angular feature is probably a child of the `Fundamentals` header.

```
{
  "title": "Fundamentals",
  "tooltip": "The fundamentals of Angular",
  "children": [ ... ]
}
```

A header node child can be an item node or another header node. If your guide page belongs under a sub-header, find that sub-header in the JSON.

Add an item node for your guide page as a child of the appropriate header node as in the following example:

```
{
  "url": "guide/docs-style-guide",
  "title": "Doc authors style guide",
  "tooltip": "Style guide for documentation authors.",
},
```

A navigation node has the following properties:

- `url` : the URL of the guide page, which is an item node only.
- `title` : the text displayed in the side nav.
- `tooltip` : text that appears when the reader hovers over the navigation link.
- `children` : an array of child nodes, which is a header node only.
- `hidden` : defined and set `true` if this is a guide page that should not be displayed in the navigation panel.

Do not create a node that is both a header and an item node by specifying the `url` property of a header node.

Code snippets

Angular.io has a custom framework that enables authors to include code snippets directly from working example applications that are automatically tested as part of documentation builds.

In addition to working code snippets, example code can include terminal commands, a fragment of TypeScript or HTML, or an entire code file.

Whatever the source, the document viewer renders them as code snippets, either individually with the code-example component or as a tabbed collection with the code-tabs component.

{@a code-example}

When to use code font

You can display a minimal, inline code snippet with the Markdown backtick syntax. Use a single backtick on either side of a term when referring to code or the name of a file in a sentence. The following are some examples:

- In the `app.component.ts` , add a `logger()` method.
- The `name` property is `Sally` .
- Add the component class name to the `declarations` array.

The Markdown is as follows:

```
* In the `app.component.ts`, add a `logger()` method.
* The <code class="no-auto-link">item</code> property is `true`.
* Add the component class name to the `declarations` array.
```

Auto-linking in code snippets

In certain cases, when you apply backticks around a term, it may auto-link to the API documentation. If you do not intend the term to be a link, use the following syntax:

```
The <code class="no-auto-link">item</code> property is `true`.
```

Hard-coded snippets

Ideally, you should source code snippets from working sample code, though there are times when an inline snippet is necessary.

For terminal input and output, place the content between `<code-example>` tags and set the language attribute to `sh` as in this example:

npm start

```
<code-example language="sh">

  npm start

</code-example>
```

Inline, hard-coded snippets like this one are not testable and, therefore, intrinsically unreliable. This example belongs to the small set of pre-approved, inline snippets that includes user input in a command shell or the output of some process.

In all other cases, code snippets should be generated automatically from tested code samples.

For hypothetical examples such as illustrations of configuration options in a JSON file, use the `<code-example>` tag with the `header` attribute to identify the context.

{@a from-code-samples}

Compilable example apps

One of the Angular documentation design goals is that guide page code snippets be examples of working code.

Authors meet this goal by displaying code snippets directly from working sample applications, written specifically for these guide pages.

Find sample applications in sub-folders of the `content/examples` directory of the `angular/angular` repository. An example folder name is often the same as the guide page it supports.

A guide page might not have its own sample code. It might refer instead to a sample belonging to another page.

The Angular CI process runs all end-to-end tests for every Angular PR. Angular re-tests the samples after every new version of a sample and every new version of Angular.

When possible, every snippet of code on a guide page should be derived from a code sample file. You tell the Angular documentation engine which code file—or fragment of a code file—to display by configuring `<code-example>` attributes.

{@a display-whole-file}

Displaying an entire code file

This Angular documentation style guide that you are currently reading has its own example application, located in the `content/examples/docs-style-guide` folder.

The following `<code-example>` displays the sample's `app.module.ts` :

The following markup produces that snippet:

```
<code-example path="docs-style-guide/src/app/app.module.ts"
header="src/app/app.module.ts">
</code-example>
```

The `path` attribute identifies the snippet's source file at the example application folder's location within `content/examples` . In this example, that path is `docs-style-guide/src/app/app.module.ts` .

The header tells the reader where to find the file. Following convention, set the `header` attribute to the file's location within the example application's root folder.

Unless otherwise noted, all code snippets in this page are from sample source code located in the `content/examples/docs-style-guide` directory.

The documentation tooling reports an error if the file identified in the path does not exist or is in the `.git-ignore` file. Most `.js` files are in `.git-ignore` .

