Notes - CSS - Flexbox

- Dr Nick Hayward

A general intro and outline for using flexbox with HTML5 compatible apps.

Contents

Intro CSS Flexbox helps solve many issues that have continued to plague layout and positioning of HTML elements and components in both client-side and cross-platform apps.

For example,

- vertical and horizontal alignment
- defining a centred position for child elements relative to their parent
- equal spacing and proportions for child nodes regardless of available space
- equal heights and widths for varied content
- & lots more…

Layout options and Flexbox In CSS, there are a number of comparative and complementary layout options to Flexbox. A knowledge of each is usefult a general understanding of page layout, design, and appropriate use of Flexbox.

As CSS has evolved, we have seen the introduction of various layout options such as `float` , grids, box alignment, and, of course, Flexbox itself.

As such, we may also see the influence of rendering options such as `block` and `inline` , which continue to inform and influence Flexbox.

For example, if we define an element with the property and value `display: flex` , it will commonly adopt the same alignment as a standard *block level* container. Standard element flow is not interrupted, for example by *float*, and the container's margins will, thankfully, not collapse.

Likewise, we may migrate content styled using other layout options such as *float* and *vertical-align* to flex items. If we initially style an element using `float` , and then set the container to flex, the default floating will no longer be rendered.

The same overriding effect may be seen with the vertical align property.

flex and grids CSS offers two similar layout options, Flexbox and Grid layout.

Whilst they may appear to fulfill the same basic layout requirements, there are subtle differences that help us control content layout and styling.

We may use Flexbox as a graceful fallback for some Grid layouts in older web browsers. Flexbox customarily includes better support than Grids in such older browsers, and may be upgraded to a Grid layout without issue where supported.

For example, if we define an element as a flex container with flex items, it may then be upgraded to Grid, and the flex properties for child elements will then be ignored. We may then style the child elements as part of the Grid layout without conflict from Flexbox.

flex vs grids However, how does Flexbox and Grid Layout differ for practical usage and styling. As such, the specifications define Flexbox as a *one-dimensional* layout option, compared with Grid Layout, which is described as *two-dimensional*.

If we consider a standard flex layout, which may allow items to wrap from line to line. Such items, however, will now define each new line as a flex container in its own right. This causes flex to ignore, as such, placement of items i other rows, trying instead to line items up with each other in the current container.

So, we may get two lines of three items, and then a final line with one item stretched the full length of the line, or container.

However, with Grid Layout we may control with precision layout in both rows and columns, thereby negating the effect described for flexbox. We may render two line of three items, and a final line with one item the same width as the other items in the previous lines.

These initial differences also help describe fundamental contrasts between Flexbox and Grid Layout usage.

For example, with Grid Layout we define the majority of layout and sizing on the container itself. We may setup *tracks*, and then the child items in such precision alignments.

By contrast, whilst we may set content direction for a flex container, sizing of flex items is defined for those items themselves.

Both options, Grid and Flexbox, are a valid and useful option for layout in CSS. Generally, we may consider the following points to help with choosing grid or flex styling.

- grid layout is preferable, in most situations, for row based layouts with matching alignment
  - e.g. trying to align items with items in rows above and below
  - grid easily supports such two dimensional layout
- consider the difference between one dimension and two dimensional requirements for layout
  - two dimensional may be equally applicable for small, single component

Basic usage For any app layout, we need to define specific elements as *flexible boxes*.

i.e. those allowed to use flexbox in a given app, e.g.

```
section {
  display: flex;
}
```

This CSS ruleset will define a `section` element as a parent *flex container*. Any child elements may now accept flex declarations.

This initial declaration, `display: flex` , also includes default values for flexbox layout of child elements.

e.g. `<div>` elements in a section will, by default, be arranged as equal sized columns with the same initial height.

Such default values for a container's flex items may include the following,

- flex items rendered in a row direction
- items will begin from the *start* edge of the main axis
  - start will be relative to current defined writing system
  - e.g. left to right for English, right to left for Arabic &c.
- items do not stretch on the main axis, but may shrink
- items will also stretch on the cross axis
- `flex-wrap` property set to `nowrap`

Such default values, and their variant usage, will be covered in later sections.

