


# Angular documentation style guide



This style guide is for anyone who contributes to the Angular documentation (this site). These guidelines should be followed by all authors. Deviations must be approved by a documentation editor.

The guidelines described here serve two purposes:

- To ensure a high-quality, consistent experience for Angular documentation users.
- To simplify the writing process for contributing authors. This guide helps you make decisions about tone, voice, and style. It also helps you find the right markup quickly.



This guide is a *living document*; it changes over time. We strive for consistency to the extent feasible, but you may find parts of our documentation that don't match this style guide. When in doubt, **follow this guide rather than imitating existing documents.**

## Scope of these guidelines

We ask all contributing authors to adhere to three aspects of style:

- **Writing style:** Word usage, grammar, capitalization, and punctuation. Adherence to Angular's writing guidelines ensures a consistent "voice", helps ensure accuracy of the information, and facilitates use world-wide, by audiences with different backgrounds.
- **Markup style:** How to include images, tables, alert boxes, and code snippets. Angular docs are written in Markdown, with custom extensions for this site. Correct markup ensures a consistent look-and-feel, and is essential for the doc to build and function correctly.

- **Angular coding style:** Coding style for example apps and code snippets. Code examples are encouraged for demonstrating how to apply the concepts and features discussed. Angular has a custom framework that enables authors to include code snippets directly from example apps that are automatically tested as part of doc builds. To contribute example code, you must understand Angular itself and the custom framework for Angular doc examples.

For each aspect of style, the following table explains where to find the primary guidelines and what this Angular Documentation Style Guide offers.

STYLE	GUIDELINES
Writing style	Primary: <a href="#">Google Developer Documentation Style Guide</a> This guide: Specifies any special considerations for Angular docs.
Markup style	Primary: This guide This guide: Specifies guidelines for markup of guides and tutorials, which are written primarily in Markdown.
Angular coding style	Primary: <a href="#">Angular Style Guide</a> . This guide: How to create, store, and include code examples in guides and tutorials.

Note: Angular API and CLI reference docs are generated from source code and/or related source files, which may have other markup styles and other ways of including code examples.

## Doc generation and tooling

To make changes to the documentation pages and sample code, clone the [Angular github repository](#) and go to the `aio/` folder.

The [aio/README.md](#) explains how to install and use the tools to edit and test your changes.

Here are a few essential commands for guide page authors.

1. `yarn setup` – installs packages; builds docs, stackblitz, and zips.
2. `yarn docs-watch --watch-only` – watches for saved content changes and refreshes the browser. The (optional) `--watch-only` flag skips the initial docs rebuild.
3. `yarn start` – starts the doc viewer application so you can see your local changes in the browser.
4. `http://localhost:4200/` – browse to the app running locally.

You can combine `yarn docs-watch` and `yarn start` into one command with `yarn serve-and-sync`.

## Guide pages

All but a few guide pages are `markdown` files with an `.md` extension.

Every guide page file is stored in the `content/guide` directory. Although the `side navigation` panel displays as a hierarchy, the directory is flat with no sub-folders. The flat folder approach allows us to shuffle the apparent navigation structure without moving page files or redirecting old page URLs.

The doc generation process consumes the markdown files in the `content/guide` directory and produces JSON files in the `src/generated/docs/guide` directory, which is also flat. Those JSON files contain a combination of document metadata and HTML content.

The reader requests a page by its Page URL. The doc viewer fetches the corresponding JSON file, interprets it, and renders it as fully-formed HTML page.

Page URLs mirror the `content` file structure. The URL for the page of a guide is in the form `guide/{page-name}`. The page for *this* "Authors Style Guide" is located at `content/guide/docs-style-guide.md` and its URL is `guide/docs-style-guide`.

*Tutorial* pages are exactly like guide pages. The only difference is that they reside in `content/tutorial` instead of `content/guide` and have URLs like `tutorial/{page-name}`.

*API* pages are generated from Angular source code into the `src/generated/docs/api` directory. The doc viewer translates URLs that begin `api/` into requests for document JSON files in that directory. This style guide does not discuss creation or maintenance of API pages.

*Marketing* pages are similar to guide pages. They're located in the `content/marketing` directory. While they can be markdown files, they may be static HTML pages or dynamic HTML pages that render with JSON data.

Only a few people are authorized to write marketing pages. This style guide does not discuss creation or maintenance of marketing pages.