To include an ignored code file in your project and display it in a guide, remove it from `.git-ignore` . Update the `content/examples/.gitignore` as follows:

my-guide !my-guide/src/something.js !my-guide/more-javascript*.js
{@a region}

Displaying part of a code file

To include a snippet of code within a sample code file, rather than the entire file, use the `<code-example>` `region` attribute. The following example focuses on the `AppModule` class and its `@NgModule()` metadata:

To render the above example, the HTML in the Markdown file is as follows:

```
<code-example
  path="docs-style-guide/src/app/app.module.ts"
  region="class">
</code-example>
```

The `path` points to the file, just as in examples that render the entire file. The `region` attribute specifies a portion of the source file delineated by an opening `#docregion` and a closing `#enddocregion` .

You can see the `class` `#docregion` in the source file below. Notice the commented lines `#docregion` and `#enddocregion` in `content/examples/docs-style-guide/src/app/app.module.ts` with the name `class` .

import { NgModule } from '@angular/core'; import { BrowserModule } from '@angular/platform-browser'; import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

// #docregion class @NgModule({ imports: [BrowserModule, FormsModule], declarations: [AppComponent], bootstrap: [AppComponent] }) export class AppModule { } // #enddocregion class

The opening and ending `#docregion` lines designate any lines of code between them as being included in the code snippet. This is why the import statements outside of the `class` `#docregion` are not in the code snippet.

For more information on how to prepare example application files for use in guides, see Source code markup.

Code snippet options

Specify the `<code-example>` output with the following attributes:

- `path` : the path to the file in the `content/examples` folder.
- `header` : the header of the code listing. This is the title of the code snippet and can include the path and extra information such as whether the snippet is an excerpt.
- `region` : displays the source file fragment with that region name; regions are identified by `#docregion` markup in the source file. See Displaying a code snippet.
- `linenums` : value may be `true` , `false` , or a `number` . The default is `false` , which means that the browser displays no line numbers. The `number` option starts line numbering at the given value. For example, `linenums=4` sets the starting line number to 4.
- `class` : code snippets can be styled with the CSS classes `no-box` and `avoid` .
- `hideCopy` : hides the copy button.
- `language` : the source code language such as `javascript` , `html` , `css` , `typescript` , `json` , or `sh` . This attribute only applies to hard-coded examples.

Displaying bad code

Occasionally, you want to display an example of less than ideal code or design, such as with **avoid** examples in the Angular Style Guide. Because it is possible for readers to copy and paste examples of inferior code in their own applications, try to minimize use of such code.

In cases where you need unacceptable examples, you can set the `class` to `avoid` or have the word `avoid` in the filename of the source file. By putting the word `avoid` in the filename or path, the documentation generator automatically adds the `avoid` class to the `<code-example>` . Either of these options frames the code snippet in bright red to grab the reader's attention.

Here's the markup for an "avoid" example in the Angular Style Guide that uses the word `avoid` in the path name:

```
<code-example
  path="styleguide/src/05-03/app/heroes/shared/hero-button/hero-
button.component.avoid.ts"
  region="example"
```

```
  header="app/heroes/hero-button/hero-button.component.ts">
</code-example>
```

Having the word "avoid" in the file name causes the browser to render the code snippet with a red header and border:

Alternatively, the HTML could include the `avoid` class as in the following:

```
<code-example
  path="docs-style-guide/src/app/not-great.component.ts"
  header="docs-style-guide/src/app/not-great.component.ts"
  region="not-great"
  class="avoid">
</code-example>
```

Explicitly applying the class `avoid` causes the same result of a red header and red border:

{@a code-tabs}

Code Tabs

Code tabs display code much like `code-examples` with the added advantage of displaying multiple code samples within a tabbed interface. Each tab displays code using a `code-pane`.

`code-tabs` attributes

- `linenums`: The value can be `true`, `false`, or a number indicating the starting line number. The default is `false`.

`code-pane` attributes

- `path`: a file in the `content/examples` folder
- `header`: what displays in the header of a tab
- `linenums`: overrides the `linenums` property at the `code-tabs` level for this particular pane. The value can be `true`, `false`, or a number indicating the starting line number. The default is `false`.

The following example displays multiple code tabs, each with its own header. It demonstrates showing line numbers in `<code-tabs>` and `<code-pane>`.