Axes Elements arranged using flexbox are laid out on two axes,

- main axis
  - axis running in the direction of the currently laid out flex items
  - e.g. rows or columns
  - start and end of axis = *main start* & *main end*
- cross axis
  - axis running perpendicular to the current main axis
  - start and end of axis = *cross start* & *cross end*

Each child element being laid out inside the flex container is called a *flex item*.

Flex direction We can set a property for the flex direction, which defines direction of the flex items relative to the main axis. i.e. the layout direction for the child elements.

Default setting is `row` , and the direction will be relative to the current browser language setting. e.g. for English language browsers this will be left to right.

```
section {
  flex-direction: column;
}
```

This will override the default `row` setting, and arrange the child items in a column.

So, we might define a default `section` element ruleset as follows,

```
section {
  display: flex;
  flex-direction: column;
}
```

This would ensure that child flex items were 1aid out in a single column. The flex direction will now run in a *block direction*, from the top of the page to the bottom.

However, we might override specific `section` elements to allow child flex items in a `row` direction.

e.g.

```
#tabs {
  flex-direction: row;
}
```

A `row` flex direction will render the content for this selector in an *inline direction*, from left to right and vice-versa relative to formatting requirements.

We may also define `row-reverse` and `column-reverse` for flex direction property.

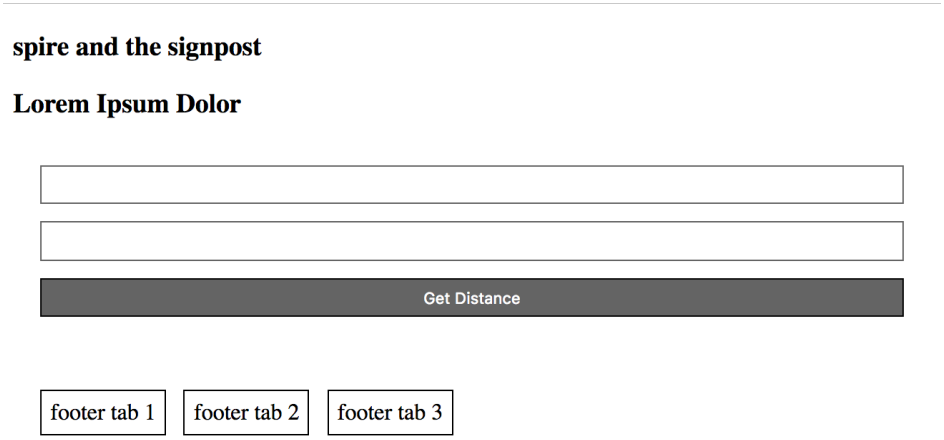

Figure 1: CSS Flexbox - flex direction

flex item wrapping To ensure that child items do not overlap their parent flex container, we may add a declaration for `flex-wrap` to a required ruleset. In effect, we may cause flex items to wrap from one line to another.

As each line wraps, we may consider each new line as an implicit container for that content. The content will expand and render for the current line, with no reference to lines before or after the current line.

For example, if the flex items are set to the default `nowrap` value for `flex-wrap` , they will instead shrink to fill the availabel space. If the items are no able to shrink, the rendered content will, instead, overflow the container.