## Markdown and HTML

While documentation guide pages ultimately render as HTML, almost all of them are written in [markdown](#).

Markdown is easier to read and to edit than HTML. Many editors (including Visual Studio Code) can render markdown as you type it.

From time to time you'll have to step away from markdown and write a portion of the document in HTML. Markdown allows you to mix HTML and markdown in the same document.

Standard markdown processors don't allow you to put markdown *within* HTML tags. But the Angular documentation markdown processor **supports markdown within HTML**, as long as you follow one rule:

**Always** follow every opening and closing HTML tag with *a blank line*.

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

**\*\*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.

The title should appear at the top of the physical page. Begin the title with the markdown `#` character. Alternatively, you can write the equivalent `<h1>`.

```
# Angular documentation style guide
```

**Only one title (`<h1>`) per document!**

Title text should be in "Sentence case", which means the first word is capitalized and all other words are lower case (unless they are technical terms that are always capitalized, like "Angular").

```
# Deprecation policy in Angular
```

**Always follow the title with at least one blank line.**

Note that the corresponding left-nav TOC text should be 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". The TOC title can also be shortened to fit in the column.

## Sections

A typical document is divided into sections.

All heading text should be in "Sentence case", which means the first word is capitalized and all other words are lower case.

Always follow the 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.

Alternatively, you can write the equivalent `<h2>` HTML tag.

The main section heading should be followed by a blank line and then the content for that heading.

```
## Sections
```

```
A typical document is divided into sections.
```

## 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.

Alternatively, you can write the equivalent `<h3>` HTML tag.

The secondary heading should be followed by a blank line and then the content for that heading.

```
### Secondary section heading
```

```
A secondary section ...
```

## Additional section headings

Try to minimize the heading depth, preferably only two. But more headings, such as this one, are permitted if they make sense.

N.B.: The [Table-of-contents](#) generator only considers main (`<h2>`) and secondary (`<h3>`) headings.

```
#### Additional section headings
```

```
Try to minimize ...
```

## Table of contents

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

You should not create your own TOC by hand. The TOC is generated automatically from the page's main and secondary section headers.

To exclude a heading from the TOC, create the heading as an `<h2>` or `<h3>` element with a class called 'no-toc'. You can't do this with markdown.

```
<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

The navigation links at the top, left, and bottom of the screen are generated from the JSON configuration file, `content/navigation.json`.

The authority to change the `navigation.json` file is limited to a few core team members. But for a new guide page, you should suggest a navigation title and position in the left-side navigation panel called the "side nav".

Look for 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. It probably looks something like this one.

```
{
  "url": "guide/architecture",
  "title": "Architecture",
  "tooltip": "The basic building blocks of Angular applications."
}
```

A navigation node has the following properties:

- `url` - the URL of the guide page (*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 (*header node only*).
- `hidden` - defined and set true if this is a guide page that should *not* be displayed in the navigation panel. Rarely needed, it is a way to hide the page from navigation while making it available to readers who should know about it. *This* "Authors Style Guide" is a hidden page.

Do not create a node that is both a *header* and an *item* node. That is, do not specify the `url` property of a *header* node.

The current guidelines allow for a three-level navigation structure with two header levels. Don't add a third header level.



## Code snippets

Guides are rich in examples of working Angular code. Example code can be commands entered in a terminal window, a fragment of TypeScript or HTML, or an entire code file.

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

### Code example

You can display a simple, 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.
```

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`.
```

For block code snippets, we generally prefer to display code with the Angular documentation *code-example* component represented by the `<code-example>` tag. The `<code-example>` tag has a `header` attribute that you use to identify the file that the example comes from. The header should be used whenever possible to establish the context of the example. See [Code snippets and code examples](#) for more details.

### Inline code-snippets

You should source code snippets [from working sample code](#) when possible. But there are times when an inline snippet is the better choice.

For terminal input and output, put the content between `<code-example>` tags, set the CSS class to `code-shell`, and set the language attribute to `sh` as in this example.

