Emmet Documentation

Back to main website

Cheat Sheet

More developer tools:

Emmet LiveStyle

Real-time bi-directional edit tool for CSS, LESS and SCSS.

Emmet Re:view

Fast and easy way to test responsive design side-by-side.

Abbreviations Syntax

Emmet uses syntax similar to CSS selectors for describing elements' positions inside generated tree and elements' attributes.

Elements

You can use elements' names like `div` or `p` to *generate* HTML tags. Emmet doesn't have a predefined set of available tag names, you can write any word and transform it into a tag: `div` → `<div></div>`, `foo` → `<foo></foo>` and so on.

Nesting operators

Nesting operators are used to position abbreviation elements inside generated tree: whether it should be placed inside or near the context element.

Child: >

You can use `>` operator to nest elements inside each other:

```
div>ul>li
```

...will produce

```
<div>
    <ul>
        <li></li>
    </ul>
</div>
```

Sibling: +

Use + operator to place elements near each other, on the same level:

```
div+p+bq
```

...will output

```
<div></div>
<p></p>
<blockquote></blockquote>
```

Climb-up: ^

With > operator you're descending down the generated tree and positions of all sibling elements will be resolved against the most deepest element:

```
div+div>p>span+em
```

...will be expanded to

```
<div></div>
<div>
    <p><span></span><em></em></p>
</div>
```

With ^ operator, you can climb one level up the tree and change context where following elements should appear:

```
div+div>p>span+em^bq
```

...outputs to

```
<div></div>
<div>
    <p><span></span><em></em></p>
    <blockquote></blockquote>
</div>
```

You can use as many ^ operators as you like, each operator will move one level up:

```
div+div>p>span+em^^^bq
```

...will output to

```
<div></div>
<div>
    <p><span></span><em></em></p>
</div>
<blockquote></blockquote>
```

Multiplication: *

With * operator you can define how many times element should be outputted:

```
ul>li*5
```

...outputs to

```
<ul>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
</ul>
```

Grouping: ()

Parenthesises are used by Emmets' power users for grouping subtrees in complex abbreviations:

```
div>(header>ul>li*2>a)+footer>p
```

...expands to

```
<div>
    <header>
        <ul>
            <li><a href=""></a></li>
            <li><a href=""></a></li>
        </ul>
    </header>
    <footer>
        <p></p>
    </footer>
</div>
```

If you're working with browser's DOM, you may think of groups as Document Fragments: each group contains abbreviation subtree and all the following elements are inserted at the same level as the first element of group.

You can nest groups inside each other and combine them with multiplication `*` operator:

```
(div>dl>(dt+dd)*3)+footer>p
```

...produces

```
<div>
    <dl>
        <dt></dt>
        <dd></dd>
        <dt></dt>
        <dd></dd>
        <dt></dt>
        <dd></dd>
    </dl>
</div>
<footer>
    <p></p>
</footer>
```

With groups, you can literally write full page mark-up with a single abbreviation, but please don't do that.

Attribute operators

Attribute operators are used to modify attributes of outputted elements. For example, in HTML and XML you can quickly add `class` attribute to generated element.

ID and CLASS

In CSS, you use `elem#id` and `elem.class` notation to reach the elements with specified `id` or `class` attributes. In Emmet, you can use the very same syntax to *add* these attributes to specified element:

```
div#header+div.page+div#footer.class1.class2.class3
```

...will output

```
<div id="header"></div>
<div class="page"></div>
```

```
<div id="footer" class="class1 class2 class3"></div>
```

Custom attributes

You can use `[attr]` notation (as in CSS) to add custom attributes to your element:

```
td[title="Hello world!" colspan=3]
```

...outputs

```
<td title="Hello world!" colspan="3"></td>
```

- You can place as many attributes as you like inside square brackets.
- You don't have to specify attribute values: `td[colspan title]` will produce `<td colspan="" title="">` with tabstops inside each empty attribute (if your editor supports them).
- You can use single or double quotes for quoting attribute values.
- You don't need to quote values if they don't contain spaces: `td[title=hello colspan=3]` will work.

Item numbering: $

With multiplication `*` operator you can repeat elements, but with `$` you can *number* them. Place `$` operator inside element's name, attribute's name or attribute's value to output current number of repeated element:

```
ul>li.item$*5
```

...outputs to

```
<ul>
    <li class="item1"></li>
    <li class="item2"></li>
    <li class="item3"></li>
    <li class="item4"></li>
    <li class="item5"></li>
</ul>
```

You can use multiple `$` in a row to pad number with zeroes:

```
ul>li.item$$$*5
```

...outputs to

```
<ul>
    <li class="item001"></li>
    <li class="item002"></li>
    <li class="item003"></li>
    <li class="item004"></li>
    <li class="item005"></li>
</ul>
```

Changing numbering base and direction

With `@` modifier, you can change numbering direction (ascending or descending) and base (e.g. start value).