e.g.

```
#tabs {
  flex-direction: row;
  flex-wrap: wrap;
}
```

So, without wrap

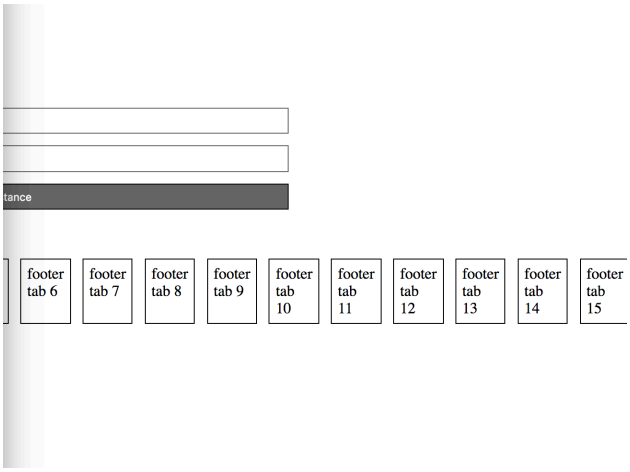

Figure 2: CSS Flexbox - no flex wrap

and, with wrap

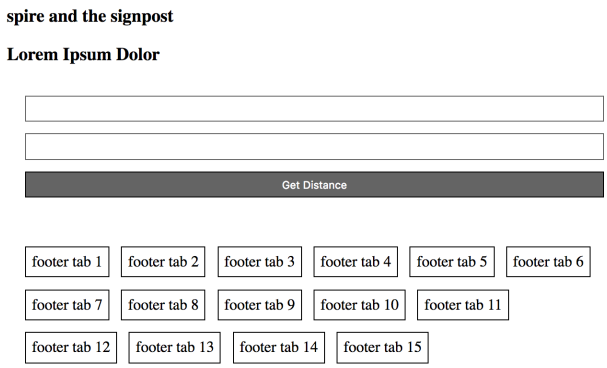

Figure 3: CSS Flexbox - flex wrap

We might also set the flex direction to reverse, which will start the flex items from the right on an English language browser.

```
#tabs {
  flex-direction: row-reverse;
  flex-wrap: wrap;
}
```

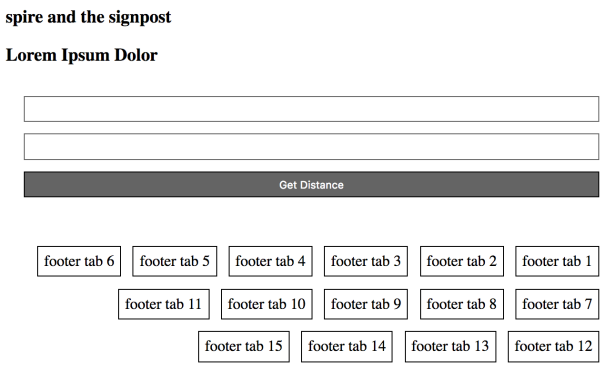

Figure 4: CSS Flexbox - flex direction reverse

`flex-flow` shorthand We may also combine *direction* and *wrap* into a single declaration, `flex-flow`, which will now contain values for both `row` and `wrap`.

e.g.

```
#tabs {
  flex-flow: row wrap;
}
```

Flex item properties We may also define properties for flex items, thereby controlling aspects of their rendering.

For example, we may use the following properties

- `flex-basis`
- `flex-grow`
- `flex-shrink`

Such properties may commonly be used to control how available space is used for flex items.

In effect, such properties may be used to modify how *available space* is apportioned for the flex items. Each property defines a different option to use this available space.

For example, if we have four flex items, each with a defined width of 100px, in a container of width 600px, there will 200px of available space. The default properties for such flex items will then place this 200px remaining space after the four rendered flex items.

However, we may use the above properties to modify whether the flex items grow, shrink &c. relative to the available space.

flex-basis We may use the `flex-basis` property to define the space a flex item leaves, in effect as remaining available space.

The default value for this property is `auto`, which forces the browser to check if the flex items have an existing set size. If a size has been defined, such as `150px`, this will become the value for `flex-basis`.

If there is no explicit size value for an item, its content will be used to determine the value of the `flex-basis` property. Each item will occupy the space required for the content, for example if we only declare an initial `display:flex` property and value.

flex-grow As noted, `flex-basis` acts as a default sizing guide for flex items. If we define the property `flex-grow` , and a positive integer value, we may also cause the item to stretch to fill the available space.

The stretching of the flex item will occur along the main axis, or as a proportion of the available space if there are other, similarly defined flex items.

For example, if we define a `flex-grow` value of `1` for each flex item in a container, then the available space will be distribute evenly, allowing the items to stretch to fill the main axis.

Such values may also be modified to represent proportional distribution of space for the flex items.

For example, a value of `2` for the first flex item, and a value of `1` for the remaining items. The first item will receive twice as much proportional space on the main axis compared to the other flex items.

flex-shrink By contrast, `flex-shrink` may be used to define a positive value to shrink a flex item relative to its flex basis.

This option is particularly useful for dynamic content in a container with limited available space.

We may also define different values per flex item, in a similar manner to `flex-grow` . Again, this is a useful option relative to different content types.

The minimum size of an item will, however, be considered as the item shrinks relative to the main axis. Whilst this behaviour may appear less consistent than `flex-grow` , it is necessary to ensure usability for this property.

Sizing of flex items For each flex item, we may need to specify apportioned space in the layout.

