Go CSS Coding Guidelines

Please discuss changes before editing this page, even *minor* ones. Many people have opinions and this is not the place for edit wars.

Go's naming conventions are adapted from Medium's style guide and the work being done in the SUIT CSS framework. Which is to say, it relies on *structured class names* and *meaningful hyphens* (i.e., not using hyphens merely to separate words). This is to help work around the current limits of applying CSS to the DOM (i.e., the lack of style encapsulation) and to better communicate the relationships between classes.

We require plain CSS over preprocessors like LESS, SASS, etc. This is to avoid additional tooling dependencies.

JavaScript

syntax: `js-<targetName>`

JavaScript-specific classes reduce the risk that changing the structure or theme of components will inadvertently affect any required JavaScript behavior and complex functionality. You must use these classes if you interact with the DOM via JavaScript. In practice this looks like this:

```
<a href="/login" class="btn btn-primary js-login"></a>
```

Again, JavaScript-specific classes should not, under any circumstances, be styled.

Components

Syntax: `[namespace-]<ComponentName>[-descendantName][--modifierName]`

Component driven development offers several benefits when reading and writing HTML and CSS:

- It helps to distinguish between the classes for the root of the component, descendant elements, and modifications.
- It keeps the specificity of selectors low.
- It helps to decouple presentation semantics from document semantics.

You can think of components as custom elements that enclose specific semantics, styling, and behavior.

namespace (optional)

If necessary, components can be prefixed with a namespace. For example, you may wish to avoid the potential for collisions between libraries and your custom components by prefixing all your components with a namespace.

```
.godoc-Button { /* ... */ }
.godoc-Tabs { /* ... */ }
```

This makes it clear, when reading the HTML, which components are part of your library.

ComponentName

The component's name must be written in camel case.

```
.MyComponent { /* ... */ }
<article class="MyComponent">
  ...
</article>
```

ComponentName–modifierName

A component modifier is a class that modifies the presentation of the base component in some form. Modifier names must be written in camel case and be separated from the component name by two hyphens. The class should be included in the HTML *in addition* to the base component class.

```
/* Core button */
.Button { /* ... */ }
/* Default button style */
.Button--default { /* ... */ }
<button class="Button Button--primary">...</button>
```

ComponentName-descendantName

A component descendant is a class that is attached to a descendant node of a component. It's responsible for applying presentation directly to the descendant on behalf of a particular component. Descendant names must be written in camel case.

```
<article class="Tweet">
  <header class="Tweet-header">
    <img class="Tweet-avatar" src="{$src}" alt="{$alt}">
    ...
  </header>
  <div class="Tweet-body">
    ...
  </div>
</article>
```

ComponentName[aria-]

Almost always, component states can be represented using ARIA (aria-expanded, aria-disabled, etc.) so they are recommended for state-based modifications of components. In the very rare case that there is a state that cannot be represented using ARIA, a CSS class of the format `is-stateName` can be used. The CSS

state name must be Camel case. Never style these attributes/classes directly; they should always be used as an adjoining class or attribute.

JS can add/remove these attributes/classes. Every component must define its own styles for the state (as they are scoped to the component).

```
.Tweet { /* ... */ }
.Tweet[aria-expanded=true] { /* ... */ }
.Tweet.is-blorked { /* ... */ }
<article class="Tweet is-blorked" aria-expanded="true">
  ...
</article>
```

Colors

Prefer lower-case hex values, RGB, or RGBA over named, HSL, or HSLA values. Use short-form hex values where applicable.

Right:

```
#fff;
#f1f2f3;
rgb(50, 50, 50);
rgba(50, 50, 50, 0.2);
```

Wrong:

```
#FFFFFF;
#F1F2F3;
white;
hsl(120, 100%, 50%);
hsla(120, 100%, 50%, 1);
```

Formatting

The following are some high level page formatting style rules.

Spacing

CSS rules should be comma separated but live on new lines:

Right:

```
.Content,
.Content-edit {
  ...
}
```

Wrong:

```
.Content, .Content-edit {
  ...
}
```

CSS blocks should be separated by a single new line. not two. not 0.

Right:

```
.Content {
  ...
}
.Content-edit {
  ...
}
```

Wrong:

```
.Content {
  ...
}

.Content-edit {
  ...
}
```

Quotes

Quotes are optional in CSS. We use single quotes as it is visually clearer that the string is not a selector or a style property.

Right:

```
background-image: url('/img/you.jpg');
font-family: 'Helvetica Neue Light', 'Helvetica Neue', Helvetica, Arial;
```

Wrong:

```
background-image: url(/img/you.jpg);
font-family: Helvetica Neue Light, Helvetica Neue, Helvetica, Arial;
```

Write Simple Selectors

Avoid sequences of simple selectors ("compound selectors").

The only exceptions are state-based selectors that are dynamically added to indicate a state change (for example, a tweet that has "[aria-expanded=true]").

Right:

```
/* State-based simple selector. */
tweet[aria-expanded=true] {
  ...
```

```
}
```

Wrong:

```
/* Sequence of simple selectors. */
button.foo.bar {
  ...
}
```

Images

Add height and width attributes to images in the html markup to help minimize layout shift during page load.

```
<img src="url/to/image.png" height="240" width="160">
```