The `linenums` attribute set to `true` on `<code-tabs>` explicitly enables numbering for all panes. However, the `linenums` attribute set to `false` in the second `<code-pane>` disables line numbering only for itself.

```
<code-tabs linenums="true">
  <code-pane
    header="app.component.html"
    path="docs-style-guide/src/app/app.component.html">
  </code-pane>
  <code-pane
    header="app.component.ts"
    path="docs-style-guide/src/app/app.component.ts"
    linenums="false">
  </code-pane>
  <code-pane
```

```
    header="app.component.css (heroes)"
    path="docs-style-guide/src/app/app.component.css"
    region="heroes">
  </code-pane>
  <code-pane
    header="package.json (scripts)"
    path="docs-style-guide/package.1.json">
  </code-pane>
</code-tabs>
```

{@a source-code-markup}

Preparing source code for code snippets

To display `<code-example>` and `<code-tabs>` snippets, add code snippet markup to sample source code files.

The sample source code for this page, located in `content/examples/docs-style-guide`, contains examples of every code snippet markup described in this section.

Code snippet markup is always in the form of a comment. The default `#docregion` markup for a TypeScript or JavaScript file is as follows:

```
// #docregion
... some TypeScript or JavaScript code ...
// #enddocregion
```

```
<!-- #docregion -->
... some HTML ...
<!-- #enddocregion -->
```

```
/* #docregion */
... some CSS ...
/* #enddocregion */
```

The documentation generation process erases these comments before displaying them in the documentation viewer, StackBlitz, and sample code downloads.

Because JSON does not allow comments, code snippet markup doesn't work in JSON files. See the section on JSON files for more information.

`#docregion`

Use `#docregion` in source files to mark code for use in `<code-example>` or `<code-tabs>` components.

The `#docregion` comment begins a code snippet region. Every line of code after that comment belongs in the region until the code fragment processor encounters the end of the file or a closing `#enddocregion`.

The following `src/main.ts` is a an example of a file with a single `#docregion` at the top of the file.

As a result, the entire file is in the `<code-example>`.

Naming a `#docregion`

To display multiple snippets from different fragments within the same file, give each fragment its own `#docregion` name as follows, where `your-region-name` is a hyphenated lowercase string:

```
// #docregion your-region-name
... some code ...
// #enddocregion your-region-name
```

Reference this region by name in the `region` attribute of the `<code-example>` or `<code-pane>` as follows:

```
<code-example
  path="your-example-app/src/app/your-file.ts"
  region="your-region-name"></code-example>
```

Because the `#docregion` with no name is the default region, you do not need to set the `region` attribute when referring to the default `#docregion`.

Nesting a `#docregion`

Place a `#docregion` within another `#docregion` as in the following example with a nested `inner-region`:

```
// #docregion
... some code ...
// #docregion inner-region
... more code ...
// #enddocregion inner-region
... yet more code ...
/// #enddocregion
```

Combining code fragments

Combine several fragments from the same file into a single code snippet by defining multiple `#docregion` sections with the same region name. The following example defines two nested `#docregion` sections.

The inner region, `class-skeleton`, appears twice—once to capture the code that opens the class definition and a second time to capture the code that closes the class definition.

// #docplaster ... // #docregion class, class-skeleton export class AppComponent { // #enddocregion class-skeleton title = 'Authors Style Guide Sample'; heroes = HEROES; selectedHero: Hero;

onSelect(hero: Hero): void { this.selectedHero = hero; } // #docregion class-skeleton } // #enddocregion class, class-skeleton

The `#docplaster` marker tells the processor what text string to use—that is, the "plaster"—to join each of the fragments into a single snippet. Place the "plaster" text on the same line. For example, `#docplaster ---` would use `---` as the "plaster" text. In the case of the previous file, the "plaster" text is empty so there will be nothing in between each fragment.

Without `#docplaster`, the processor inserts the default plaster—an ellipsis comment—between the fragments.

Here are the two corresponding code snippets for side-by-side comparison.

The above example also demonstrates that one `#docregion` or `#enddocregion` comment can specify two region names, which is a convenient way to start or stop multiple regions on the same code line. Alternatively, you could put these comments on separate lines as in the following example:

// #docplaster ... // #docregion class // #docregion class-skeleton export class AppComponent { // #enddocregion class-skeleton title = 'Authors Style Guide Sample'; heroes = HEROES; selectedHero: Hero;

onSelect(hero: Hero): void { this.selectedHero = hero; } // #docregion class-skeleton } // #enddocregion class // #enddocregion class-skeleton

JSON files

The `<code-example>` component cannot display portions of a JSON file because JSON forbids comments. However, you can display an entire JSON file by referencing it in the `<code-example>` `src` attribute.

For large JSON files, you could copy the nodes-of-interest into Markdown backticks, but as it's easy to mistakenly create invalid JSON that way, consider creating a JSON partial file with the fragment you want to display.