If we wanted to set space as an equal proportion for each flex item, we may add the following to a child item ruleset,

```
div.fTab {
  flex: 1;
}
```

This defines each child flex item `<div class="fTab">` to occupy an equal amount of space, after considering margin and padding, within the given row.

n.b. this value is proportional, so it doesn't matter if the value is 1 or 100 &c.

However, we may define additional flex proportions for specific child items. e.g.

```
div.fTab:nth-child(odd) {
  flex: 2;
}
```

This means that each odd *flex-item* will now occupy twice the available space in the current direction.

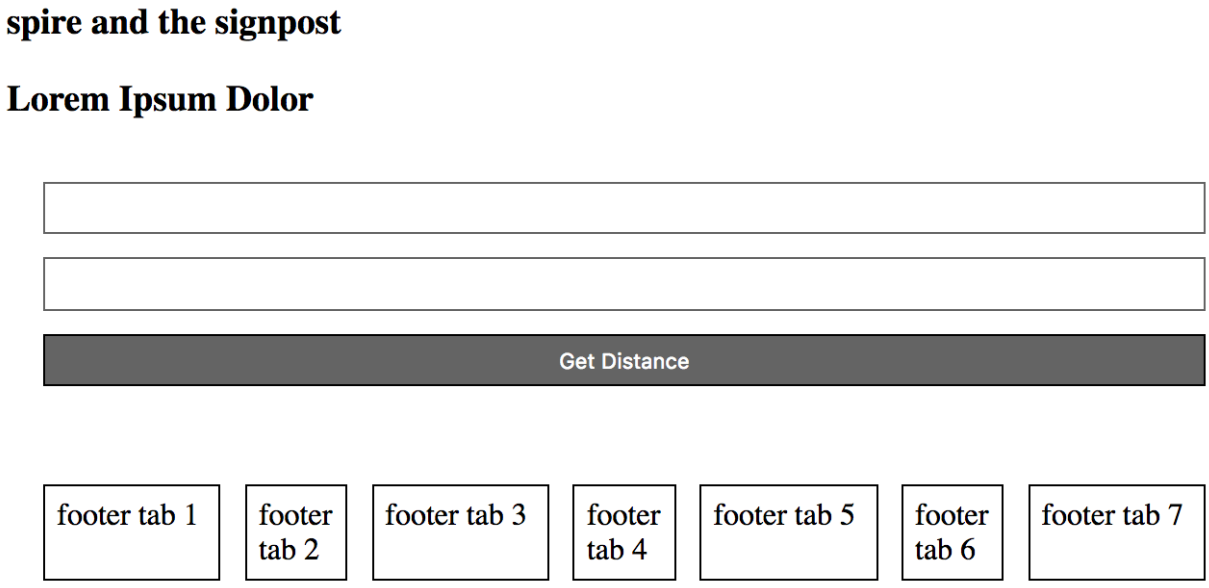

Figure 5: CSS Flexbox - flex item sizing

minimum size We may then set a minimum size for a flex item, e.g.

```
div.fTab {
  flex: 1 100px;
}
```

or a relative unit for the size,

```
div.fTab {
  flex: 1 20%;
}
```

This means each flex item will initially be given a minimum of `20%` of the available space. Then, the remaining space will be defined relative to proportion units.