```
npm start
```

```
<code-example language="sh" class="code-shell">
  npm start
</code-example>
```

Inline, hand-coded snippets like this one are *not* testable and, therefore, are 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.

**Do not write inline code snippets** unless you have a good reason and the editor's permission to do so. 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, you should still use The `<code-example>` tag with the `header` attribute to identify the context.

## Code snippets and code samples

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

We meet this goal by displaying code snippets that are derived directly from standalone code samples, written specifically for these guide pages.

The author of a guide page is responsible for the code sample that supports that page. The author must also write end-to-end tests for the sample.

Code samples are located in sub-folders of the `content/examples` directory of the `angular/angular` repository. An example folder name should be 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 itself.

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.

## Code snippet from a file

This "Authors Doc Style Guide" has its own sample application, located in the `content/examples/docs-style-guide` folder.

The following *code-example* displays the sample's `app.module.ts`.

src/app/app.module.ts

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

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

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Here's the brief markup that produced that lengthy snippet:

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

You identified the snippet's source file by setting the `path` attribute to sample folder's location *within* `content/examples`. In this example, that path is `docs-style-guide/src/app/app.module.ts`.

You added a header to tell the reader where to find the file by setting the `header` attribute. Following convention, you set the `header` attribute to the file's location within the sample's root folder.

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

The doc tooling reports an error if the file identified in the path does not exist or is *git-ignored*.

Most `.js` files are *git-ignored*. If you want to include an ignored code file in your project and display it in a guide you must *un-ignore* it.

The preferred way to un-ignore a file is to update the `content/examples/.gitignore` like this:

```
content/examples/.gitignore
```

```
# my-guide  
!my-guide/src/something.js  
!my-guide/more-javascript*.js
```

## Code-example attributes

You control the *code-example* output by setting one or more of its attributes:

- `path`- the path to the file in the `content/examples` folder.
- `header`- the header of the code listing.
- `region`- displays the source file fragment with that region name; regions are identified by *docregion* markup in the source file, as explained [below](#).
- `linenums`- value may be `true`, `false`, or a `number`. When not specified, line numbers default to `false` (i.e. no line numbers are displayed). The rarely used `number` option starts line numbering at the given value. `linenums=4` sets the starting line number to 4.
- `class`- code snippets can be styled with the CSS classes `no-box`, `code-shell`, and `avoid`.
- `hideCopy`- hides the copy button
- `language`- the source code language such as `javascript`, `html`, `css`, `typescript`, `json`, or `sh`. This attribute only works for inline examples.

## Displaying a code fragment

Often you want to focus on a fragment of code within a sample code file. In this example, you focus on the `AppModule` class and its `NgModule` metadata.

```
@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

First you surround that fragment in the source file with a named *docregion* as described [below](#). Then you reference that *docregion* in the `region` attribute of the `<code-example>` like this

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

A couple of observations:

1. The `region` value, `"class"`, is the name of the `#docregion` in the source file. Confirm that by looking at [content/examples/docs-style-guide/src/app/app.module.ts](#)
2. Omitting the `header` is fine when the source of the fragment is obvious. We just said that this is a fragment of the `app.module.ts` file which was displayed immediately above, in full, with a header. There's no need to repeat the header.

## Example of bad code

Sometimes you want to display an example of bad code or bad design.

You should be careful. Readers don't always read carefully and are likely to copy and paste your example of bad code in their own applications. So don't display bad code often.

When you do, set the `class` to `avoid`. The code snippet will be framed in bright red to grab the reader's attention.

Here's the markup for an "avoid" example in the [Angular Style Guide](#).

```
<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>
```

app/heroes/hero-button/hero-button.component.ts

```
/* avoid */

@Component({
  selector: '[tohHeroButton]',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

## Code Tabs

Code tabs display code much like *code examples* do. The added advantage is that they can display multiple code samples within a tabbed interface. Each tab is displayed using *code pane*.

### Code-tabs attributes

- `linenums`: The value can be `true`, `false` or a number indicating the starting line number. If not specified, it defaults to `false`.

### Code-pane attributes

- `path` - a file in the content/examples folder
- `header` - seen 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. If not specified, it defaults to `false`.

The next example displays multiple code tabs, each with its own header. It demonstrates control over display of line numbers at both the `<code-tabs>` and `<code-pane>` levels.

app.component.html      app.component.ts      app.component.css