You can't test a partial file nor use it in the application, but at least your editor can confirm that it is syntactically correct. You can also store the partial file next to the original, so it is more likely that the author will remember to keep the two in sync.

Here's an example that excerpts certain scripts from `package.json` into a partial file named `package.1.json`.

```
<code-example path="docs-style-guide/package.1.json" header="package.json (selected scripts)"></code-example>
```

In some cases, it is preferable to use the name of the full file rather than the partial. In this case, the full file is `package.json` and the partial file is `package.1.json`. Since the focus is generally on the full file rather than the partial, using the name of the file the reader edits, in this example `package.json`, clarifies which file to work in.

Partial file naming

The step-by-step nature of the guides necessitate refactoring, which means there are code snippets that evolve through a guide.

Use partial files to demonstrate intermediate versions of the final source code with fragments of code that don't appear in the final app. The sample naming convention adds a number before the file extension, as follows:

```
package.1.json
app.component.1.ts
app.component.2.ts
```

Remember to exclude these files from StackBlitz by listing them in the `stackblitz.json` as illustrated here:

Source code styling

Source code should follow Angular's style guide where possible.

Hexadecimals

Hexadecimal should use the shorthand where possible, and use only lowercase letters.

{@a live-examples}

Live examples

Adding `<live-example></live-example>` to a page generates two default links: .

The first is a link to the StackBlitz example, which the default `stackblitz.json` file defines. You can find the `stackblitz.json` file in the `content/examples/example-app` directory, where `example-app` is the sample application folder you're using for the guide.

By default, the documentation generator uses the name of the guide as the name of the example. So, if you're working on `router.md`, and use `<live-example></live-example>` in the document, the documentation generator looks for `content/examples/router`. Clicking this link opens the code sample on StackBlitz in a new browser tab.

The second link downloads the sample app.

Define live examples by one or more `stackblitz.json` files in the root of a code sample folder. Each sample folder usually has a single unnamed definition file, the default `stackblitz.json`.

Live Example for named StackBlitz

You can create additional, named definition files in the form `name.stackblitz.json`. The Testing guide (`aio/content/guide/testing.md`) references a named StackBlitz file as follows:

```
<live-example stackblitz="specs">Tests</live-example>
```

The `stackblitz` attribute value of `specs` refers to the `examples/testing/specs.stackblitz.json` file. If you were to leave out the `stackblitz` attribute, the default would be `examples/testing/stackblitz.json`.

Custom label and tooltip

Change the appearance and behavior of the live example with attributes and classes. The following example gives the live example anchor a custom label and tooltip by setting the `title` attribute:

```
<live-example title="Live Example with title"></live-example>
```

The browser renders the following:

You can achieve the same effect by putting the label between the `<live-example>` tags:

```
<live-example>Live example with content label</live-example>
```

The browser renders the following:

Live example with content label

Live example from another guide

To link to an example in a folder where the name is not the same as the current guide page, set the `name` attribute to the name of that folder.

For example, to include the Router guide example in this style guide, set the `name` attribute to `router`, that is, the name of the folder where that example resides.

```
<live-example name="router">Live example from the Router guide</live-example>
```

Live example from the Router guide

Live Example without download

To omit the download link, add the `noDownload` attribute.

```
<live-example noDownload>Just the StackBlitz</live-example>
```

The browser renders the following:

Just the StackBlitz

Live Example with download-only

To omit the live StackBlitz link and only link to the download, add the `downloadOnly` attribute.

```
<live-example downloadOnly>Download only</live-example>
```

The browser renders the following:

Download only

Embedded live example

By default, a live example link opens a StackBlitz example in a separate browser tab. You can embed the StackBlitz example within the guide page by adding the `embedded` attribute.

For performance reasons, StackBlitz does not start right away. Instead, the `<live-example>` component renders an image. Clicking the image starts the process of launching the embedded StackBlitz within an `<iframe>`.

The following is an embedded `<live-example>` for this guide:

```
<live-example embedded></live-example>
```

The browser renders the following `<iframe>` and a `<p>` with a link to download the example:

{@a anchors}

Anchors

Every section header is also an anchor point. Another guide page could add a link to this "Anchors" section with the following:

```
<div class="alert is-helpful">

See the ["Anchors"](guide/docs-style-guide#anchors "Style Guide&mdash;Anchors")
```

```
section for details.

</div>
```

The browser renders the following:

See the "Anchors" section for details.