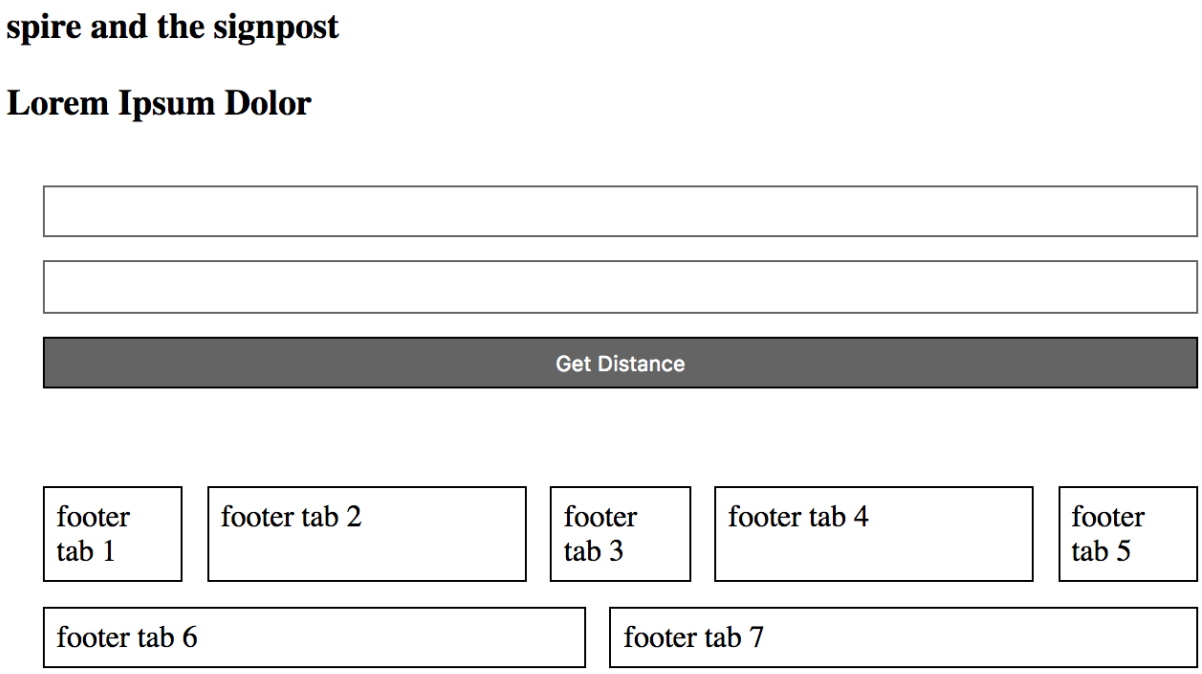

Figure 6: CSS Flexbox - flex item sizing - minimum size

shorthand values As noted for flex item sizes, we may also use a shorthand notation for flex properties in CSS rulesets.

We may commonly combine values for `flex-grow` , `flex-shrink` , and `flex-basis` for the single property `flex` .

For example, we may values for these properties as follows

```
.fTab {
    flex: 1 1 auto;
}
```

This sets values in a specific order corresponding to `flex-grow` , then `flex-shrink` , and lastly `flex-basis` .

If we omit the final value, then `flex-basis` will use the default value.

We may also use various pre-defined values for the `flex` property. These include

- initial - resets values to initial Flexbox value for each property
  - same as `flex: 0 1 auto`
- auto - same concept as `initial` , but items may also shrink as needed to avoid overflowing
  - same as `flex: 1 1 auto`
- none - creates inflexible *flex* items, which may not grow or shrink but layout using `flex-basis: auto`
  - same as `flex: 0 0 auto`
- a positive number - items may grow and shrink, but begin with `flex-basis: 0` as starting point
  - e.g. `flex: 1`
  - same as `flex: 1 1 0`

Flex item alignment Flexbox also allows us to define alignment for flex items in each flex container, relative to the main and cross axes.

We may use the following properties to help define alignment for items in a flex container,

- `align-content` - use to control space between lines relative to the cross axis
- `align-items` - control flex items on the cross axis
- `align-self` - use to control a single flex item on the cross axis
- `colum-gap` , `gap` , `row-gap` - use to create gaps or gutters between flex items
- `justify-content` - use to control all flex items on the main axis

For example, we might want to specify a centred alignment for flex items,

```
#tabs {
  flex-direction: row;
  flex-wrap: wrap;
  align-items: center;
}
```

So, `align-items: center` will cause the flex item in the flex container to be centred along the cross axis. However, they'll still maintain their basic dimensions.

The default value for `align-items` is `stretch` , which will cause flex items to stretch and fill the height of a flex container by default. Such default *stretching* may be determined by the height of the tallest item or a custom height specified in CSS, for example.

We can also modify the value for `align-items` to either `flex-start` or `flex-end` . As expected, such values will align flex items to either the start or end of the cross axis.

So, we may use the following values to control alignment with the `align-items` property,

- `baseline`
- `center`
- `flex-end`
- `flex-start`
- `stretch`

override align per flex item As with `flex` , we can also override alignment per flex item. Using the `align-self` property, we may add a value for positioning.

In effect, we may use `align-items` to set a *group* value for `align-self` on all of the flex items in a container. Then, we may use `align-self` to explicitly define a value for a single defined item.

e.g. a sample declaration might be as follows

```
div.fTab:nth-child(even) {
  flex: 2;
  align-self: flex-end;
}
```

`align-self` accepts the same values as `align-items` and a value of `auto` , which will reset the value to the group's container.

justify content for flex item We may also specify `justify-content` for flex items in a flex container.