For example, to change direction, add `@-` after `$`:

```
ul>li.item$@-*5
```

…outputs to

```
<ul>
    <li class="item5"></li>
    <li class="item4"></li>
    <li class="item3"></li>
    <li class="item2"></li>
    <li class="item1"></li>
</ul>
```

To change counter base value, add `@N` modifier to `$`:

```
ul>li.item$@3*5
```

…transforms to

```
<ul>
    <li class="item3"></li>
    <li class="item4"></li>
    <li class="item5"></li>
    <li class="item6"></li>
    <li class="item7"></li>
</ul>
```

You can use these modifiers together:

```
ul>li.item$@-3*5
```

…is transformed to

```
<ul>
    <li class="item7"></li>
    <li class="item6"></li>
    <li class="item5"></li>
    <li class="item4"></li>
    <li class="item3"></li>
</ul>
```

Text: {}

You can use curly braces to add text to element:

```
a{Click me}
```

...will produce

```
<a href="">Click me</a>
```

Note that `{text}` is used and parsed as a separate element (like, `div`, `p` etc.) but has a special meaning when written right after element. For example, `a{click}` and `a>{click}` will produce the same output, but `a{click}+b{here}` and `a>{click}+b{here}` won't:

```
<!-- a{click}+b{here} -->
<a href="">click</a><b>here</b>

<!-- a>{click}+b{here} -->
<a href="">click<b>here</b></a>
```

In second example the <b> element is placed *inside* <a> element. And that's the difference: when `{text}` is written right after element, it doesn't change parent context. Here's more complex example showing why it is important:

```
p>{Click }+a{here}+{ to continue}
```

...produces

```
<p>Click <a href="">here</a> to continue</p>
```

In this example, to write `Click here to continue` inside <p> element we have explicitly move down the tree with > operator after `p`, but in case of a element we don't have to,

since we need <a> element with `here` word only, without changing parent context.

For comparison, here's the same abbreviation written without child > operator:

```
p{Click }+a{here}+{ to continue}
```

...produces

```
<p>Click </p>
<a href="">here</a> to continue
```

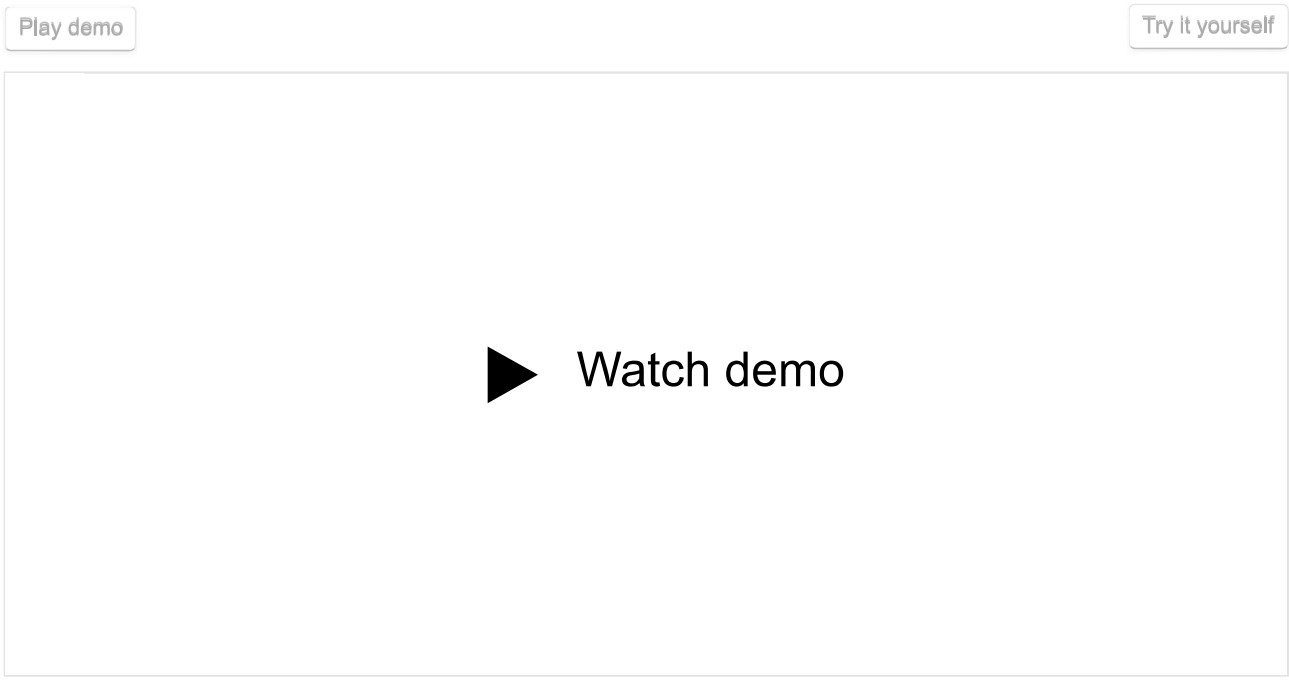
Notes on abbreviation formatting

When you get familiar with Emmet's abbreviations syntax, you may want to use some formatting to make your abbreviations more readable. For example, use spaces between elements and operators, like this:

```
(header > ul.nav > li*5) + footer
```

But it won't work, because space is a *stop symbol* where Emmet stops abbreviation parsing.

Many users mistakenly think that each abbreviation should be written in a new line, but they are wrong: you can type and expand abbreviation *anywhere in the text*:



This is why Emmet needs some indicators (like spaces) where it should stop parsing to not expand anything that you don't need. If you're still thinking that such formatting is required for complex abbreviations to make them more readable:

- Abbreviations are not a template language, they don’t have to be “readable”, they have to be “quickly expandable and removable”.
- You don’t really need to write complex abbreviations. Stop thinking that “typing” is the slowest process in web-development. You’ll quickly find out that constructing a single complex abbreviation is much slower and error-prone than constructing and typing a few short ones.

Support: info@emmet.io
Created with DocPad and Gulp.js
Minimal theme by orderedlist

View page source on GitHub