Notice that the above example includes a title of "Style Guide—Anchors". Use titles on anchors to create tooltips and improve UX.

When navigating within a page, you can omit the page URL when specifying the link that scrolls up to the beginning of this section, as in the following:

```
... the link that [scrolls up](#anchors "Anchors") to ...
```

{@a section-anchors}

Section header anchors

While the documentation generator automatically creates anchors for headers based on the header wording, titles can change, which can potentially break any links to that section.

To mitigate link breakage, add a custom anchor explicitly, just above the heading or text to which it applies, using the special `{@a name}` syntax as follows:

{@a section-anchors}

Section header anchors

Then reference that anchor like this:

```
This is a [link to that custom anchor name](#section-anchors).
```

The browser renders the following:

This is a link to that custom anchor name.

When editing a file, don't remove any anchors. If you change the document structure, you can move an existing anchor within that same document without breaking a link. You can also add more anchors with more appropriate text.

As an alternative, you can use the HTML `<a>` tag. When using the `<a>` tag, set the `id` attribute—rather than the `name` attribute because the documentation generator does not convert the `name` to the proper link URL. For example:

```
<a id="anchors"></a>

## Anchors
```

Alerts and callouts

Alerts and callouts present warnings, extra detail, or references to related topics.

An alert or callout should not contain anything essential to understanding the main content. Instructions or tutorial steps should be in the main body of a guide rather than in a subsection.

Alerts

Alerts draw attention to short, important points. For multi-line content, see callouts.

See the live examples section for more information.

Note that at least one blank line must follow both the opening and closing `<div>` tags. A blank line before the closing `</div>` is conventional but not required.

```
<div class="alert is-helpful">

  See the [live examples](guide/docs-style-guide#live-examples "Live examples")
section for more information.

</div>
```

There are three different levels for styling the alerts according to the importance of the content.

Use the following the `alert` class along with the appropriate `is-helpful`, `is-important`, or `is-critical` CSS class, as follows:

```
<div class="alert is-helpful">

A helpful, informational alert.

</div>
```

```
<div class="alert is-important">

An important alert.

</div>
```

```
<div class="alert is-critical">

A critical alert.

</div>
```

The browser renders the following:

A helpful, informational alert.

An important alert.

A critical alert.

Callouts

Callouts, like alerts, highlight important points. Use a callout when you need a header and multi-line content.

If you have more than two paragraphs, consider creating a new page or making it part of the main content.

Callouts use the same styling levels as alerts.

Use the CSS class `callout` in conjunction with the appropriate `is-helpful`, `is-important`, or `is-critical` class. The following example uses the `is-helpful` class:

```
<div class="callout is-helpful">

<header>A helpful or informational point</header>

  **A helpful note**. Use a helpful callout for information requiring explanation.
  Callouts are typically multi-line notes.
  They can also contain code snippets.

</div>
```

The browser renders the three styles as follows:

A helpful or informational point

A helpful note. Use a helpful callout for information requiring explanation. Callouts are typically multi-line notes. They can also contain code snippets.

An important point

An important note. Use an important callout for significant information requiring explanation. Callouts are typically multi-line notes. They can also contain code snippets.

A critical point

A critical note. Use a critical callout for compelling information requiring explanation. Callouts are typically multi-line notes. They can also contain code snippets.

When using callouts, consider the following points:

- The callout header text style is uppercase.
- The header does not render in the table of contents.
- You can write the callout body in Markdown.
- A blank line separates the `<header>` tag from the Markdown content.
- Avoid using an `<h2>`, `<h3>`, `<h4>`, `<h5>`, or `<h6>`, as the CSS for callouts styles the `<header>` element.

Use callouts sparingly to grab the user's attention.

Trees

Use trees to represent hierarchical data such as directory structure.

sample-dir

```
<div class='file'>
  src
</div>
```

```
<div class='children'>

  <div class='file'>
    app
  </div>

  <div class='children'>

    <div class='file'>
      app.component.ts
    </div>

    <div class='file'>
      app.module.ts
    </div>

  </div>

  <div class='file'>
    styles.css
  </div>

  <div class='file'>
    tsconfig.json
  </div>

</div>

<div class='file'>
  node_modules ...
</div>

<div class='file'>
  package.json
</div>
```

Here is the markup for this file tree.

```
 <div class='filetree'>
     <div class='file'>
         sample-dir
     </div>
     <div class='children'>
         <div class='file'>
           src
         </div>
         <div class='children'>
             <div class='file'>
               app
             </div>
             <div class='children'>
```

```
                <div class='file'>
                  app.component.ts
                </div>
                <div class='file'>
                  app.module.ts
                </div>
              </div>
              <div class='file'>
                styles.css
              </div>
              <div class='file'>
                tsconfig.json
              </div>
            </div>
            <div class='file'>
              node_modules ...
            </div>
            <div class='file'>
              package.json
            </div>
        </div>
    </div>
```

Images

Store images in the `content/images/guide` directory in a folder with the **same name** as the guide page. Because Angular documentation generation copies these images to `generated/images/guide/your-guide-directory`, set the image `src` attribute to the runtime location of `generated/images/guide/your-guide-directory`.