This property allows us to define the position of a flex item relative to the main axis.

The default value is `flex-start` , and we then modify it relative to one of the following

- `flex-end` - aligns items initially from the end of the container
- `center` - aligns items from the centre of the container
- `space-around`
  - distributes each flex item evenly along main axis with space at either end
- `space-between`
  - same as `space-around` without space at either end
  - `space-evenly`
    - items aligned with a full-size space on either end

As expected, the default value `flex-start` will align items to the start of the container.

alignment per axis In effect, we may define alignment relative to each axis using a specific declaration.

For example, for the main axis we may use `justify-content` and for the cross axis we use `align-items` .

We may also choose to modify the default main axis using the property `flex-direction` . We may explicitly set the value for this property to either `row` or `column` , which will also influence the rendering of flex items and the defined properties.

Such alignment, as noted above, is, of course, *writing mode* aware. For example, if we define the value of `justify-content` as `start` , and the writing mode is left to right for English, the rendered flex items will, of course, begin on the left of the container.

However, if we then update the writing mode for Arabic, Japanese &c. the value `start` for the flex items direction will adapt to match the required initial start position and flex item rendering.

alignment on cross axis In addition to aligning items relatives to the space defined by the container, we may use the property `align-content` to control distribution of this space between rows.

To allow this property to work as expected, however, we need to ensure there is sufficient available height in the overall container. In effect, this will include excess height beyond the space required to render the flex items using defaults.

The `align-content` property will then determine how to use this extra space based upon the chosen value. These values include the same options defined for the `justify-content` property.

Order flex items We may also modify the layout order of flex items without directly changing the underlying source order. In addition to the previously mentioned reverse options, `column-reverse` and `row-reverse` , we may also target individual flex items and specify their visual order in the rendered container.

n.b. yet another modification we can't do with traditional CSS layout options…

We use the following pattern to specify order,

```
div.fTab:first-child {
  order: 1;
}
```

The `order` property is intended to allow items to be arranged in ordinal groups. Such groups are assigned a numerical value, and then the group is rendered in the order defined by such values with lowest values first.

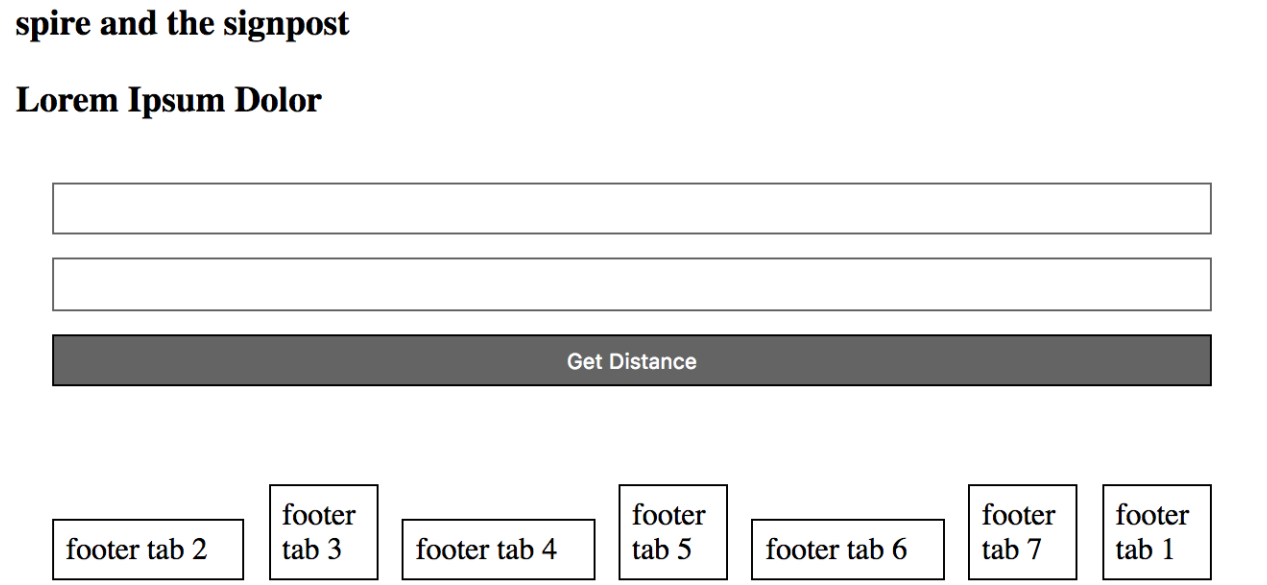

Figure 7: CSS Flexbox - flex item order 1

In this example, the first flex item will now move to the end of the tab list.