```
1. <code><h2>My Heroes</h2></code>
2. <code><h2>My Heroes</h2></code>
3. <code><ul class="heroes"></code>
4. <code>  <li *ngFor="let hero of heroes"</code>
5. <code>    [class.selected]="hero === selectedHero"</code>
6. <code>    (click)="onSelect(hero)"</code>
7. <code>    <span class="badge">{{hero.id}}</span></code>
  <code>    {{hero.name}}</code>
8. <code>  </li></code>
9. <code></ul></code>
10. <code><div *ngIf="selectedHero"></code>
11. <code>  <h2>{{selectedHero.name}} details!</h2></code>
12. <code>  <div><label>id: </label>{{selectedHero.id}}</code>
  <code>  </div></code>
13. <code>  <div></code>
14. <code>    <label>name: </label></code>
15. <code>    <input [(ngModel)]="selectedHero.name"</code>
  <code>      placeholder="name"/></code>
16. <code>  </div></code>
17. <code></div></code>
```

Here's the markup for that example.

Note how the `linenums` attribute in the `<code-tabs>` explicitly enables numbering for all panes. The `linenums` attribute in the second pane disables line numbering for *itself only*.

```
<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>
```

## Source code markup

You must add special code snippet markup to sample source code files before they can be displayed by `<code-example>` and `<code-tabs>` components.

The sample source code for this page, located in [context/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. Here's the default *docregion* markup for a TypeScript or JavaScript file:

```
// #docregion
... some code ...
// #enddocregion
```

Different file types have different comment syntax so adjust accordingly.



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

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

The doc generation process erases these comments before displaying them in the doc viewer. It also strips them from stackblitz and sample code downloads.

Code snippet markup is not supported in JSON files because comments are forbidden in JSON files. See [below](#) for details and workarounds.

## *#docregion*

The *#docregion* is the most important kind of code snippet markup.

The `<code-example>` and `<code-tabs>` components won't display a source code file unless it has a *#docregion*.

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 `src/main.ts` is a simple example of a file with a single *#docregion* at the top of the file.

src/main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from
'./environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
```

## Named *#docregions*

You'll often display multiple snippets from different fragments within the same file. You distinguish among them by giving each fragment its own *#docregion name* as follows.

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

Remember to refer to this region by name in the `region` attribute of the `<code-example>` or `<code-pane>` as you did in an example above like this:

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

The *#docregion* with no name is the *default region*. Do *not* set the `region` attribute when referring to the default *#docregion*.

## Nested *#docregions*

You can nest `#docregions` within `#docregions`

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

The `src/app/app.module.ts` file has a good example of a nested region.

## Combining fragments

You can combine several fragments from the same file into a single code snippet by defining multiple `#docregions` with the *same region name*.

Examine the `src/app/app.component.ts` file which defines two nested `#docregions`.

The inner, `class-skeleton` region appears twice, once to capture the code that opens the class definition and once 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
```

Here's are the two corresponding code snippets displayed side-by-side.

```
export class AppComponent {  
  title = 'Authors Style Guide Sample';  
  heroes = HEROES;  
  selectedHero: Hero;  
  
  onSelect(hero: Hero): void {  
    this.selectedHero = hero;  
  }  
}
```

Some observations:

- The `#docplaster` at the top is another bit of code snippet markup. It tells the processor how to join the fragments into a single snippet.  
In this example, we tell the processor to put the fragments together without anything in between - without any "plaster". Most sample files define this *empty plaster*.  
If we neglected to add, `#docplaster`, the processor would insert the *default* plaster - an ellipsis comment - between the fragments.  
Try removing the `#docplaster` comment yourself to see the effect.
- One `#docregion` comment mentions *two* region names as does an `#enddocregion` comment. This is a convenient way to start (or stop) multiple regions on the same code line. You could have put these comments on separate lines and many authors prefer to do so.

## JSON files

Code snippet markup is not supported for JSON files because comments are forbidden in JSON files.

You can display an entire JSON file by referring to it in the `src` attribute. But you can't display JSON fragments because you can't add `#docregion` tags to the file.

If the JSON file is too big, you could copy the nodes-of-interest into markdown backticks.

Unfortunately, it's easy to mistakenly create invalid JSON that way. The preferred way is to create a JSON partial file with the fragment you want to display.

You can't test this partial file and you'll never use it in the application. But at least your IDE can confirm that it is syntactically correct.

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

#### package.json (selected scripts)