For example, images for this "Angular documentation style guide" are in the `content/images/guide/docs-style-guide` folder, but the `src` attribute specifies the `generated` location.

The following is the `src` attribute for the "flying hero" image belonging to this guide:

```
src="generated/images/guide/docs-style-guide/flying-hero.png"
```

Use the HTML `<img>` tag

Specify images using the `<img>` tag. **Do not use the Markdown image syntax, ![...](...).**

For accessibility, always set the `alt` attribute with a meaningful description of the image.

Nest the `<img>` tag within a `<div class="lightbox">` tag, which styles the image according to the documentation standard.

```
<div class="lightbox">
  <img src="generated/images/guide/docs-style-guide/flying-hero.png" alt="flying
hero">
</div>
```

Note that the HTML `<img>` element does not have a closing tag.

The browser renders the following:

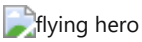

flying hero

Image captions and figure captions

A caption appears underneath the image as a concise and comprehensive summary of the image. Captions are optional unless you are using numbered figures, such as Figure 1, Figure 2, and so on. If you are using numbered figures in a page, follow the guidelines in Figure captions in the Google Developer Documentation Style Guide.

Image dimensions

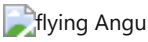
The doc generator reads the image dimensions from an image file and adds `width` and `height` attributes to the `<img>` tag automatically.

To control the size of the image, supply your own `width` and `height` attributes.

Here's the "flying hero" markup with a 200px width:

```
<div class="lightbox">
  <img src="generated/images/guide/docs-style-guide/flying-hero.png"
    alt="flying Angular hero"
    width="200">
</div>
```

The browser renders the following:


flying Angular hero

Wide images

To prevent images overflowing their viewports, **use image widths under 700px**. To display a larger image, provide a link to the actual image that the user can click to see the full size image separately, as in this example of `source-map-explorer` output from the "Ahead-of-time Compilation" guide:


toh-pt6-bundle

The following is the HTML for creating a link to the image:

```
<a href="generated/images/guide/docs-style-guide/toh-pt6-bundle.png" title="Click to
view larger image">
  <div class="lightbox">
    <img src="generated/images/guide/docs-style-guide/toh-pt6-bundle-700w.png"
alt="toh-pt6-bundle" width="300px">
  </div>
</a>
```

Image compression

For faster load times, always compress images. Consider using an image compression web site such as tinypng.

Floated images

You can float the image to the left or right of text by applying the class="left" or class="right" attributes respectively. For example:

```
<img src="generated/images/guide/docs-style-guide/flying-hero.png"
   alt="flying Angular hero"
   width="200"
   class="left">

This text wraps around to the right of the floating "flying hero" image.

Headings and code-examples automatically clear a floated image. If you need to force
a piece of text to clear a floating image, add `<br class="clear">` where the text
should break.

<br class="clear">
```

The browser renders the following:

 flying Angular hero

This text wraps around to the right of the floating "flying hero" image.

Headings and `<code-example>` components automatically clear a floated image. To explicitly clear a floated image, add `<br class="clear">` where the text should break.

Generally, you don't wrap a floated image in a `<figure>` element.

Floats within a subsection

If you have a floated image inside an alert, callout, or a subsection, apply the `clear-fix` class to the `<div>` to ensure that the image doesn't overflow its container. For example:

```
<div class="alert is-helpful clear-fix">

  <img src="generated/images/guide/docs-style-guide/flying-hero.png"
    alt="flying Angular hero"
    width="100"
    class="right">

  A subsection with **Markdown** formatted text.

</div>
```

The browser renders the following:

 flying Angular hero

A subsection with **Markdown** formatted text.

Help with documentation style

For specific language and grammar usage, a word list, style, tone, and formatting recommendations, see the Google Developer Documentation Style Guide.

If you have any questions that this style guide doesn't answer or you would like to discuss documentation styles visit the Angular repo and file a documentation issue.