This is due to the default order for flex items. By default, all flex items have an `order` value set to `0` .

So, the higher the `order` value, the later the item will appear in the list &c.

Items with the same order will revert to the order in the source code.

It's also possible to ensure that certain items will always appear first, or at least before default `order` values, by using a negative value for the `order` declaration.

e.g.

```
div.fTab:last-child {
  order: -1;
}
```

Nesting flex containers and items Flexbox can also be used to create nested patterns and structures.

For example, we may set a flex item as a flex container for its child nodes.

e.g. we might add a banner to the top of a page,

```
<section id="banner">
  <header id="page-header">
    <h3>spire and the signpost</h3>
    <h5>point to the stars...</h5>
  </header>
  <section id="search">
    <input type="text" id="searchBox"/>
    <button id="searchBtn">Search</button>
  </section>
</section>
```

For this HTML, we may set `#banner` , `#page-header` , and `#search` as flex containers. e.g.

```
#search {
  display: flex;
}
```

We may then specify various declarations for `#search` , e.g.

```
#search {
  display: flex;
  flex-direction: row;
  flex: 2;
  align-self: flex-start;
}
```

which will include values for itself and any child elements.

So, if we then add some rulesets for the nested flex items, e.g.

```
#searchBox {
  flex: 4;
}

#searchBtn {
  flex: 1;
}
```

we get a simple proportional split of `4:1` for the input field to the button.

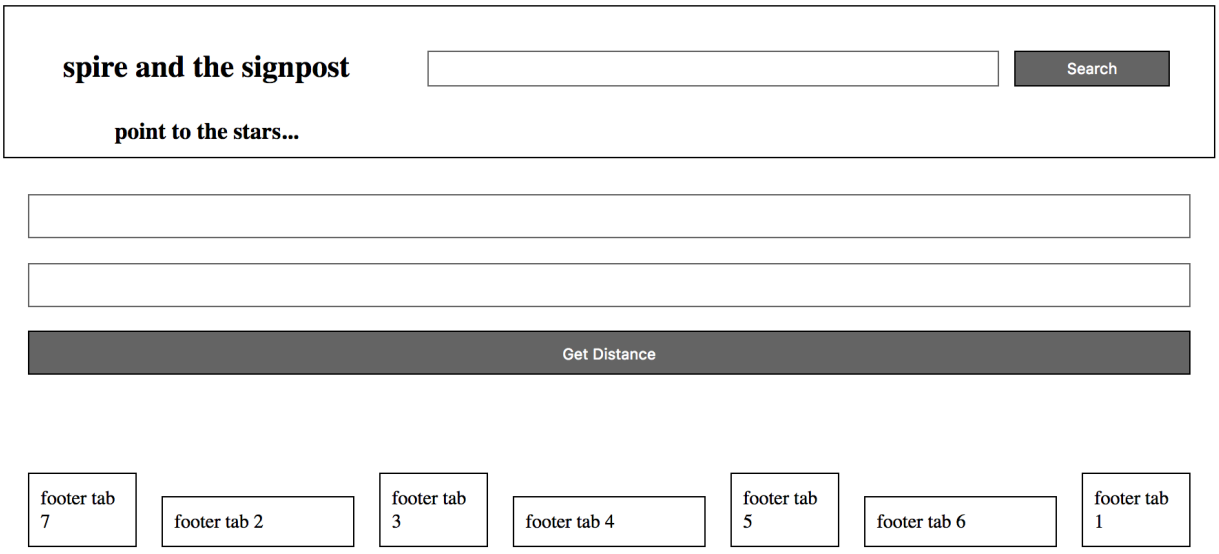

Figure 8: CSS Flexbox - nested flex containers

References

- MDN - CSS Flexbox
- MDN - Relationship of Grid Layout to other layout methods