```
{
  "scripts": {
    "start": "concurrently \"npm run build:watch\" \"npm run serve\"",
    "test": "concurrently \"npm run build:watch\" \"karma start karma.conf.js\"",
    "lint": "tslint ./src/**/*.ts -t verbose"
  }
}
```

**<code-example**

`path="docs-style-guide/package.1.json"`

`header="package.json (selected scripts)"></code-`

**example>**

## Partial file naming

Many guides tell a story. In that story, the app evolves incrementally, often with simplistic or incomplete code along the way.

To tell that story in code, you'll often need to create partial files or intermediate versions of the final source code file with fragments of code that don't appear in the final app.

Such partial and intermediate files need their own names. Follow the doc sample naming convention. Add a number before the file extension as illustrated here:

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

You'll find many such files among the samples in the Angular documentation.

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

stackblitz.json

```
{
  "description": "Authors style guide",
  "files": [
    "!**/*.d.ts",
    "!**/*.js",
    "!**/*.[1,2,3].*"
  ],
  "tags": ["author", "style guide"]
}
```

## Live examples

By adding `<live-example>` to the page you generate links that run sample code in the Stackblitz live coding environment and download that code to the reader's file system.

Live examples (AKA "stackblitz") are defined 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`.

You can create additional, named definition files in the form `name.stackblitz.json`. See `content/examples/testing` for examples.

The schema for a `stackblitz.json` hasn't been documented yet but looking at the `stackblitz.json` files in the example folders should tell you most of what you need to know.

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

[live example](#) / [download example](#)

1. a link to the Stackblitz defined by the default `stackblitz.json` file located in the code sample folder with the same name as the guide page.
2. a link that downloads that sample.

Clicking the first link opens the code sample on StackBlitz in a new browser tab.

You can change the appearance and behavior of the live example with attributes and classes.

## Custom label and tooltip

Give the live example anchor a custom label and tooltip by setting the `title` attribute.

[Live Example with title](#) / [download example](#)

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

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

[Live example with content label](#) / [download example](#)

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

## Live example from another guide

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

[Live Example from the Router guide](#) / [download example](#)

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

## Live Example for named Stackblitz

To link to a Stackblitz defined by a named `stackblitz.json` file, set the `stackblitz` attribute. The following example links to the Stackblitz defined by `second.stackblitz.json` in the current guide's directory.

[live example](#) / [download example](#)

```
<live-example stackblitz="second"></live-example>
```

## Live Example without download

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

Just the Stackblitz

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

## Live Example with download-only

To skip the live Stackblitz link and only link to the download, add the `downloadOnly` attribute.

Download only

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

## Embedded live example

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

For performance reasons, the Stackblitz does not start right away. The reader sees an image instead. Clicking the image starts the sometimes-slow process of launching the embedded Stackblitz within an iframe on the page.

Here's an embedded live example for this guide.

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



Run Project 

You can also [download this example](#).

## Anchors

Every section header tag is also an anchor point. Another guide page could add a link to this section by writing:

See the ["Anchors"](#) section for details.

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

```
See the ["Anchors"](guide/docs-style-guide#anchors  
"Style Guide - Anchors") section for details.
```

```
</div>
```

When navigating within the page, you can omit the page URL when specifying the link that [scrolls up](#) to the beginning of this section.

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

### Ugly, long section header anchors

It is often a good idea to *lock-in* a good anchor name.

Sometimes the section header text makes for an unattractive anchor.  
[This one](#) is pretty bad.

```
[This one](#ugly-long-section-header-anchors) is pretty bad.
```

The greater danger is that a future rewording of the header text would break a link to this section.

For these reasons, it is often wise to add a custom anchor explicitly, just above the heading or text to which it applies, using the special syntax like this.

```
{@a ugly-anchors}  
  
#### Ugly, long section header anchors
```

Now [link to that custom anchor name](#) as you did before.

```
Now [link to that custom anchor name](#ugly-anchors) as  
you did before.
```

Alternatively, you can use the HTML `<a>` tag.

If you do, be sure to set the `id` attribute - not the `name` attribute! The docs generator will not convert the `name` to the proper link URL.

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


## Alerts and Callouts

Alerts and callouts present warnings, extra detail or references to other pages. They can also be used to provide commentary that *enriches* the reader's understanding of the content being presented.

An alert or callout *must not* contain anything *essential* to that understanding. Don't put a critical instruction or a tutorial step in a subsection.

## Alerts

Alerts draw attention to short important points. Alerts should not be used for multi-line content (use [callouts](#) instead).



You'll learn about styles for live examples in the [section below](#).

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

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

```
You'll learn about styles for live examples in the  
[section below](guide/docs-style-guide#live-examples  
"Live examples").
```

```
</div>
```

There are three different *urgency levels* used to style the alerts based on the severity or importance of the content.



A critical alert.



An important alert.



A helpful, informational alert.

Here is the markup for these alerts.

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

A critical alert.

```
</div>
```

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

An important alert.

```
</div>
```

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

A helpful, informational alert.

```
</div>
```

## Callouts

Callouts, like alerts, are meant to draw attention to important points. Use a callout when you want a riveting 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 *urgency levels* that alerts do.

### A CRITICAL POINT

Pitchfork hoodie semiotics, roof party pop-up *paleo* messenger messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix

### AN IMPORTANT POINT

Pitchfork hoodie semiotics, roof party pop-up *paleo* messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix

## A HELPFUL OR INFORMATIONAL POINT

Pitchfork hoodie semiotics, roof party pop-up *paleo* messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix

Here is the markup for the first of these callouts.

```
<div class="callout is-critical">
<header>A critical point</header>

**Pitchfork hoodie semiotics**, roof party pop-up
_paleo_ messenger bag cred Carles tousled Truffaut yr.
Semiotics viral freegan VHS, Shoreditch disrupt
McSweeney's. Intelligentsia kale chips Vice four dollar
toast, Schlitz crucifix

</div>
```

Notice that:

- the callout header text is forced to all upper case
- the callout body can be written in markdown
- a blank line separates the `</header>` tag from the markdown content

Callouts are meant to grab the user's attention. They are not for casual asides. Please use them sparingly.

## Trees

Trees can represent hierarchical data.

```
sample-dir
├── src
│   ├── app
│   │   ├── app.component.ts
│   │   └── app.module.ts
│   ├── styles.css
│   └── tsconfig.json
└── node_modules ...
```

```
package.json
```

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>
```

## Tables

Use HTML tables to present tabular data.

Framework	Task	Speed
AngularJS	Routing	Fast
Angular v2	Routing	<i>Faster</i>
Angular v4	Routing	<b>Fastest :)</b>

Here is the markup for this table.

```

<style>
  td, th {vertical-align: top}
</style>

<table>
  <tr>
    <th>Framework</th>
    <th>Task</th>
    <th>Speed</th>
  </tr>
  <tr>
    <td><code>AngularJS</code></td>
    <td>Routing</td>
    <td>Fast</td>
  </tr>
  <tr>
    <td><code>Angular v2</code></td>
    <td>Routing</td>
    <!-- can use markdown too; remember blank lines -->
    <td>

      *Faster*

    </td>
  </tr>
  <tr>
    <td><code>Angular v4</code></td>
    <td>Routing</td>
    <td>

      **Fastest :)**

    </td>
  </tr>
</table>

```

## Images

### Image location

Store images in the `content/images` directory in a folder with the same URL as the guide page. Images for this "Authors Style Guide" page belong in the `content/images/guide/docs-style-guide` folder.



Angular doc generation copies these image folders to the *runtime* location, `generated/images`. Set the image `src` attribute to begin in *that* directory.

Here's the `src` attribute for the "flying hero" image belonging to this page.

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

## Use the HTML `<img>` tag

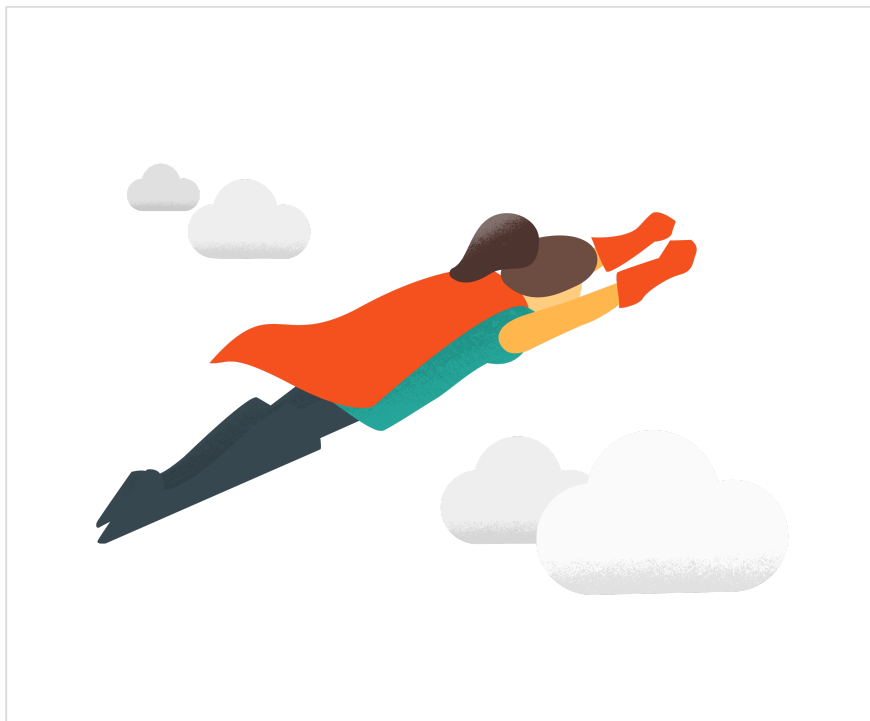
Do not use the markdown image syntax, `![...](...)`.

Images should be specified in an `<img>` tag.

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

You should nest the `<img>` tag within a `<div class="lightbox">` tag, which styles the image within a drop-shadow frame. You'll need the editor's permission to skip the `lightbox` class on its `div` encapsulation.

Here's a conforming example



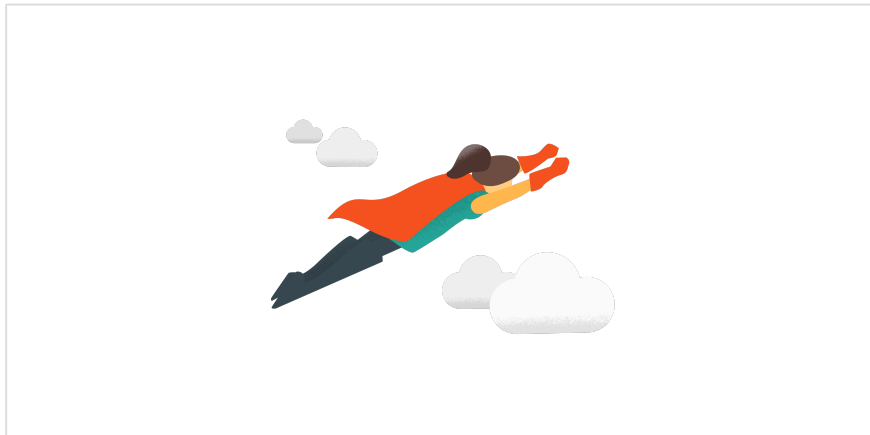
```
<div class="lightbox">
  
</div>
```

*Note that the HTML image element does not have a closing tag.*

## Image dimensions

The doc generator reads the image dimensions from the file and adds width and height attributes to the `img` tag automatically. If you want to control the size of the image, supply your own width and height attributes.

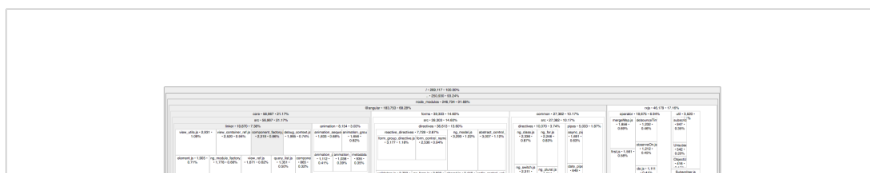
Here's the "flying hero" at a more reasonable scale.



```
<div class="lightbox">
  
</div>
```

Wide images can be a problem. Most browsers try to rescale the image but wide images may overflow the document in certain viewports.

**Do not set a width greater than 700px.** If you wish to display a larger image, provide a link to the actual image that the user can click on to see the full size image separately as in this example of [source-map-explorer](#) output from the "Ahead-of-time Compilation" guide:



[illegible]

## Image compression

Large image files can be slow to load, harming the user experience. Always compress the image. Consider using an image compression web site such as [tinypng](#).

# Floating images

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



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

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

The markup for the above example is:

```

```

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

Headings and code-examples automatically clear a floating 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">
```

Note that you generally don't wrap a floating image in a `<figure>` element.

## Floating within a subsection

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



A subsection with **markdown** formatted text.

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

    
```

A subsection with **markdown** formatted text.

```
</div>
```

